




GPU accelerated CAD to inspection data deviation colormap generation

Venu Kurella ^a, Bob Stone ^b and Allan Spence ^a

^aMcMaster University; ^bOrigin International Inc.

ABSTRACT

Inspection of stamped sheet metal parts can require CAD to inspection data comparison rates exceeding 1 million points per second. With current serial CPU algorithms, comparing the nominal CAD to the digitizer data can take several minutes. The thousands of cores available with Graphical Processing Units (GPUs) therefore offer an attractive alternative. This paper describes GPU accelerated algorithms and memory management optimization for both point-facet and facet-facet CAD to inspection point cloud data deviation colormap generation. Computation time reduction from over 4 minutes to 2 seconds – a speed-up of 124X – was achieved. In collaboration with an industry partner, the result is implemented as a practical Windows compatible DLL that can be deployed to the factory floor.

KEYWORDS

Graphical processing unit; parallel computing; point clouds; deviation map

1. Introduction

Sheet metal stamping production rates approach one part every second [4, 5]. With increasing demand that comprehensive geometric quality conformance information be approved by final assembly plant management prior to shipment, the associated digitizing analysis of millions of points requires extremely fast algorithms. For example, an industrial blue LED snapshot sensor 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 microcomputer CPUs. This paper investigates the alternative of using massively parallel Graphical Processing Unit (GPU) hardware.

Use of this hardware exploits the parallel GPU architecture 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 were not intended 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.

Early GPU computing required researchers to mask arithmetic operations as graphical tasks to perform computations on CAD parts [13] or tool paths [3]. The NVIDIA CUDA programming language [17] revolutionized GPU computing. It led to applications such as rendering [7], filtering [8] and collision detection [14]. Sheet metal strain measurement was reported by Kinsner et al. [10]. Other computational applications such as distance queries between NURBS surfaces [12] and feature-fitting of geometric primitives [16], showed respectively 300X and 18X speed-up. While 2.5X - 1000X speed-up achievements are reported in literature [11], 20-30X is considered worthwhile. Erdos et al. [6] suggest GPU computing for fast mapping of CAD with point cloud data. Iterative Closest Point (ICP) like registration methods [1, 2] have already been implemented on GPUs [15]. The subsequent brute force facet-by-facet deviation estimation of the registered data is very computationally intensive. In this work, we investigated speed-up of point-facet matching (PFM) and facet-facet matching (FFM) algorithms that estimate deviations and report them as informational colormaps. A similar smallest sphere distance finding algorithm showed 5X speed-up with naïve GPU implementation [9]. We show that, using a Tesla K40 GPU,

careful algorithm optimization and advanced memory management delivers an impressive 124X speed-up.

2. Algorithm challenges and solutions

PFM and FFM both accept faceted CAD model data as input. Actual part data from scanners is provided as facets (for FFM) and points (for PFM). Output is the average (or first for FFM) deviation between a model facet and its matching actual points (or facet for FFM) displayed as a colormap. Details of the computations that take place on every model facet in the matching algorithms are discussed in Table 1. The registration algorithm is based on the well-known ICP [1] method. The transformation matrix is initiated by manual matching of a few widely separated points chosen from both the digitizer and facet data. Because of the expected high number of digitizer points as compared to the size of the CAD facets, the algorithms begin by transforming the facets into the digitizer part coordinate system. This is followed by binary search of digitizer points/facets to find the actual point/facet, j , nearest to the CAD facet. The final step is a refined search using matching parameters, in the neighborhood of j , to find matches whose deviation is within a given threshold. The first /average of the deviations is calculated and reported as color map. As originally implemented, both the binary and neighborhood search algorithms are intricate and time consuming due to loops, branches and many memory and function calls. The complexity is $O(m * \log(n))$ where m and n are the number of model facets and actual points/facets respectively. Actual facets/points are sorted based on their distances from origin, thus, facilitating binary search which takes $O(\log n)$ time. Since the refined matches (if any) are found in close vicinity of the binary search result, the overall computational time of the algorithm per model facet remains $O(\log n)$.

2.1. Challenges

The existing serial inspired methods face GPU challenges in both algorithm and memory implementation.

