

# Accelerated point set registration method



Journal of Defense Modeling and  
Simulation: Applications,  
Methodology, Technology  
1–19

© The Author(s) 2023

DOI: 10.1177/15485129221150454

journals.sagepub.com/home/dms



Ryan M Raettig<sup>1</sup> , James D Anderson<sup>2</sup>, Scott L Nykl<sup>1</sup> , and Laurence D Merkle<sup>1</sup>

## Abstract

In computer vision and robotics, point set registration is a fundamental issue used to estimate the relative position and orientation (pose) of an object in an environment. In a rapidly changing scene, this method must be executed frequently and in a timely manner, or the pose estimation becomes outdated. The point registration method is a computational bottleneck of a vision-processing pipeline. For this reason, this paper focuses on speeding up a widely used point registration method, the iterative closest point (ICP) algorithm. In addition, the ICP algorithm is transformed into a massively parallel algorithm and mapped onto a vector processor to realize a speedup of approximately an order of magnitude. Finally, we provide algorithmic and run-time analysis.

## Keywords

Point set registration, ICP, parallel computing, GPU, CUDA, optimization

## 1. Introduction

Biological systems provide the ability for humans to rapidly process imagery, detect objects, and determine their pose relative to the scene. These biological systems contribute to a human's perception. This process quickly happens, enabling humans to make responsive decisions in a rapidly changing environment. In contrast, with respect to robotics, self-driving vehicles, and autonomous systems, these biological systems must be replaced by hardware and software using computer vision. This process utilizes image and vision-processing algorithms to enable these inorganic systems to sense their surroundings in dynamic environments.

Vision systems capture images of their environment and generate sensed point clouds through feature detection and stereo block matching<sup>1</sup> between multiple images. These sensed points must then be processed to create a semantic interpretation of an environment. Point registration<sup>2</sup> is the first step to create a semantic interpretation. It enables the autonomous system to align these sensed points with a known truth model. In this manner, the autonomous system can classify the sensed object as well as estimate its relative pose.

The human retina has a higher concentration of cells at the center of the retina called the fovea.<sup>3</sup> Thus, humans tend to focus on the center of their field of view, as more detail provides more information for perception.

With computer vision systems, higher resolution images provide a more accurate orientation of objects but consequently an exponential computational growth based on pixel density. Thus, processing times increase when the point set registration method saturates or uses all available hardware processors. This issue inhibits the autonomous system from rapidly processing imagery and quickly making decisions or actions on it. For this reason, either reducing the number of sensed points or accelerating the point registration method is required to achieve real-time object registration. However, reducing the number of sensed points reduces the information gained from the imagery, leading to a less accurate perception of objects' classification, position, and orientation. This concept illuminates the value and the problem solved by a parallel point set registration method. This paper focuses on accelerating a point set registration method.

<sup>1</sup>Department of Electrical and Computer Engineering, Air Force Institute of Technology, USA

<sup>2</sup>Department of Computer Science and Engineering, Wright State University, USA

### Corresponding author:

Ryan M Raettig, Department of Electrical and Computer Engineering, Air Force Institute of Technology, 2950 Hobson Way, Wright-Patterson Air Force Base (WPAFB), OH 45433-7765, USA.

Email: ryan.raettig@spaceforce.mil

One military application this research is pivotal in is the United States Air Force (USAF) automated aerial refueling (AAR) effort.<sup>4</sup> AAR intends to supplement USAF refueling missions, such as with the KC-46 aircraft, by achieving the capability to control the boom and safely refuel, independent of input from the boom operator. Using the tanker's stereo-vision cameras and a computer vision processing pipeline, the receiving aircraft's relative pose can be estimated. From this information, the tanker's boom can be safely moved into the receiver's refueling receptacle. ICP has shown the required accuracy to support USAF refueling operations.

This paper makes the following contributions:

1. The point set registration method of the iterative closest point (ICP)<sup>5</sup> algorithm, using Previous Nearest Neighbor Optimized Delaunay Walk (PNNOPT) nearest neighbor,<sup>6</sup> is transformed from a serial algorithm to a massively parallel algorithm that executes efficiently on a vector processor such as a graphics processing unit (GPU).
2. Using this, we empirically quantify over a  $25 \times$  speedup on point clouds of realistic sizes.
3. The parallel algorithm is shown implemented in Compute Unified Device Architecture (CUDA) on an NVIDIA RTX 3080 (10 GB) GPU.
4. Finally, the parallel algorithm is analyzed and theoretical and real run times are compared.

