Scaling Point Set Registration in 3D across Thread Counts on Multicore and Hardware Accelerator Platforms through Autotuning for Large Scale Analysis of Scientific Point Clouds

Piotr Luszczek, Jakub Kurzak, Ichitaro Yamazaki, David Keffer
University of Tennessee
1122 Volunteer Blvd., Suite 203
Knoxville, Tennessee 37996-3450, USA

Jack Dongarra
University of Tennessee
1122 Volunteer Blvd., Suite 203
Knoxville, Tennessee 37996-3450, USA
Oak Ridge National Laboratory, USA
Manchester University, UK

***Abstract*—In this article, we present an autotuning approach applied to systematic performance engineering of the EM-ICP (Expectation-Maximization Iterative Closest Point) algorithm for the point set registration problem. We show how we were able to exceed the performance achieved by the reference code through multiple dependence transformations and automated procedure of generating and evaluating numerous implementation variants. Furthermore, we also managed to exploit code transformations that are not that common during manual optimization but yielded better performance in our tests for the EM-ICP algorithm. Finally, we maintained high levels of performance rate in a portable fashion across a wide range of HPC hardware platforms including multicore, many-core, and GPU-based accelerators. More importantly, the results indicate consistently high performance level and ability to move the task of data analysis through point-set registration to any modern compute platform without the concern of inferior asymptotic efficiency.**

***Keywords*-Portable performance engineering; Point set registration; Autotuning with code generation**

I. Introduction

The algorithms for registration of point sets are commonly used in many aspects of computer vision. But in many areas of science, these methods can be used for analysing data arriving from hardware instruments. In particular, such methods are necessary in order to produce unambiguous descriptions of atomic scale structures from large data sets originating in Atomic Probe Tomography (APT) [1], [2] and multimodal electron microscopy (EM) [3], [4]. APT can generate data sets that include as many as 10^7 atoms in a single image acquisition frame. Work is underway for electron microscopes to relay time-resolved frames, resulting in an explosion of data that truly puts the analysis of the output of these analytical techniques of registration squarely within the realm of "big data." On the technical side, the goal is to be able to resolve both atomic identity and position. The incoming instrument data is in the form of sets of atomic (x, y, z) coordinates in three-dimensional (3D) space accompanied by identification of the atom type out of a handful of elements that are commonly fused and subsequently analyzed to discover their radial distribution functions and energy landscapes. Such data is in many ways similar, in its basic form, to visualization tasks but the registration of the points will be followed derivation of physics, chemistry, or material science profiles that inform the scientists of emergence properties of the analyzed samples. Fast and accurate derivation of optimal implementations of the registration algorithms is the subject of this paper.

In particular, we use Expectation-Maximization (EM) Iterative Closest Point (ICP), or EM-ICP for short. EM-ICP is a stochastic method for registration of surfaces. It improves issues found in other algorithms related to minimizing non-convex cost function. Other registration algorithms applicable for our problem sets and implementation methodology are given in Section II.

In its simplest mathematical form, registration of point sets X and Y may be expressed as:

$$\min_f \|f(X) - Y\| \tag{1}$$

where the points sets come from a 3D space:

$$X = \{x_1, x_2, \ldots, x_\ell\}, Y = \{y_1, y_2, \ldots, y_m\} \text{ with } x_i, y_j \in \mathbb{R}^3 \tag{2}$$

and the function f is taken to represent a combination of rotation, scaling, and translation:

$$f : X \mapsto R \times X + t \tag{3}$$

these restrictions result in transformation that is called *rigid registration* and is the main focus here but a more general *non-rigid registration* allows the transformation to be affine. This includes anisotropic scaling and skews. Further generalization is also possible and might allow unknown point set registration. Note that the basic formulation is usually assumed to be robust in the sense that it can handle correctly noisy input data with outliers and some of the points missing from the input data set.

In recent times, significant effort has gone into the evaluation of various techniques for characterizing local atomic environments [5]. To an extent, we follow this

978-1-5386-2715-0/17/$31.00 ©2017 IEEE

1: Start with $R^0 \leftarrow I$, $t^0 \leftarrow 0$
2: **loop**
3: Find closest point $\vec{y}_{i*} \in Y$ for each $\vec{x}_i \in X$:
$i* = \arg\min_{j=1,\ldots n_y} \left\| \vec{x}_i - R^{(k-1)}\vec{y}_j + \vec{t}^{(k-1)} \right\|$
4: Build the ordered correspondence $Y^* = \{y_{1*}, \ldots, y_{n_x *}\}$
5: Find rigid R^*, $\vec{t}^*$ to minimize MSE(X, Y^*)
6: $R^k \leftarrow R^*$, $\vec{t}^k \leftarrow \vec{t}^*$
7: **end loop**

Figure 1. Outline of the algorithmic steps of EM-ICP (MSE = mean square error.)

approach and adopt tools from image reconstruction in the field of computer visualization in order to build a highly-resolved atomic structure from a heavily defective data set such as one obtains from APT. From a mathematical point of view, in Equation (1), of most importance is the minimization of the Frobenius norm: $|| \cdot ||_F$. The norm is computed for a set of matrices representing the difference between a model reference configuration, $\mathbf{m}$, denoting the true, average local structure, and the local configuration (data), $\mathbf{d}_i$, around atom i, where for an APT experiment, i ranges from 1 to $I \approx 10^7$. The minimization problem then becomes the following:

$$\min_{R_i, P_i} \sum_{i=1}^{I} ||\mathbf{m} - P_i \mathbf{d}_i R_i||_F^2, \tag{4}$$

where each configuration has a unique permutation, P_i, and rotation, R_i, matrix (both real and orthogonal) in order to make it invariant to the arbitrary orientation and numbering generated by the experimental process. This general approach to alignment is called point set registration or, in this case, 3D-3D registration [6].

The simplified outline of the EM-ICP algorithm is shown in Figure 1. The initial guess for the transformation is an identity rotation/scaling matrix R and zero translation matrix $\vec{t}$ which are then consequently updated by minimizing Mean Square Error (MSE).