Conditional tests and branches (Figure 1(a)) exhibit poor instruction level parallelism (ILP). To address this, filtering flags were used within the GPU algorithm to bundle tests into a single branch (Figure 1(b)). As an example, a snippet of the facet-facet matching algorithm is discussed below.

Computation 1: Z distance between CAD facet and inspection facet

Filtering Condition 1: Is distance less than threshold

Computation 2: Dot product of normal vectors of CAD facet and inspection facet

Filtering Condition 2: Dot product > 0 (normal directions within 90 degrees)

Critical Condition 1: Does model facet normal intersect the actual facet (this condition is critical as further calculations cannot proceed without the point of intersection)

Computation 3: Euclidean distance between the center of the actual facet and the point of intersection

Data structures were simplified and matching parameters were bundled for continuous memory accesses.

2.2. Experimental set-up

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, running under Microsoft Windows 7. Results from a single core of the Intel CPU were compared with NVIDIA Tesla 2880 core K40. Programming was done in Visual Studio 2013 with NVIDIA Nsight 4.1 and CUDA 6.5. Scanned part data is provided as input, and a CAD model of the part was used to generate the facets. Iterations produce colormap deviation values of every model facet. The maximum facet edge length parameter was

Table 1. PFM and FFM matching algorithms. Here *actual* means actual facet and actual point for FFM and PFM respectively. *Model* means model facet. Vicinity is a fixed distance parameter.

Step	FFM	PFM
1	<i>Transform</i>	<i>model to actual</i>
2	<i>Binary search</i>	Find the closest <i>actual</i> based on its distance from origin
For all <i>actuals</i> in the vicinity of the closest <i>actual</i>		
3	<i>Refined tests</i>	Is the <i>actual's</i> location and orientation close to that of the <i>model</i> ?
4	<i>If refined tests passed</i>	Does the <i>model</i> normal pierce the <i>actual</i> ?
5	<i>Is the distance less than</i>	Estimate the projected distance between the <i>model</i> and the <i>actual</i> Deviation threshold <i>model</i> radius
6	<i>If false</i>	Move to the neighboring <i>actual</i> in the vicinity and go to step 3
7	<i>Exit when</i>	First deviation found Average of all the deviations found