The paper's subsequent parts are arranged as follows. Related work presents a literature review of related works to accelerating the ICP algorithm. Methodology presents the definition of the accelerated ICP algorithm, target parallel platform, decomposition of the algorithm, optimization, analysis of the algorithm, and experiments. Results presents the performance and accuracy of our algorithm. Finally, the paper finishes with a conclusion and future work of this research.

## 2. Related work

### 2.1. ICP algorithm

The ICP algorithm registers a sensed three-dimensional (3D) point cloud onto a reference truth 3D point cloud. It can be used to align a plethora of shapes approximated as point sets. Besl and McKay<sup>5</sup> discusses the parameters to which the ICP algorithm can be applied: "(1) sets of points, (2) sets of line segments, (3) sets of parametric curves, (4) sets of implicit curves, (5) sets of triangles, (6) sets of parametric surfaces, and (7) sets of implicit surfaces." Besl refers to  $P$  as the point set the ICP algorithm is attempting to align. In real environments,  $P$  inherently is generated from a sensor, for example, a high-definition

- a. Compute the closest points:  $Y_k = C(P_k, X)$  (cost:  $0(N_p N_x)$  worst case,  $0(N_p \log N_x)$  average).
- b. Compute the registration:  $(\vec{q}_k, d_k) = Q(P_0, Y_k)$  (cost:  $O(N_p)$ ).
- c. Apply the registration:  $P_{k+1} = \vec{q}_k(P_0)$  (cost:  $O(N_p)$ ).
- d. Terminate the iteration when the change in mean-square error falls below a preset threshold  $\tau > 0$  specifying the desired precision of the registration:  $d_k - d_{k+1} < \tau$ .

**Figure 1.** Steps in each ICP iteration directly from Besl's paper *A Method for Registration of 3-D Shapes*.<sup>5</sup>

camera. To align  $P$ , ICP must have prior knowledge of the known truth model. Besl refers to this truth point set as  $X$ . In this manner, the ICP algorithm aligns  $P$  onto  $X$ .

The following list explains the ICP algorithm from the study by Besl and McKay.<sup>5</sup>

1. The first step is to compute nearest neighbor correspondences between  $P$  and  $X$  and assign the correspondences to  $Y$ .
2. The second step is to compute the registration or the rotation and translation that transforms  $P$  onto  $X$ .
3. The last step is to apply the registration to  $P$  and calculate the error between  $P$  and  $X$ .
4. If the error is lower than a set threshold, the algorithm exits returning the rotation and translation. Otherwise, the algorithm iterates again to increase the accuracy of the rotation and translation to minimize this error.

Figure 1 shows these steps from the study by Besl and McKay.<sup>5</sup>

The accelerated ICP algorithm described in this paper is based on the "classic" point-to-point ICP algorithm by Besl. The point-to-point approach has not only shown increased accuracy but also increased processing time because of the nearest neighbor search task.<sup>7</sup> Point-to-projection<sup>8</sup> is an approach that eliminates this task but gives a less accurate pose estimation.<sup>7</sup> Point-to-(tangent)-plane<sup>9</sup> is the most accurate approach but most computationally expensive.<sup>7</sup> In this approach, both the nearest neighbor correspondences and the intersection surface are required to be computed.<sup>7</sup> To execute nearest neighbor<sup>10</sup> correspondences, the following approaches can be applied: brute force,<sup>11</sup> k-d tree,<sup>12-14</sup> Delaunay traversal,<sup>6,15</sup> or a number of alternative variations. Presented in this paper, the parallel mapping and optimization strategies can be deployed regardless of the preferred ICP variation or nearest neighbor approach.

## 2.2. Accelerated point set registration

**2.2.1. Parallel ICP.** Langis et al.<sup>16</sup> implemented a parallel ICP algorithm in a parent-child model.<sup>17</sup> The parent process spawns children processes to compute nearest neighbor correspondences. The children report these correspondences to the parent. The parent then calculates the rotation and translation based on these correspondences and applies the rotation and translation to  $P$ . The following is Langis explaining the parallel iterations: “Each child concurrently computes the correspondences between points in  $I_f$ , and the points of  $I_r$ . The resulting correspondences are then sent back to the parent process.”<sup>16</sup>  $I_f$  and  $I_r$  represent the sensed and truth model in this example. The parent process becomes the bottleneck in this model with potentially many children attempting to report concurrently.