In this paper, we study performance engineering method of autotuning based on *benchtesting methodology*. This method combinatorially compounds the search space of tuning parameters and then subsequently prunes the said space combinatorially with a set of constraints. Both the tuning parameters and the constraints are provided by the user using the knowledge of the problem (3D registration in this paper) and the template of the implementation kernel (a parametrized version of the reference cod3). The user remains oblivious to the interaction of tuning parameters and constraints. This is because both are processed and subsequently inserted at the optimal place in the automatically generated code which explores the tuning space and prunes away large subsets in that space. This results in nearly additive (rather than multiplicative) compounding of the resulting search space. We show how this combination of techniques results in an automatically generated code that outperforms the manually optimized implementation and may be obtained in a sub-exponential time contrary to what the combinatorial explosion of space size might initially suggest.

II. Related Work

Iterative Closest Point (ICP) algorithm [7], [8] may be characterized by both simple implementation structure and a low computational cost. Over the years, both of these aspects have contributed to its popularity and spawned numerous variants [9], [10] including EM-ICP [11]. The Expectation Maximization (EM) algorithm for Gaussian Mixture Model (GMM) may be shown [12] to be equivalent to Robust Point Matching (RPM) algorithm [13] alternating soft-assignment of correspondences and point-set transformation. It is worth noting that RPM comes in multiple variants [14], [15], [16]. Finally, Coherent Point Drift (CPD) algorithm [17] performs non-rigid registration with a use of a regularizer.

Implementations of these methods are available in various forms on multiple hardware platforms. Commonly, a sequential code may be obtained with rare occurrences of enhancements for either multicore or hardware accelerated machines. Support for ICP is available, for example, in the Point Cloud Library (PCL) [18]. We use these codes, or the fastest representative (if available), as the basis for our implementation and then update them to the modern HPC software stack. We had to update to the current version of CUDA (from the available code that was using CUDA version 5) that supports the GPU platforms we used in our tests. Some implementations would work with as recent versions of CUDA as 6.5[1]. Our implementation is highly customizable and targets the most recent versions available and supported by NVIDIA: 7.5 and 8.0 with initial work towards compatibility with beta-releases of CUDA 9.0 (currently not widely available to the public).

III. Profiling and Performance Analysis

Our survey of existing codes for EM-ICP revealed that the freely available implementations[2] focus on visualization tasks and image processing workflows that are often optional in case of scientific instruments. Our focus is to provide very high ingest rates of the data coming from the hardware sensors and be able to process them in time before the long-term storage system becomes overwhelmed. Consequently, the support for High Performance Computing (HPC) techniques is poor and we faced the choice of retrofitting the codes for multithreading, modern accelerator libraries, and performance profiling or write our version from scratch. We chose the former and used this updated code as a reference

[1] One such implementation is ICPCUDA available at https://github.com/mp3guy/ICPCUDA.

[2] We did not consider commercial implementations for this study but only the codes that can be obtained under open source or educational license.

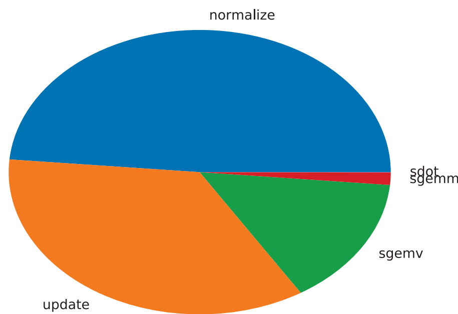

Figure 2. Performance profile of EM-ICP implementation in CUDA running on NVIDIA Kepler accelerator.

- Compiler: GNU `gcc`, Intel `icc`
- Runtime: GNU `gomp`, Intel `iomp`
- Number of threads: 1, ..., 40, ..., 70, ...
- Hyperthreading: yes, no (enabled with OpenMP or kernel affinity of threads)
- Interleaving: compact, spread, round-robin
- Main memory page mapping: round-robin, application-specific (first touch, random, ...)
- Software-exposed parallelism: collapse(2), collapse(3), ...
- Thread scheduling selection: static, chunk-based, dynamic, ...
- Parallel grain selection: chunk size
- Vectorization: 2, 4, 8, 16

Figure 3. Autotuing parameter space used for optimization of performance through OpenMP parallelization and directives.

point that served as the basis for optimization on a variety of modern HPC platforms that we describe in detail below.

As the first step in the performance engineering, we acquired an application profile through a dedicated run with an instrumented version the code. This allowed to identify the bottleneck portions of the code that may then be targeted with our autotuning methodology in order to maximize the potential speedup benefits. Figure 2 shows a typical performance profile on one of the tested devices. It is representative of the time breakdown that we observed on the other machines used in tests. The codes for point set registration require 32-bit floating-point arithmetic for computation and hence most of the code uses single-precision float data types and the profile from the figure uses that precision and changes to this precision will be noted explicitly. The sections of the profile correspond to the following steps of the algorithm from Figure 1:

- Calls to update are made in step 3.
- Calls to sdot are made in steps 4.
- Calls to sgemv are made in steps 4 and 6.
- Calls to normalize are made in step 5.
- Calls to sgemm are made in step 6

The common technique for efficient implementations is to off-load the compute-intensive parts of the code to highly tuned numerical libraries. In the case of results from Figure 2, the calls to NVIDIA CUBLAS make optimal use of the compute units (SGEMM) and the available memory bandwidth (SGEMV and SDOT). Unfortunately, once these sections of the code achieve the performance rate that is close to hardware-optimality, the other parts of the code become the main sources of slowdown. These two slowdown parts are named update and normalize in Figure 2. In terms of the operations from algorithm in Figure 1, they represent updates and normalization of the correspondence and error metrics throughout the iterations. Focusing on these two portions exclusively targets nearly 90% of the execution time across our tested hardware.