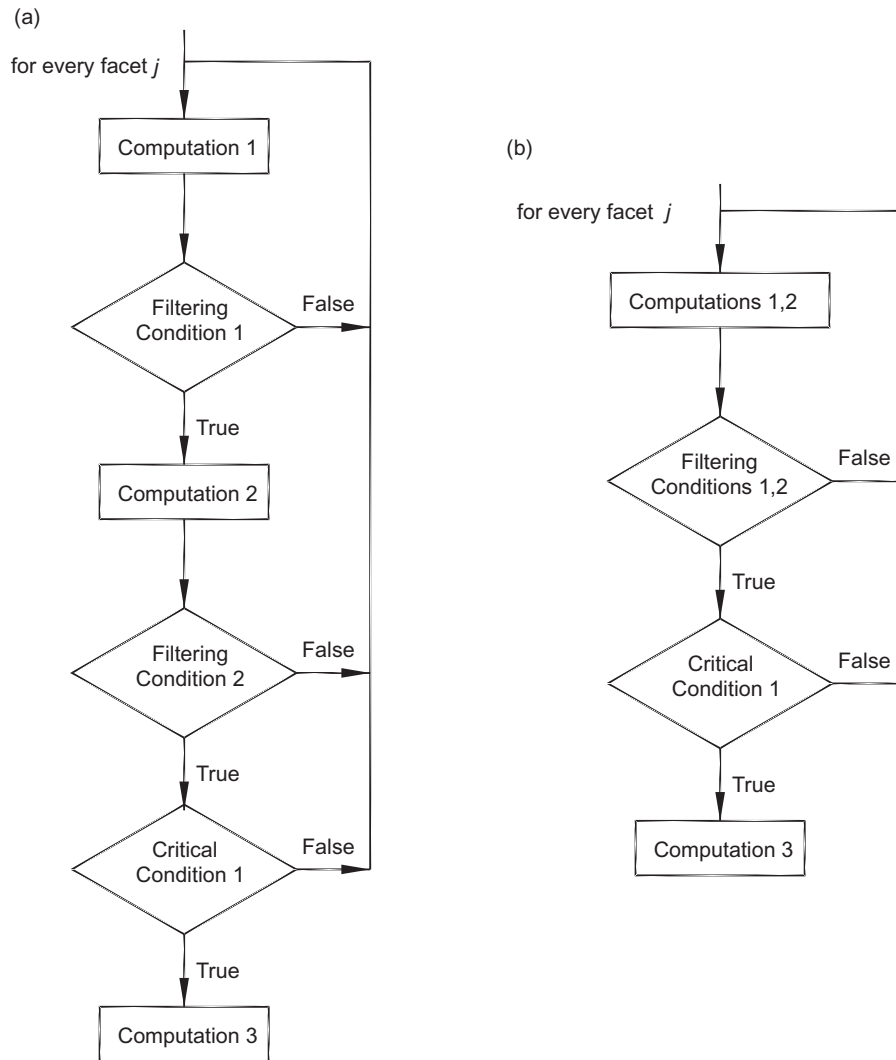


Figure 1. Instruction Level Parallelism (ILP) for FFM: (a) CPU branches in neighborhood search with filtering and critical conditions, and (b) Improving ILP on the GPU by bundling conditions.

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

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

3. Point-facet matching

3.1. Initial experiments and results

For PFM, scanned part data (Figure 2(a)) input contains 424307 points. Each sample point includes the direction to the scanner to get an approximate local part orientation. Since this information is not as strong as the direction of surface normal at that point, the neighborhood search in PFM finds all possible close matches within a given threshold. The result is the average of the deviation

of the model facet with each of the matched actual points (Figure 2(b)). Three sets of test data were generated using model facet sizes (with maximum facet edge lengths) of 2 mm, 1 mm and 0.5 mm leading to 0.1 Million, 0.3 M and 1 M model facets respectively. After improving ILP, experiments on the K40 GPU achieved speed-ups of 8X, 22X and 46X. Performance analysis was conducted to understand the bottlenecks and to investigate the scope of further acceleration.

3.2. Performance analysis

To understand the bottlenecks in the naïve GPU implementation, performance was analyzed using the NVIDIA Nsight tool in Visual Studio. GPU multiprocessors execute the computational instructions as sets of 32 threads called warps. On every multiprocessor, only a few of the active warps are eligible for execution. Profiling showed

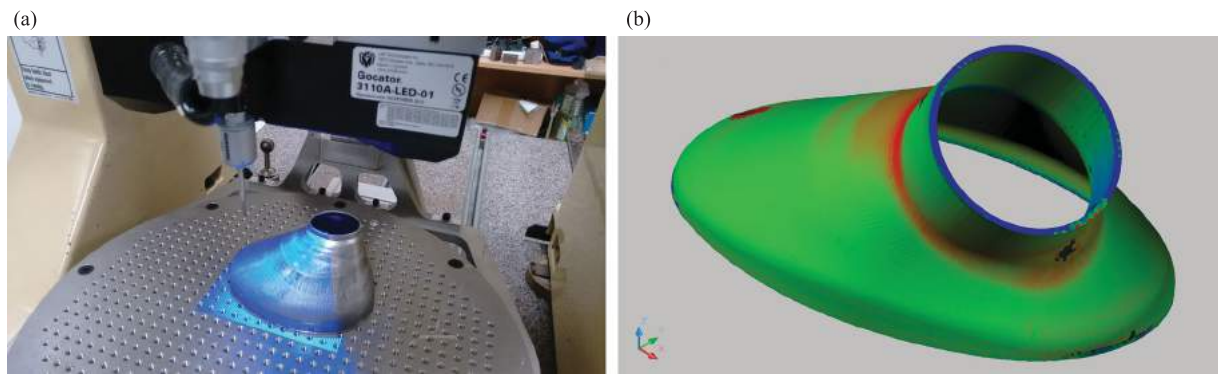


Figure 2. Point-Facet Matching: (a) Actual part, and (b) Points fit to its model to produce a colormap of average deviations.

that, at any given instance, active warps were stalled, i.e. not eligible to move to the next instruction, due to the following reasons (Figure 3):

- Memory throttle: It occurs when there are large number of pending memory operations.
- Memory dependency: When load and store cannot happen because resources are unavailable or are being completely utilized.
- Pipe busy: It means that the load/store and arithmetic pipelines are not available for computations.
- Execution dependency: This is because input for an instruction is not available.