**2.2.2. GPU point-cloud registration.** Rahman et al.<sup>18</sup> implemented a fast GPU point-cloud registration algorithm mapped into four blocks on the GPU. In this example, a block is a separate process mapped on the GPU. The first block finds the centers of mass of the point clouds. The second block transforms the point clouds to the origin. The third block executes the singular value decomposition (SVD)<sup>19</sup> technique. The fourth block finds the rotation and translation.

In contrast to Rahman’s algorithm based on SVD to align two point clouds, the implementation presented in this paper utilizes Besl’s ICP algorithm.

**2.2.3. Expectation-maximization ICP.** Tamaki et al.<sup>20</sup> implemented an accelerated ICP algorithm implemented in CUDA. The algorithm was based on soft assign nearest neighbor correspondence and called Expectation-Maximization (EM)-ICP.<sup>21</sup> Through this, an estimation of nearest neighbor correspondence was assigned. Based on these estimated correspondences, an estimated rotation and translation were calculated. In contrast, the accelerated ICP algorithm in this paper computes exact nearest neighbor correspondence and thus returns a rotation and translation transformation with increased accuracy.

## 2.3 CUDA and GPUs

GPUs were originally made for computer graphics processing to concurrently execute vertex and pixel pipelines. Thus, GPUs have many cores for processing. For this reason, developers aimed to utilize these cores for not just graphics, but also general-purpose computing. NVIDIA introduced CUDA for developers to do just that through one application programming interface (API).

NVIDIA released the first version of CUDA, 1.0, on 23 June 2007.<sup>22</sup> CUDA presents a Single-Instruction, Multiple-

Thread (SIMT)<sup>17</sup> model to the GPU. CUDA is specifically designed for NVIDIA GPU architectures.<sup>22</sup> As of writing this paper, the current version of CUDA is 11.0. As NVIDIA GPU architectures advanced, various features have been added to CUDA. The compute capability of an NVIDIA architecture correlates to the supported features of CUDA on said architecture.<sup>23</sup>

Oden<sup>24</sup> goes over an interesting discussion of Python packages that use C-CUDA and Numba-CUDA for GPU implementations. In every practical sense, the CUDA presented in this paper is C-CUDA. Oden explains that using Python for GPU implementations is not a preferable avenue for performance because the user is limited to pre-compiled libraries. However, Oden does go over metrics that show C-CUDA libraries are faster than Numba-CUDA libraries. Oden shows C-CUDA Python packages reaching 85% performance and Numba-CUDA Python packages reaching 50% performance.<sup>24</sup> With this in mind, in an accelerated ICP implementation, using the low-level C-CUDA programming language directly yields the best performance increases because the most control is given to the user. For these reasons, CUDA will be used in this research to implement an accelerated ICP algorithm.