Once we identified the code sections that are important for performance engineering, we proceeded with defining the parameter space of available configurations that needs to be explored for generating code that would execute efficiently. Figure 3 shows a summary of that space for multicore hardware (GPU-specific considerations are presented below) with open source[3] and commercially developed[4] software stacks. The search space for autotuning optimization is multidimensional and heterogeneous in the sense that it includes different categories of parameters, namely: binary (for example use, hyperthreading or not use it), ordinal/categorical/enumeration (for example, compiler-version pairs: `gcc` 6.0, `gcc` 7.0, `icc` 2016, `icc` 2017), integer (for example, an integer range of loop blocking parameter values), and continuous (for example, cache reuse ratio). Theoretically, the entire search space may be represented by a Cartesian product of these variables which would result in combinatorial explosion in the size of the search space. In practice, the search space is much less regular due to a number of software and hardware constraints. For example:

- Optimization problem might be solved for the GNU compiler but is an issue for the Intel compiler.
- The results might change between different versions of the compilers: often new versions result in improvements but performance regressions are also possible.
- Internal interactions between the compiler, OpenMP runtime, and the processor firmware could further complicate the optimization.

We have developed Domain Specific Language (DSL) called

[3]Due to the fact that the GNU and LLVM compilers share the OpenMP runtime, we only considered GNU `gomp` but a LLVM-specific solution is under development.

[4]We only considered one commercial compiler but other choices are also possible, for example the PGI Group or Microsoft Visual Studio compilers that support a version of the OpenMP standard.

- Target occupancy level
  - NVIDIA provides occupancy calculator
  - Good performance is not equivalent to good occupancy but there exist occupancy thresholds that are necessary for achieving sufficiently high levels of relevant performance metric.
- Read coalescing: X-points only, Y-points only, both X and Y points
- Thread grid shape
- Thread block shape
- Thread mapping to data (application-level thread affinity):
  - grid-block: $\mathbb{G}^{i \times j} \times \mathbb{T}^{R \times C} \longrightarrow \mathbb{R}^{X \times Y}$: user data
- Number of partial warp (warp thread count smaller than $2^5 = 32$
- SM(X) streaming Unit utilization in number of CUDA cores as $\ell \times 2^k$

Figure 4. Optimization Space with CUDA Parallelization

LANAI (LANguage for Autotuning Infrastructure) [19] to deal with these issues but further details are beyond the scope of this article.

A. *Optimizations specific to NVIDIA GPUs and CUDA Software Stack*

So far, we discussed the autotuning optimization space with the focus on multicore hardware that may be, in a generic form, characterized by multipurpose compute cores with a deep memory hierarchy of caches and complex main memory structures. Hardware accelerators such as compute-oriented GPUs differ from multicore hardware in a number of important ways including higher number of floating-point units, higher bandwidth to/from the main memory, and higher latency to the main memory. The memory hierarchy of GPUs is more shallow and often smaller caches are used on per-device basis. Despite these differences, the breakdown of execution time from Figure 2 is applicable to both types of compute platforms. Furthermore, this similarity applies across multiple GPU devices and compiler tool chains which might feature, for example, different approaches to instruction predication and branch removal algorithms. Such details depend on availability of Special Function Units (SFUs) on the target device and available hardware predication length. As a practical example, consider the difference between NVIDIA's Kepler and Maxwell architectures: the former features high end compute cards and has full support for 64-bit precision floating point arithmetic while the latter only targets gaming and rendering markets with 32-fold slowdown of 64-bit instructions.

The profiling on the NVIDIA hardware is done through either the `nvprof` command line tool or the `nvvp` GUI application. They are assisted by the hardware counters for minimal overhead on the running code. They were used to gather the profile and bottleneck information from the reference code.

Table I
FP16 AND ITS HARDWARE SUPPORT IEEE 754 (2008)

Precision	Width	Exponent	Mantissa	Epsilon	Max
Quadruple	128	15	112	$O(10^{-34})$	$O(10^{4932})$
Extended	80	15	64	$O(10^{-19})$	$O(10^{308})$
Double	64	11	52	$O(10^{-16})$	$O(10^{308})$
Single	32	8	23	$O(10^{-7})$	$O(10^{38})$
Half†	16	5	10	$O(10^{-3})$	65504

† defined only for storage

To maximize the bandwidth achieved by our implementation, we aim at efficient use of global and shared memory banks as well as enforce coalesced reads through stride-1 accesses. This is done explicitly since the GPU compilers often do not automatically handle Instruction-Level Parallelism (ILP). Also by design, CUDA shifts the Thread-Level Parallelism (TLP) to the user as the threads must be explicitly created and managed by the user code inside kernel functions. These considerations result in a modified search space for the GPU-optimized autotuning that is shown in Figure 4.

B. *Using Limited Precision Arithmetic*

1) Hardware Landscape for 16-bit Floating-Point: In 2017, a new type of hardware extension became much more main stream: a 16-bit floating-point precision arithmetic (FP16). This was precipitated by the initial experiments in deep network training with limited floating-point accuracy [20], [21]. Numerous hardware vendors and supercomputing sites are now involved with the trend. Consider the announced AMD GPUs MI5, MI8, MI25 whose model number corresponds to the peak performance of the card in FP16: 5 Tflop/s, 8 Tflop/s, and 25 Tflop/s, respectively. Softbank subsidiary ARM, announced publicly an extension to its NEON Vector Floating Point (VFP) to include FP16 in the V8.2-A architecture specification. NVIDIA GPUs that feature FP16 are widely available Tegra TX1 and Pascal P100 cards. In pre-release stage are Volta-based V100 and DG100 with the Xavier car platform down the line. TSUBAME 3 is one of the first supercomputers to prominently feature FP16 but other sites with NVIDIA Pascal hardware can fully utilize this new functionality. This is in line with our experiments in using FP16 in HPC benchmarking [22].

2) FP16 Considerations for Point Set Registration: FP16 precision is officially defined by the IEEE 754 standard. Its features are compared with the other floating-point precisions in Table I. Note that FP16 is not technically a compute format but only a storage format. This may be reflected by the choice of the runtime semantics of the announced Tensor Core unit in NVIDIA's Volta architecture that internally computes in FP32 arithmetic but consume and produce FP16 operands.

This is similar to the common practice of the way the Fused-Multiply-Add (FMA) instruction is implemented with higher precision for the intermediate results.