Hence the existing algorithm structure does not use the GPU cores effectively. 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 high 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 encountered. No speed-up was observed because there is not enough computation between the (many) memory accesses to hide the latency. As seen from the causes (Figure 3), memory throttle and dependencies are the leading causes. The high utilization of the memory resources is keeping the pipeline busy. Common approaches to tackle these issues

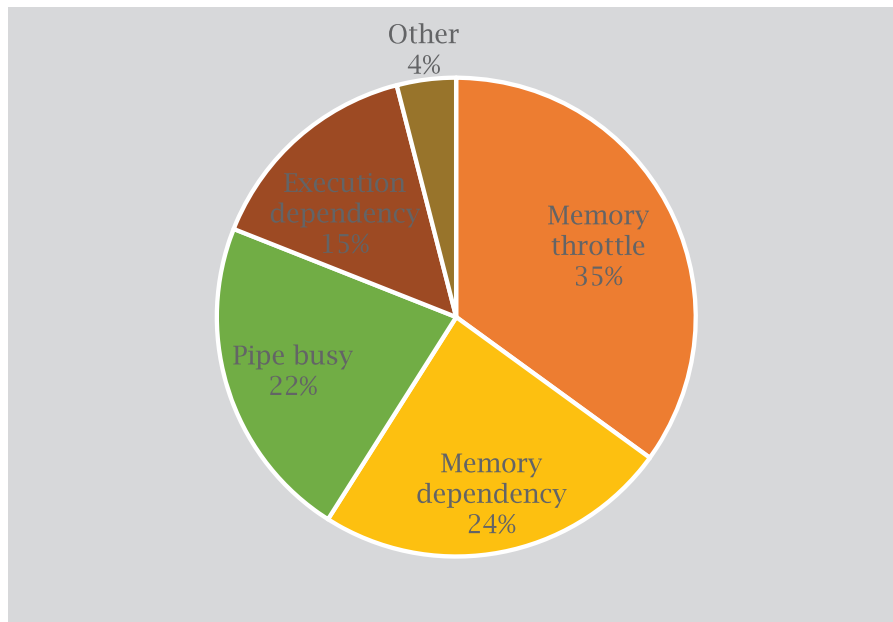


Figure 3. Results of performance analysis showing the warp stall reasons.

are bundling the memory operations, lowering memory access times, increasing ILP and allowing contiguous memory accesses. Hence to improve the performance, either the algorithm parallelism has to be increased (such as by complete looping), or the memory accesses times have to be lowered (such as using texture memory). The potential of these improvements is explored below.

3.3. Complete looping

A simple way to achieve near-perfect parallelism is to eliminate branching and loop over all points, instead of a specific neighborhood (Table 1). 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 over all of the 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). Due to these reasons, the complete looping approach failed to improve speed-up. Therefore, as discussed in the next section, attempts were made to lower memory access times using texture memory (Table 2).

3.4. 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

Table 2. Performance parameters before and after implementation of texture memory.

	Naïve	With texture memory
Warp occupancy	5.17%	20.03%
GFLOP/s	26.09	99.64
Instructions per clock executed (in a GPU streaming multiprocessor)	0.14	0.66

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 as evident from the significant improvement in performance parameters shown in Table 2. The results discussed in the next subsection show fraction-of-a-second computation times for two of the three cases.

3.5. Results

The computation times and speed-ups are summarized in Table 3. While naïve usage of memory, itself, showed an impressive 46X speed-up, texture memory further boosted it to 124X. It can be noticed that the 0.3 M facets case takes time less than the 0.1 M case. This is likely because as the number of model facets gets close to the actual facets number (~ 0.4 M), fewer texture cache misses occur leading to a higher speed. A practical colormap resolution is achieved when the number of CAD facets is nearly the same as the number of cloud points.

4. Facet-facet matching

4.1. Setup

The part used for performing FFM experiments is shown in Figure 4. Its actual part data contains 702429 facets each storing its normal direction. Unlike PFM where only the point and scanning direction is available, here the facet and normal information can be used to find a tighter match using various position and orientation tests. Once a match is found, the neighborhood search exits returning the deviation. This means for every model facet, FFM can find deviation sooner than PFM. While this might be computationally faster, it should be noted that such actual part facet data can only be obtained by rigorous and time consuming pre-processing of raw noisy point clouds data from scanners.