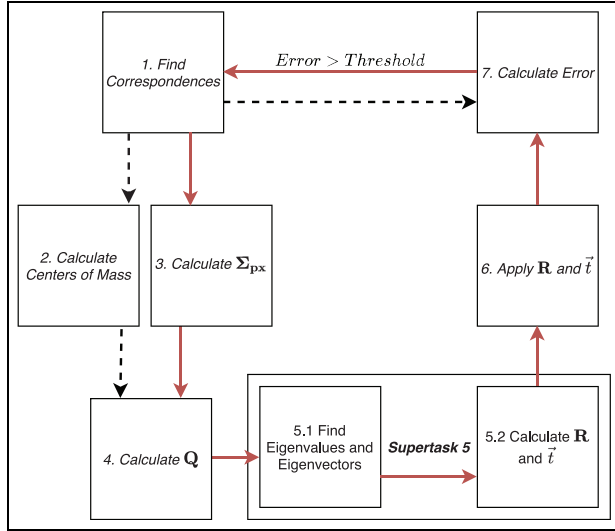
## 3. Methodology

### 3.1. Definition of algorithm

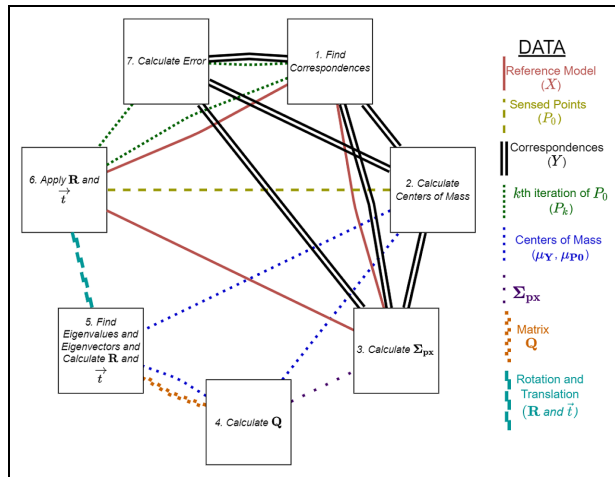
The goal of this research is to accelerate the ICP algorithm through parallelism to quickly align sensed points to a known reference model via a rotation and translation that minimizes the error between these sets. Thus, the steps in Figure 1 are parallelized by the accelerated ICP algorithm. Figure 2 shows the task dependency graph.<sup>17</sup> Figure 3 shows the task interaction graph.<sup>17</sup> This graph shows privatization will be an effective optimization strategy by replicating data and reducing data contention among tasks.

The accelerated ICP algorithm must execute sequentially through adjacent tasks in the solid red loop in Figure 2. This solid red loop represents the critical path of the accelerated ICP algorithm. In this way, the critical path is the combination of the critical paths of each ordered task. Supertasks 2 and 3, that is, Calculate Centers of Mass and Calculate  $\Sigma_{px}$ , respectively, may be executed in parallel, but supertask 3, that is, Calculate  $\Sigma_{px}$  is in the critical path. Excluding tasks with reductions, the critical path within each task is similar. In tasks with reductions, the critical path is  $\log_2(n)$ , which is the number of steps or levels in the reduction.

For the remainder of the paper, the following symbols and rules are used:  $X$  is the reference or truth model,  $P_0$  represents the original sensed points,  $P_k$  is the  $k$ th iteration of the registration applied to  $P_0$ ,  $Y$  is the list of nearest neighbors or correspondences of  $P_k$  and  $X$ . The sets:  $X$ ,  $P_0$ ,  $P_k$ , and  $Y$  contain  $3 \times 1$  column vectors of 3D points.



**Figure 2.** Task dependency graph of accelerated ICP steps from Figure 1 with critical path in solid red.



**Figure 3.** Task interaction graph of accelerated ICP supertasks from Figure 1 with supertask 5 consolidated.

$\Sigma_{px}$  is a cross-covariance matrix generated between  $P_0$  and  $Y$ .  $Q$  is a  $4 \times 4$  symmetric matrix from which the unit eigenvector corresponding to the maximum eigenvalue represents the axis of rotation. Matrices are column-major and indexed by a (row, column) subscript in the algorithms. Finally, the optimal rotation and translation are represented by  $R$  and  $\vec{t}$ , respectively.

In the sections and ideas that follow, the accelerated ICP algorithm has been implemented via CUDA. A CUDA core is the processing unit and a CUDA thread is the processing context. Throughout the paper, we will be referencing and describing the mapping of CUDA threads inside of tasks.

### 3.2. Target parallel platform

As stated previously in the Introduction section, the target parallel platform in this research is an NVIDIA RTX 3080 GPU. This is compared against the target platform for the serial implementation of ICP, which is a machine with an Intel Xeon central processing unit (CPU) with 3.10 GHz clock speed and 128 GB of RAM.

The NVIDIA RTX 3080 GPU has 8704 CUDA cores and delivers up to 29.8 teraflops.<sup>25</sup> The RTX 3080 has a core clock speed of 1440 MHz and a memory clock speed of 1188 MHz. The memory size is 10 GB and memory bandwidth is 760.3 GB/s.<sup>26</sup> The RTX 3080 has an NVIDIA Ampere architecture with a compute capability of 8.0.<sup>27</sup>

The control structure used through CUDA is the Single Program Multiple Threads (SPMT).<sup>23</sup> CUDA version 11.0 is used in this accelerated ICP algorithm implementation.<sup>23</sup> In this manner, one program is compiled and run on the GPU, but each CUDA core has a specific control sequence mapped to it by the program. CUDA cores in the same block can communicate via shared memory. CUDA cores in different blocks communicate via global memory.