From the stand point of point set registration, FP16 has a potential benefit of increased bandwidth and compute intensity: 2-fold increase on the NVIDIA Pascal cards. The primary consideration is the limited range of the FP16 values as Table I indicates. At the algorithmic level, the use of limited precision in most calculations may serve as a opportunistic regularization scheme which, among other things, could prevent overfitting in noisy input data scenario.

IV. Formal Search Space Description Through Parameter Ranges and Constraints

The search space for autotuning optimization is multidimensional and heterogeneous[5]. It comprises different categories of parameters, namely:

- $\mathbb{B}_i$ binary type, for example: use either vector or scalar operations;
- $\mathbb{O}_j$ ordinal/categorical/enumeration type, for example: use shared memory, texture cache or Level 1 cache;
- $\mathbb{I}_k$ integer type, for example: a range of matrix blocking factors; and
- $\mathbb{C}_\ell$ continuous (represented as floating-point) type, for example: GPU occupancy above 30%.

Theoretically, the entire search space $\mathcal{S}$ may be represented by a Cartesian product of these dimensions:

$$\mathcal{S} = \prod_i \mathbb{B}_i \times \prod_j \mathbb{O}_j \times \prod_k \mathbb{I}_k \times \prod_\ell \mathbb{C}_\ell \quad (5)$$

In practice, the search space is much less regular than the above formula due to a number of software and hardware constraints. Definition of these complex search space may be done through a language designed for this purpose. In our case, we used the LANguage for Autotuning Infrastructure [19]. We used LANAI to specify a search space for a typical autotuning task with parameters and constraints derived from a matrix multiplication on a GPU [23]. The specification is written as a Python code. The space is a product of ranges of matrix sizes, block dimensions, threads-per-block counts, and so on. The conditions include maximum sizes per thread-block and the total number of threads etc. LANAI compiler generates a C code, that is used to prune the search space to limit the number of kernel candidates that have to be run on the GPU to assess their performance. The eligible kernel configurations are exported as CSV (Comma-Separated Values) file that is used by the benchtesting module, which is responsible for building and executing the kernels.

For completeness, we show in Figure 5 a quick overview of the syntax, semantics, and the generated code for a simple iterator and a constraint. Full details of the LANAI can be found elsewhere [19].

[5]In order to keep the discussion focused, we will mostly limit the examples to GPU accelerators and their parameters but the concepts are easily translatable to other HPC platforms such as x86 multicore CPUs, Intel Xeon Phi KNL, IBM POWER8 SMT.

	Semantics	LANAI syntax
Iterator	$\mathbb{I}_1 = \{1, 3, 5, 7\}$	I1 = range(1,8,2)
Generated C code:		`for (I1 = 1;I1 < 8;I1 += 1)`
Constraint	$\mathbb{E}_1 = \{i_1 \geq 3\}$	E1 = (I1 >= 3)
	Generated C code:	`if(! I1 >= 3) break;`

Figure 5. Sample syntax and semantics of LANAI code and the generated C code.

Equation (5) defines the search space, and as a consequence, dictates that the growth of the size of the space is combinatorial with the number product subsets. This is only true if the subsets are unconstrained. However, Figure 5 shows that constraints are part of the LANAI syntax and thus offer the user very effective means of limiting the size of the search space. Introducing these constraints compounds combinatorially due to Cartesian product. Thus the combinatorial growth of additional subsets is counteracted by the combinatorial diminishing due to new constraints. These makes the approach very efficient in practice and results in manageable number of combinations to consider and reasonable time to evaluate all the eligible cases. One important technical aspect of producing manageable search spaces is generating loop nests and loop termination statements in just the right order. This is done by organizing the iterators and constraints as a Direct Acyclic Graph (DAG) and generating the exploration code in a topological order which guarantees minimality of the explored space points [19].

V. Performance Results

In this section, we present the full set of results across a wide range of hardware platforms from the common x86 *multicore* server processors through the specialized *many-core* HPC machines with an enhanced feature set beyond the commodity hardware. We also include the compute-oriented GPU cards that come from a broad category of systems referred to as *hardware accelerators*.

Throughout this section, we present results for many variants of the equivalent implementation of the algorithm from Figure 1. Some of these variants correspond to the reference implementation mentioned in Section II. We indicate the performance of the reference code throughout the text to make it clear by how much our methodology outperforms it.

A. Description of the Tested Hardware Platforms

We performed our autotuning experiments on a number of hardware platforms and a quick summary of these is given in Table II. Out of the machines shown in the table, Intel Phi KNL is potentially the newest and might be the least familiar.

Table II
SUMMARY OF HARDWARE PLATFORMS USED IN THE PERFORMANCE TESTS.

Architecture	Manufacturer	Model	Name
x86	Intel	2620	Haswell
Phi	Intel	7280	Knights Landing
K40c	NVIDIA	GK110B	Kepler
P100	NVIDIA	GP100	Pascal

Table III
DETAIL CHARACTERISTICS OF THE INTEL XEON PHI KNIGHTS LANDING PLATFORM A MANYCORE CPU USED IN THE TESTS. THE PERFORMANCE NUMBERS AT THE BOTTOM OF THE TABLE ARE REPORTED VERBATIM FROM THE INTEL DOCUMENTATION AND PERFORMANCE MATERIALS.

Metric name	Metric value
Core count	68
Hardware thread count	272
Vector FPUs lengty	512 bits
Main memory RAM	DDR4
Max DDR4 RAM	384 GiB
DDR4 latency	≈140 ns
Fast RAM	16 GiB MCDRAM
Max MCDRAM	16 GiB
MCDRAM latency	≈170 ns
MCDRAM configuration modes	flat, cached, mixed
Level 3 cache	0
Peak FP32	6093 Gflop/s†
SGEMM	4065 Gflop/s
Peak FP64	3046 Gflop/s‡
DGEMM	2070 Gflop/s
LINPACK Benchmark	2000 Gflop/s
STREAM MCDRAM	490 GB/s
STREAM DDR4	90 GB/s

† 68 × 1.4 GHz × 2 VPUs × FMA × 16 AVX lanes
‡ 68 × 1.4 GHz × 2 VPUs × FMA × 8 AVX lanes

It presents an alternative to the superscalar x86 architecture[6] from Intel that is solely based on low-power Atom cores with up to 72 of such cores on the chip connected with a mesh interconnect and divided into 4 logical quadrants with NUMA-like consequences for data affinity. Further details on cache and memory hierarchy with on-chip characteristics is out of scope of this article. Initially planned in two hardware versions as self-boot servers and leveraged-boot cards, they are not aligned with the Intel HPC strategy in that form. Currently, only the self-boot units are commercially available and already installed at various computing center sites. The relevant details on the Intel Xeon Knights Landing (KNL) machine are given in Table III as they are available from public sources.