Table 3. Point-Facet Matching: Computation time (seconds) and speed-up (s-u) of Tesla K40.

Number of model facets	Max. facet edge length (mm)	Intel Xeon E5 single core	NVIDIA Tesla K40			
			Naïve memory		With texture	
			Time	s-u	Time	s-u
103,966 (~ 0.1 M)	2.000	22.6291	2.827	8	0.964	23
345,592 (~ 0.3 M)	1.000	75.1014	3.347	22	0.883	85
1,188,408 (~ 1 M)	0.500	259.126	5.676	46	2.098	124

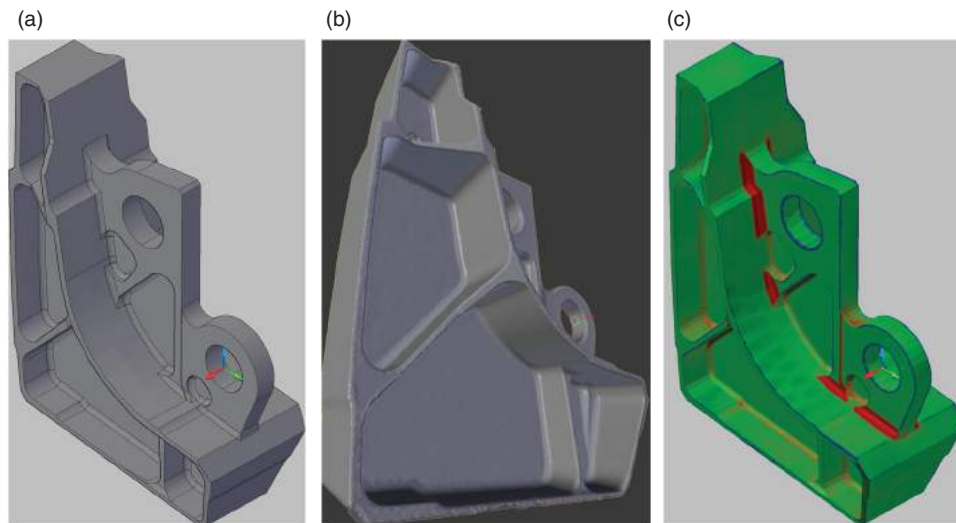


Figure 4. Facet-facet matching: (a) Model, (b) Actual data, and (c) Colormap.

Table 4. Facet-Facet Matching: Computation time (seconds) and speed-up (s-u) of Tesla K40.

Number of model facets	Max. facet edge length (mm)	Intel Xeon E5 CPU single core	NVIDIA Tesla K40			
			Naïve memory		With texture	
			Time	s-u	Time	s-u
89,850 (~ 0.1 M)	0.157	13.596	1.876	7	1.154	12
322,010 (~ 0.3 M)	0.078	49.929	3.151	16	1.993	25
1,147,568 (~ 1 M)	0.039	174.558	7.381	24	6.343	28

4.2. Results

As with PFM, the maximum facet edge length parameter of the part CAD model (Figure 4(a)) was varied to generate three testing data sets. Preliminary experiments on FFM showed an impressive 7X, 16X and 24X accelerations for the three datasets respectively (Table 4). Learning from PFM, texture memory implementation is made without attempting complete looping. This boosted the speed-ups further to 12X, 25X and 28X respectively (Table 4). For example, looking at the 1 million facets case, FFM that would take about 3 minutes on the CPU can be completed in about 6 seconds using the K40 GPU.

It can be seen that the final speed-ups are lower than those observed for point-facet matching. This is because FFM algorithm has more branches and function calls. There is also at least twice as much work on the CPU side during copying data to basic texture data types e.g. facet vertex location data is needed for FFM but not PFM. Hence PFM is more parallelizable than FFM on the GPU.

5. Analysis and conclusions

5.1. Consistency check

Experiments were conducted to check the consistency of the GPU results with time. PFM and FFM were iterated 100 times on the GPU, resetting after each iteration. The

three facet sizes were studied and iterations output deviation values of every model facet. Maximum absolute error over the iterations is estimated. Sample maximum absolute error distribution for the 1 M facet case in PFM is shown in Figure 5. In all the three facet experiments, it was found to be of the order of $1E-6$ for both PFM and FFM. Since the input variables are declared as single precision floats (6 significant digits), this is reasonable.