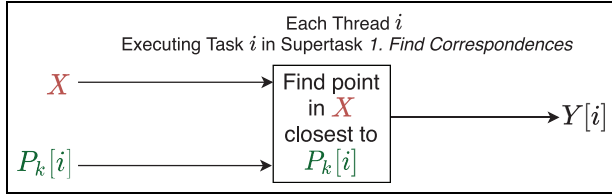
The accelerated ICP algorithm employs the data-parallel model.<sup>17</sup> The tasks are decomposed and mapped in such a way to take advantage of data-parallelism at the thread level inside CUDA kernels on the GPU. Each task takes advantage of block distributions of the data into the thread blocks, increasing the locality of interaction within the thread block and splitting the computation between threads. The CPU–GPU CUDA setup is a representation of a manager–worker setup. Here, the CPU is the manager and the GPU is the worker. In this setup, multiple GPUs could potentially be added as workers. However, in this research, only a single GPU is leveraged.

### 3.3. Algorithm decomposition and mapping

The data decomposition technique<sup>17</sup> is used throughout the accelerated ICP algorithm. The nature of CUDA enables for a very fine-grain task decomposition. For this reason, the granularity of the decomposition leverages any possible parallelism in the algorithm among threads.

**3.3.1. Decomposition.** The following list contains the supertasks from the decomposition of the accelerated ICP algorithm.

1. Find correspondences
2. Calculate centers of mass
3. Calculate  $\Sigma_{px}$
4. Calculate  $Q$
5. (5.1) Find eigenvalues and eigenvectors and (5.2) Calculate  $R$  and  $\vec{t}$



**Figure 4.** Task interaction graph of supertask 1—Find correspondences—outlined in Algorithm 1.

**Algorithm 1.** Supertask 1. Find Correspondences pseudo code.

```

1: function NEARESTNEIGHBOR ( $P_k[i]$ ,  $X$ ,  $Y[i]$ )
2:    $d_{min} = \max(\text{float})$ 
3:    $j = -1$ 
4:    $counter = 0$ 
5:   for each  $x \in X$  do
6:     if  $\text{squaredDistance}(x, P_k[i]) < d_{min}$  then
7:        $j = counter$ 
8:        $d_{min} = \text{squaredDistance}(x, P_k[i])$ 
9:     end if
10:     $counter = counter + 1$ 
11:   end for
12:    $Y[i] = X[j]$ 
13: end function

```

**Algorithm 2.** Supertask 2. Calculate centers of mass of  $P_0$  and  $Y$  pseudo code.

```

1: function CENTEROFMASS( $C$ ,  $\mu$ )
2:    $\mu = \frac{\sum C}{|C|}$ 
3: end function
4:  $\text{CenterOfMass}(P_0, \mu_{P_0})$ 
5:  $\text{CenterOfMass}(Y, \mu_Y)$ 

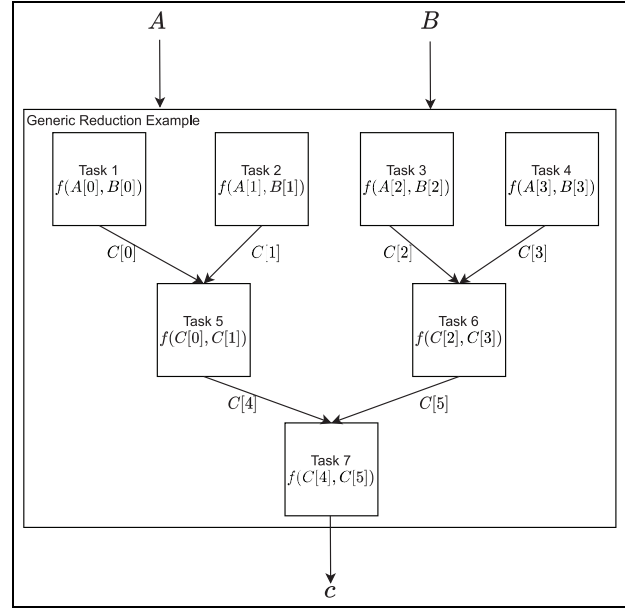
```

6. Apply  $\mathbf{R}$  and  $\vec{t}$
7. Calculate error

Supertask 1. Find correspondences.