B. Timing Results for GNU and Intel Compilers on the x86 Machine

The first set of results come from the Intel x86 Haswell machine and are charted in Figures 6 and 7 for the GNU and Intel compiler, respectively. In order to limit the number of points in the chart, only the most representative configurations are shown. The configuration called orig-omp is the original code parallelized with OpenMP directives. We consider this code as a reference point that needs to be improved in order for us to claim advantage for our methodology. The orig-no-omp configuration is the reference point with the OpenMP parallelization disabled. Including this configuration in the charts shows how the GCC compiler struggles to parallelize the registration loop nest as almost all other parallel configuration run mostly at the same speed except for the opt-omp-noif-collapse configuration. That last configuration clearly outperforms all the others as it enables more parallelism and makes optimization and cross-thread work assignment for the GNU compiler. However, none of the optimizations tested allowed the GNU compiler to dip below 1 second for the largest point cloud size (40000 points). The Intel compiler has no problem to run under one second for a few optimization configurations. We did not

[6]Note that for the most part Xeon and Xeon Phi processors are binary compatible with the exception of the extra AVX512 instructions.

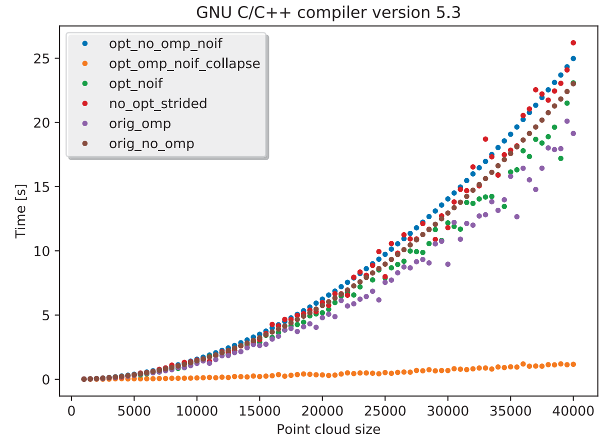

Figure 6. Point set registration timing results on the x86 Haswell machine with the GNU compiler for various OpenMP configurations.

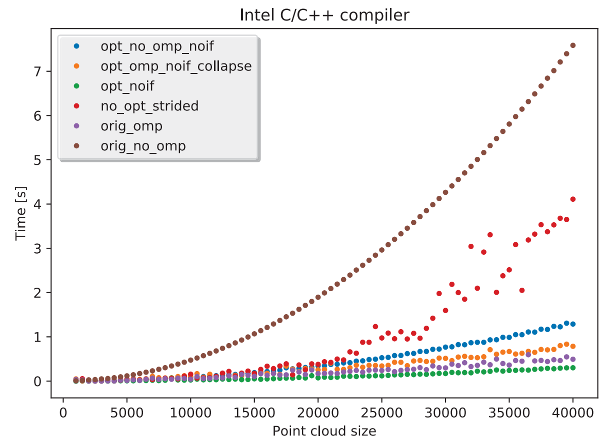

Figure 7. Point set registration timing results on the x86 Haswell machine with the Intel compiler for various OpenMP configurations.

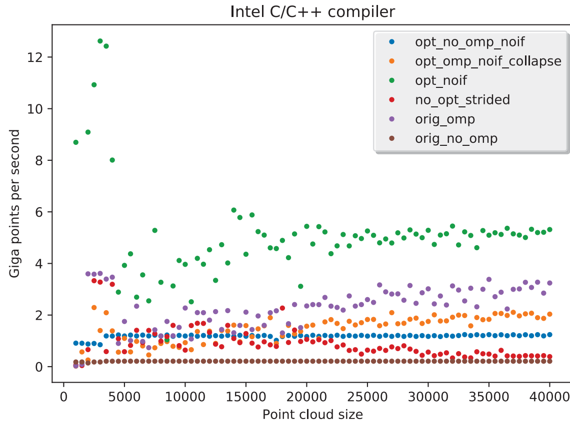

Figure 8. Point set registration performance rate results on the x86 Haswell machine with the Intel compiler for various OpenMP configurations.

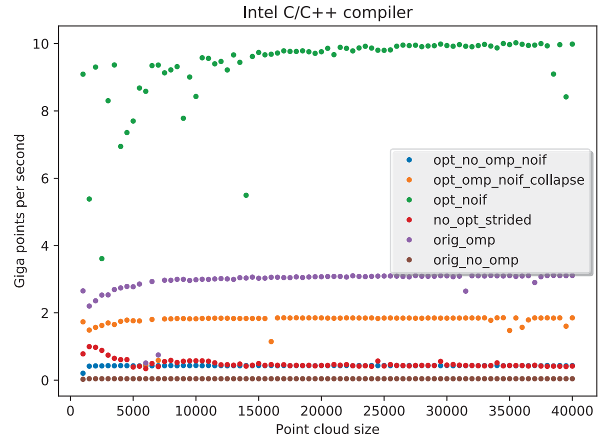

Figure 9. Point set registration performance rate results on the Xeon Phi Knights Landing machine with the Intel compiler for various OpenMP configurations.

conduct any further experiments to understand whether this may be related to either the generated instruction mix or more efficient OpenMP runtime. We leave this question for future work.