5.2. CPU-GPU result comparison

The deviation results of PFM and FFM algorithms from the GPU and the CPU were also compared. Maximum absolute differences for each of the three test datasets are presented in Table 5. The differences for PFM are within the limits of single floating point precision. The gap in facet-facet matching cases is relatively high – likely because it uses more precision sensitive tests compared to point-facet matching and does not average deviations. Results are especially affected when the precision sensitive tests match poorly oriented actual and model facets, for example those on the corners of the geometry. Figure 6 shows the absolute difference on log scale for 0.3 M dataset in FFM. Other causes for the differences could be different rounding methods and order of the steps of computation executed by the CPU and the GPU. As it can be seen from the plot, there is a drift in the differences,

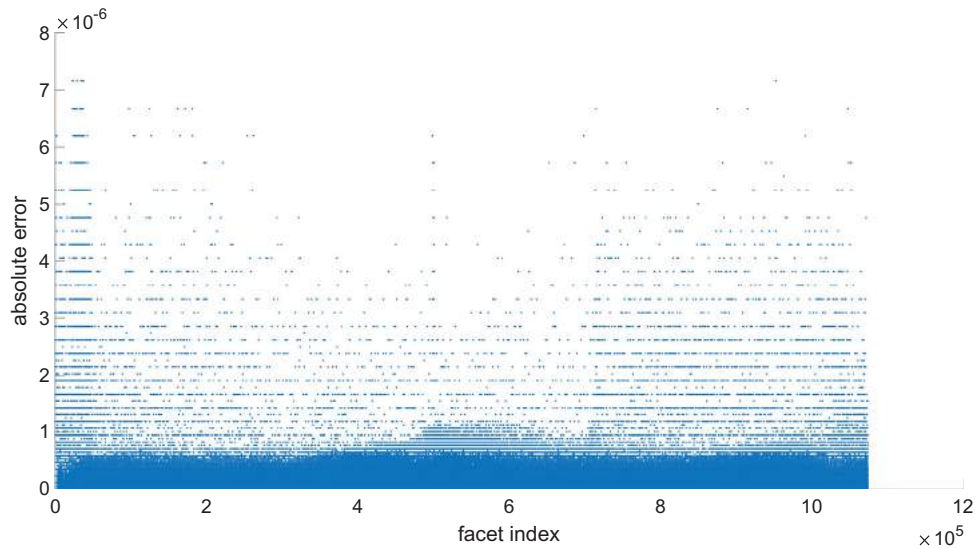


Figure 5. Consistency of PFM deviation results over 100 iterations for the 1 M model facet dataset.

Table 5. Maximum absolute differences of deviation results from the CPU and the GPU.

	# model facets	0.1 M	0.3 M	1 M
Order of maximum absolute error	Facet-Facet Matching	1E-3	1E-4	Two mismatches, remainder 1E-3
	Point-Facet Matching	1E-6	1E-6	1E-6