This is the first supertask in the algorithm and it computes the nearest neighbor matches between the sensed points and the reference model. Algorithm 1 shows the pseudo-code for each thread. Squared Euclidean distance is used to eliminate the expensive computational requirement of a square root. Each thread  $i$  inputs  $X$  and  $P_k$  and assigns  $Y[i]$  to the point in  $X$  closest to  $P_k[i]$ . In this manner, a thread is launched for each member in  $P_k$ .

In real environments,  $P_k$  can vary from a couple of 100 points to 1000s of points. The number of points depends on sensor configuration and the accuracy required. The

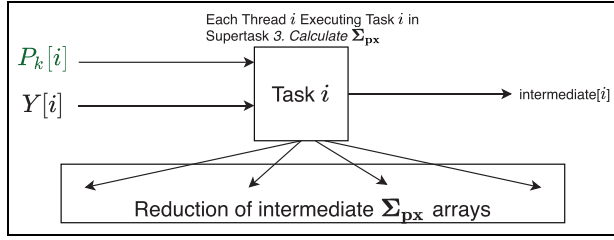


**Figure 5.** This shows the task interaction graph of a generic reduction of two arrays with four elements. Specific to the supertask, the function is common among all inner tasks. This example can be expanded to imply the task interaction graph for supertask 2—Calculate centers of mass, and the reduction portion of supertask 7—Calculate error. This is outlined in Algorithms 2 and 7.

accelerated ICP implementation has been tested on point cloud sizes from 4k to 63k and typically executes less than 30 iterations to achieve an error under  $1 \times 10^{-6}$ . Figure 4 shows the task interaction graph for tasks inside supertask 1—Find Correspondences. Because this is a one-to-one mapping, this supertask has a degree of concurrency of the number of sensed points.

Supertask 2. Calculate centers of mass.

This step maps to the beginning of Besl’s “Compute the registration.”<sup>5</sup> This step computes the estimated rotation ( $\mathbf{R}$ ) and translation ( $\vec{t}$ ) between the sensed points and the reference model. The centers of mass of  $Y$  and  $P_0$  are calculated in this supertask. Algorithm 2 shows the pseudo code for this supertask. In this example,  $\mu_Y$  is the center of mass of  $Y$  and  $\mu_{P_0}$  is the center of mass of  $P_0$ . This supertask consists of two all-to-one reductions. Figure 5 shows the task interaction graph of a single reduction for tasks inside supertask 2—Calculate centers of mass. Because this is a binary tree reduction, this supertask has a maximum degree of concurrency of half the number of sensed points. However, this supertask has an average degree of concurrency of the number of threads active throughout the binary tree reduction divided by the number of levels



**Figure 6.** Task interaction graph of each task in supertask 3—Calculate  $\Sigma_{px}$ —a 1 to 1 map and then a reduction for each of the nine values in the  $\Sigma_{px}$  matrix used in Algorithm 3.

---

**Algorithm 3.** Supertask: 3. Calculate  $\Sigma_{px}$  pseudo code.

---

```

1: function DOT_Σpx(P0, Y, Σpx)
2:   F = I3
3:   for each x,y ∈ P0,Y do
4:     B = x · y
5:     F = F + B
6:   end for
7:   Σpx = F / |P0|
8: end function

```

---

in the tree. Let  $n$  be the number of sensed points. The average degree of concurrency is  $\frac{2n-1}{\log_2(n)+1}$ .

Supertask 3. Calculate  $\Sigma_{px}$ .

In this supertask,  $\Sigma_{px}$ , a  $3 \times 3$  matrix, is generated from the point correspondences. Algorithm 3 shows the pseudo code to calculate  $\Sigma_{px}$ . In this algorithm,  $I_3$  represents a  $3 \times 3$  identity matrix. Figure 6 demonstrates the first step of this supertask is a one-to-one map in which each thread computes a dot product. The degree of concurrency of the map is the number of sensed points. In the final portion of this supertask, summing the dot products is an all-to-one binary tree reduction. As stated previously, if  $n$  is the number of sensed points, the average degree of concurrency for this supertask is  $\frac{2n-1}{\log_2(n)+1}$ .