C. Application-Specific Performance Metric for Cross-Platform Comparisons

As we prepare for a more exhaustive comparisons across multiple hardware platform *and* input data sizes, using absolute timing for measuring performance becomes problematic. We recognize the importance of *time-to-solution* as an ultimate metric relevant to the end user but we would also like to be able to compare in a much more flexible fashion when there are many non-constant parameters of the tested hardware and the input data sets. At the same time, we strive to have the new metric be equivalent to the time-to-solution measurement with data size constant. The common performance metric in HPC codes is Gflop/s which has the one important advantage that it is one metric to use across all input data sets and even applications. Among many downsides is that Gflop/s rating assumes that there is uniform (preferably linear) relationship between Gflop/s and the essential application speed (commonly time-to-solution). This particular downside splinters in a multitude negative consequences such as artificial efforts to maximize the number of floating-point instructions that often come free if executed on cache-resident data but contribute very little to the application's ultimate goal. Instead, we derive an application-relevant metric based on the asymptotic performance theory [24].

In particular, for registration problems, the relevant performance metric is the number of points registered per second. Using this rate, we can quickly compare either scenes or surfaces with different number of points and still be able to map the rating back to time-to-solution. We use Giga-points-per-second (Gpts) unit for the execution rate that is defined as:

$$r = \frac{n_X n_Y}{t} 10^{-9} \text{ Gpts}$$

where n_X and n_Y represent the number of the input and output points, respectively. The scaling factor of 10^{-9} is a standard SI unit prefix that happens to render all the rates in this article to fall within the human-familiar range of small numbers that are greater than 1 and less than 50. Figure 8 shows this new metric applied to the timings from Figure 7. As an immediate advantage, we can see subtleties of the optimal configuration: the asymptotic performance rate is nearly 6 Gpts and the cache effects can be seen until 5000 points when the working set fits in cache and the performance rate is higher due to the higher bandwidth and lower latency of accessing data.

D. Performance Results on Many-core and GPU Device Cards

Figure 9 shows the performance rate results on the Xeon Phi KNL machine. Two clear differences may be observed:

1) Cache effects are mostly gone for small data sizes as compared with the x86 runs.
2) The inherent variability of timing measurements is gone and the graphs are much more smooth especially for the data sizes when the working set exceeds the cache size and the data has to be streamed from the main memory.

Figure 10 shows the first attempt of measuring performance on the K40c GPU but without coalescing the reads from the main memory. This is clearly against the common optimization technique used on GPUs and the effect of fixing this problem is shown in Figure 11 when coalescent reads were used. The performance increase is nearly 2-fold which confirms the importance of this optimization. In fact, this is what was chosen for the reference code with blocking factor

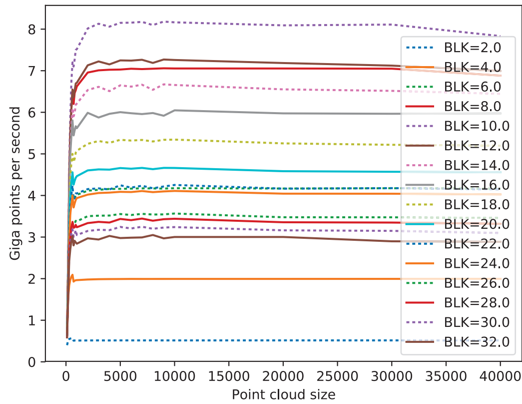

Figure 10. Point set registration performance rate results on the NVIDIA Kepler K40c card with the NVIDIA `nvcc` compiler for various loop blocking configurations with non-coalesced reads.

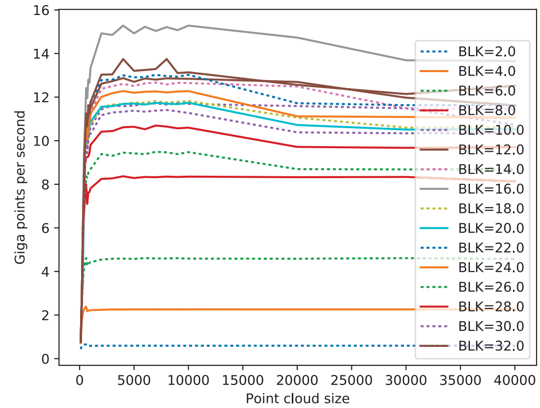

Figure 12. Point set registration performance rate results on the NVIDIA Kepler K40c card with the NVIDIA `nvcc` compiler for various loop blocking configurations without the use of shared memory.

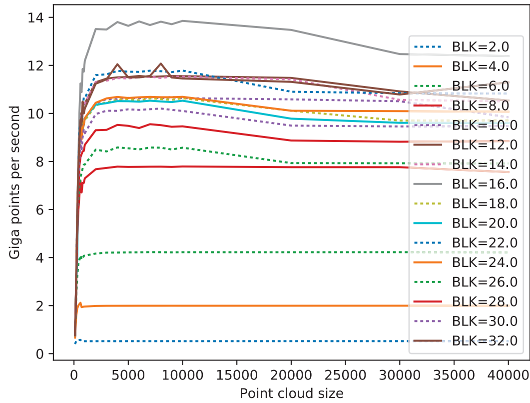

Figure 11. Point set registration performance rate results on the NVIDIA Kepler K40c card with the NVIDIA `nvcc` compiler for various loop blocking configurations with coalesced reads.

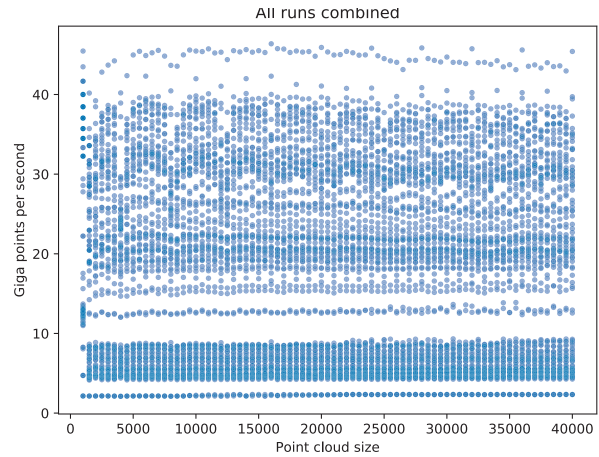

Figure 13. Point set registration performance rate results on the NVIDIA Pascal P100 card with the NVIDIA `nvcc` compiler for all loop blocking configurations.