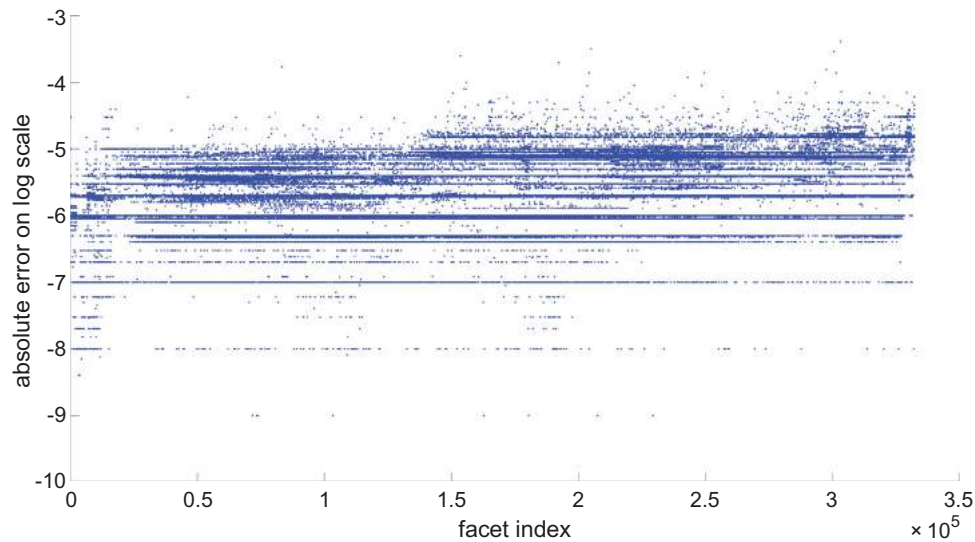


Figure 6. Absolute differences between the CPU and GPU deviation results for 0.3 M model facet dataset in FFM.

with values increasing with facet index. This is because the actual facets are ordered with respect to their distance from the origin. Points nearer the origin are stored with more precision than those farther from the origin resulting in the drift.

5.3. Industry software implementation

The research was motivated by a collaboration initiative involving Origin International Inc. to improve the speed of their CheckMate measurement analysis software. The

existing CPU method, initially developed for a low number of touch probe data points, was impractical for the large point clouds measured with non-contact digitizers. Accordingly, the implementation was realized using a Windows DLL that provided a bridge between the proprietary CheckMate source code and the newly developed parallel GPU code. Origin provided agreed upon function call interfaces that could switch between existing serial CPU and the newly developed parallel GPU algorithms. This approach offers a win-win university-industry collaboration.

5.4. Conclusion

In conclusion, this work demonstrates the feasibility and the method of accelerating point-facet and facet-facet matching of a dense and complex data, including colormap generation. It delivers the product as a practical and industrially applicable library compatible with Microsoft Windows.

5.5. Future work

Future work will focus on estimating normal vectors for the point cloud data obtained from the multi-sensor inspection system discussed in previous work [18], and the subsequent estimation of deviations using the algorithm discussed in this paper. The normal directions used in point-facet matching denote the direction of scanning.

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.

ORCID

Venu Kurella  <http://orcid.org/0000-0003-3547-7266>

Bob Stone  <http://orcid.org/0000-0002-6917-969X>

Allan Spence  <http://orcid.org/0000-0002-6835-6498>

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://dx.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://dx.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] Dolcemascolo, Darren: *Improving the Extended Value Stream: Lean for the Entire Supply Chain*, Productivity Press, New York, NY, 2006, ISBN-13 978-1-56327-333-9.
- [5] Dallan, Andrea: *Punching and stamping: How to compare production cost*, Technical Report, Dallan Rollformers and Systems, 2005. <http://www.dallan.com/assets/allegati/pdf/punching-and-stamping.pdf>
- [6] Erdos, G.; Nakano, T.; Vancza, J.: Adapting CAD models of complex engineering objects to measured point cloud data, *CIRP Annals*, 63(1), 2014, 157–160. <http://dx.doi.org/10.1016/j.cirp.2014.03.090>
- [7] 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.
- [8] 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>
- [9] 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://dx.doi.org/10.1080/16864360.2015.108418>
- [10] Kinsner, M.; Spence, A.; Capson, D.: GPU accelerated sheet forming grid measurement, *Computer-Aided Design and Applications*, 7(5), 2010, 675–684. <http://dx.doi.org/10.3722/cadaps.2010.675-684>
- [11] Kinsner, M.: *Close-range machine vision for gridded surface measurement*, Ph.D. Thesis, McMaster University, Hamilton, Canada, 2011. <http://hdl.handle.net/11375/11075>
- [12] 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.
- [13] 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://dx.doi.org/10.1080/16864360.2007.10738517>
- [14] 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://dx.doi.org/10.3722/cadaps.2011.849-85>
- [15] 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>
- [16] 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. <http://dx.doi.org/10.1016/j.jmsy.2008.07.004>
- [17] Sanders, J.; Kandrot, E.: *CUDA by example: an introduction to general-purpose GPU programming*, Addison-Wesley Professional, Boston, MA, 2010, ISBN-13 978-0-13-138768-3.
- [18] Xue, K.; Kurella, V.; Spence, A.: *Multi-Sensor Blue LED and Touch Probe Inspection System*, *Computer-Aided Design and Applications*, 2016. <http://dx.doi.org/10.1080/16864360.2016.1168226>