BLK set to 32 to achieve 1-to-1 match with the number of threads in a thread warp. Our search indicates that it is possible to achieve even better registration rate with a smaller blocking factor.

A more counter-intuitive effect is to forgo the use of the shared memory which only occasionally leads to performance improvement. This happens to be the case for the registration algorithm we evaluated and Figure 12 shows that it is possible to gain a percentage point of performance if the shared memory is not used.

E. Limited Precision Implementation

We finally proceed to test the influence of low-precision hardware on the performance of the EM-ICP implementation. We use NVIDIA Pascal GPU card with the P100 chip that features dedicated FP16 units that perform the arithmetic instructions that are twice as fast as the corresponding FP32 instructions. The size of the FP16 data is twice as small and hence it is natural to expect two-fold increase in performance for both compute-bound and bandwidth-bound portions of the code.

First, Figure 13 shows all data points collected from running on the P100 card in FP32. This gives the reader an idea of the amount of autotuning experiments conducted automatically for the generated code versions from the eligible configurations. To clearly indicate the range of possible performance metrics, Figure 14 shows the best and the worst performance rate when FP16 arithmetic is used. It may seem underwhelming when compared with all the prior results presented so far. To identify how FP16 and FP32 arithmetic results compare against each other, we show in Figure 15 just the relevant graphs. As the figure indicates, the FP16 arithmetic performs faster in absolute terms but the

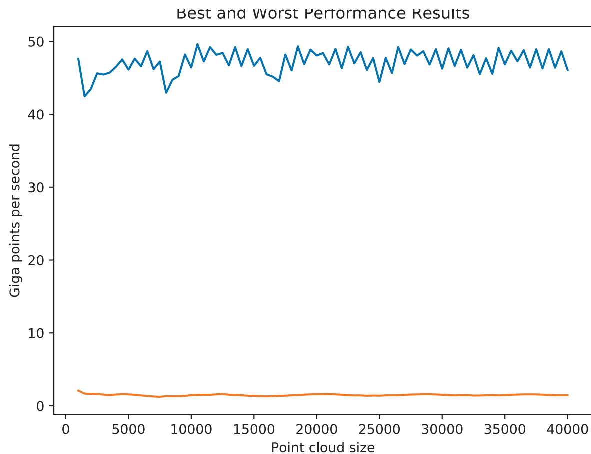

Figure 14. Point set registration performance rate results on the NVIDIA Pascal P100 card with the NVIDIA `nvcc` compiler for the best and worst autotuning configurations using FP16 arithmetic.

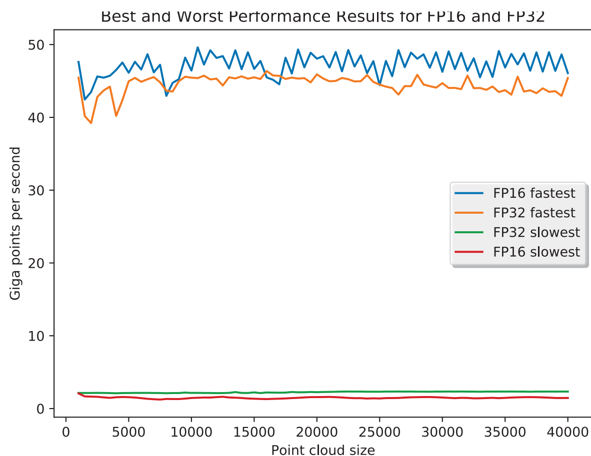

Figure 15. Point set registration performance rate results on the NVIDIA Pascal P100 card with the NVIDIA `nvcc` compiler for the best and worst autotuning configurations using either FP16 or FP32 arithmetic.

improvement is not as high as the 2-fold higher performance rate suggested by the raw hardware specification. Next, we examine this result in more detail.

```
#pragma omp parallel for
for (i ← 0; i < n_X; i ← i+1)
  for (j ← 0; j < n_Y; j ← j+1)
    ...
#pragma omp parallel for collapse(2)
for (i ← 0; i < n_X; i ← i+1)
  for (j ← 0; j < n_Y; j ← j+1)
    ...
```

Figure 16. Optimal Configuration for GNU Compiler on x86

```
// FP16-FP32 conversions contribute additional
// overheads and hit against hardware limit
cvt.f32.f16        f, r   // convert from FP16
sqrt.approx.f32    f, f   // compute in FP32
cvt.rn.f16.f32     r, f   // convert to FP16
```

Figure 17. A Glance at PTX (NVIDIA's Pseudo-Assembly) around the computation of the approximate square root for MSE (Mean Square Error).

VI. Discussion

As Figure 6 indicated, the GNU `gcc` compiler does not seem enable optimized parallelization levels and, as a consequence, efficient use of many threads remains lacking in resulting implementation speed. This may be remedied by the user when a lot of parallelism is exposed through introducing a large index sets manually with the pragma directives. In OpenMP specifically, this may be achieved with the `collapse` clause that allows to merge loop nests for a combined index set that is subsequently divided among the available threads. Such an optimization will result from the transformation shown in Figure 16. Our autotuning framework allows to introduce the clause in the generated variants and then test a number of loop-collapsing parameters to find out the optimal setting for the tested hardware platform.

Another surprising result uncovered during the autotuning procedure is presented in Figure 15 that shows a comparatively small difference between the performance obtained from the variants of code that use either FP16 and FP32 arithmetic. In our analysis, this may be attributed to the conversion overhead. We managed to confirm this information by looking at the PTX code generated by the NVIDIA `nvcc` compiler. The resulting PTX code is shown in Figure 17 in a somewhat stylized form when all the spurious details were removed and the essential information on data types highlighted of emphasis. The hardware conversion units on the NVIDIA Pascal chips are limited in number, unlike, for example, floating point units. This slows down the execution within a single thread warp and contributes to the overall slowdown of the code. This clearly obviates any performance gains that might have been obtain from using FP16 to begin with. This is the case when autotuning process exposes bottlenecks that might have not been otherwise present in non-optimized or manually optimized code due to a limited exposure to the variety implementation variants. In addition to the limited count of square root units, there is a limit on conversion throughput and the thread-synchronizing effect of using Special Function Units (SFUs) that decreases the number of threads eligible for execution.

VII. Conclusions and Future Work

In this article, we presented an application of the autotuning approach to the EM-ICP algorithm that is a stochastic method used for point set registration. We applied various transformations and used an automated procedure to generating a

set of implementation variants. This allowed us to, first, exceed the performance achieved by the reference code that was only optimized manually. Then we explored the entire space of potential performance-oriented implementations to engineer stable and portable performance levels that makes the EM-ICP code available across a large range of hardware platforms. Our methodology generated implementations for multicore, many-core, and accelerator-equipped machines.

Extending this work to a wider range of registration codes is a natural future direction that we intend to pursue. However, due to a large number of algorithms and algorithmic variants for the registration problem, we will be guided in our selection by the need of the scientific fields that are in need of point set registration implementations for analysis of large data sets. These computational science fields benefit the most from the performance engineering efforts that we presented because of high rates and large volumes of data coming from their experimental instruments such as Atomic Probe Tomography microscopes in material science.

Acknowledgements

This research was supported by NSF through grant CCF-1527706 and ACI-1642441.

References

[1] D. Larson, T. Prosa, R. Ulfig, B. Geiser, and T. Kelly, *Local Electrode Atom Probe Tomography: A User's Guide*. Springer, 2013.

[2] M.K.Miller and R. Forbes, *Atom-Probe Tomography: The Local Electrode Atom Probe*. Springer, 2014.

[3] Y.-M. Kim, A. Morozovska, E. Eliseev, M. Oxley, R. Mishra, S. Selbach, T. Grande, S. Pantelides, S. Kalinin, and A. Borisevich, "Direct observation of ferroelectric field effect and vacancy-controlled screening at the bifeo3/laxsr1-xmno3 interface," *Nat Mater*, vol. 13, no. 11, pp. 1019–1025, 2014.

[4] K. Sohlberg, S. Rashkeev, A. Borisevich, S. Pennycook, and S. Pantelides, "Origin of anomalous pt-pt distances in the pt/alumina catalytic system," *ChemPhysChem*, vol. 5, no. 12, pp. 1893–1897, 2004.

[5] A. Bartok, R. Kondor, and G. Csanyi, "On representing chemical environments," *Physical Review B*, vol. 87, no. 18, 2013.

[6] H. Li and R. Hartley, "The 3D-3D registration problem revisited," in *ICCV 2007: Eleventh IEEE International Conference on Computer Vision*, 2007.

[7] P. J. Besl and N. D. McKay, "A method for registration of 3-D shapes," *IEEE PAMI*, vol. 14, no. 2, pp. 239–256, February 1992.

[8] Z. Zhang, "Iterative point matching for registration of free-form curves and surfaces," *IJCV*, vol. 13, no. 2, pp. 119–152, October 1994.

[9] A. W. Fitzgibbon, "Robust registration of 2D and 3D point sets," *Image and Vision Computing*, vol. 21, pp. 1145–1153, 2003.

[10] S. Rusinkiewicz and M. Levoy, "Efficient variants of the ICP algorithm," in *International Conference on 3D Digital Imaging and Modeling (3DIM)*, 2001, pp. 145–152.

[11] S. Granger and X. Pennec, "Multi-scale EM-ICP: A fast and robust approach for surface registration," in *ECCV 2002*, ser. LNCS 2353, A. H. et al., Ed. (c) Springer-Verlag Berlin Heidelberg, 2002, pp. 418–432.

[12] H. Chui and A. Rangarajan, "A feature registration framework using mixture models," in *IEEE Workshop on MMBIA*, June 2000, pp. 190–197.

[13] S. Gold, C. P. Lu, A. Rangarajan, S. Pappu, and E. Mjolsness, "New algorithms for 2D and 3D point matching: Pose estimation and corresp." in *NIPS*, vol. 7. The MIT Press, 1995, pp. 957–964.

[14] A. Rangarajan, H. Chui, E. Mjolsness, L. Davachi, P. S. Goldman-Rakic, and J. S. Duncan, "A robust point matching algorithm for autoradiograph alignment," *MIA*, vol. 1, no. 4, pp. 379–398, 1997.

[15] H. Chui and A. Rangarajan, "A new algorithm for non-rigid point matching," in *CVPR*, vol. 2. IEEE Press, June 2000, pp. 44–51.

[16] ——, "A new point matching algorithm for non-rigid registration," *CVIU*, vol. 89, no. 2-3, pp. 114–141, February 2003.

[17] A. Myronenko, X. Song, and M. A. Carreira-Perpiñán, "Non-rigid point set registration: Coherent Point Drift," in *NIPS*, 2007, pp. 1009–1016.

[18] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[19] P. Luszczek, M. Gates, J. Kurzak, A. Danalis, and J. Dongarra, "Search space generation and pruning system for autotuners," in *Proceedings of IPDPSW*, ser. The Eleventh International Workshop on Automatic Performance Tuning (iWAPT) 2016. Chicago, IL, USA: IEEE, May 23rd 2016.

[20] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015. [Online]. Available: http://arxiv.org/abs/1502.02551

[21] ——, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 1737–1746. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045303

[22] P. Luszczek, J. Kurzak, I. Yamazaki, and J. Dongarra, "Towards numerical benchmark for half-precision floating point arithmetic," in *2017 IEEE High Performance Extreme Computing Conference*, 2017.

[23] H. Anzt, B. Haugen, J. Kurzak, P. Luszczek, and J. Dongarra, "Experiences in autotuning matrix multiplication for energy minimization on gpus," *Concurrency and Computation, Practice and Experience*, vol. 27, no. 17, pp. 5096–5113, December 10 2015.

[24] R. W. Hockney and I. J. Curington, "$f_{\frac{1}{2}}$: A parameter to characterize memory and communication bottlenecks," *Parallel Computing*, vol. 10, no. 3, pp. 277–286, May 1989.