Supertask 4. Calculate  $Q$ .

This supertask generates  $Q$ , a  $4 \times 4$  matrix, from which  $R$  and  $\vec{t}$  will be calculated. Algorithm 4 shows the pseudo-code for supertask 4—Calculate  $Q$ . The centers of mass and  $\Sigma_{px}$  were calculated in the previous two supertasks. Figure 10 shows the task interaction graph for the tasks

---

**Algorithm 4.** Supertask 4. Calculate  $Q$  pseudo code.

---

```

1: function CALC_Q(Σpx, μP0, μY, Q)
2:   M = μP0 · μY
3:   D = Σpx - M
4:   E = DT
5:   A = D - E
6:   Δ = [a2,3 a3,1 a1,2]
7:   traceD = tr(D)
8:   S = D + E
9:   s1,1 = s1,1 - traceD
10:  s2,2 = s2,2 - traceD
11:  s3,3 = s3,3 - traceD
12:  Q = [
    [traceD Δ1 Δ2 Δ3]
    [Δ1 s1,1 s1,2 s1,3]
    [Δ2 s2,1 s2,2 s2,3]
    [Δ3 s3,1 s3,2 s3,3]
  ]
13: end function

```

---

within supertask 4—Calculate  $Q$ . The maximum degree of concurrency for this supertask is 16.

Supertask 5. Find eigenvalues and eigenvectors and calculate  $R$  and  $\vec{t}$ .

In this supertask,  $R$  and  $\vec{t}$  are calculated based on the eigenvalues and eigenvectors of  $Q$  and both point clouds' centers of mass.  $Q$  was generated in the previous supertask, and the centers of mass were calculated in supertask 2—Calculate centers of mass. After finding the eigenvalues and eigenvectors of  $Q$  in supertask 5.1, that is, Find eigenvalues and eigenvectors, the eigenvector corresponding to the maximum eigenvalue is assigned to the quaternion  $\hat{q}$ . From this quaternion, supertask 5.2—Calculate  $R$  and  $\vec{t}$ —builds  $R$  and  $\vec{t}$ . Figure 7 shows the supertask interaction graph for the tasks within supertask 5. Find eigenvalues and eigenvectors and calculate  $R$  and  $\vec{t}$ . Algorithm 5 shows the pseudo code for this supertask. The maximum degree of concurrency for this supertask is again 16.

Supertask 6. Apply  $R$  and  $\vec{t}$ .

This supertask applies the estimated rotation and translation to the sensed points. Algorithm 6 shows the pseudo code for this supertask.  $P_k$  is assigned with the points from  $P_0$  with  $R$  and  $\vec{t}$  applied. Figure 8 shows the task interaction graph for the tasks within supertask 6—Apply  $R$  and  $\vec{t}$ . Because this is a one-to-one mapping, the degree of concurrency for this supertask is the number of sensed points.

Supertask 7. Calculate error.

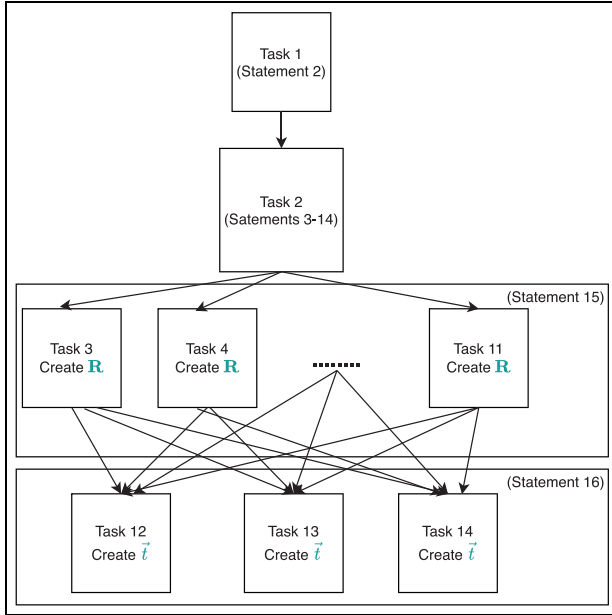
This is the final supertask that calculates the error between  $P_k$  and  $X$ . Algorithm 7 shows the pseudo code for this

**Algorithm 5.** Supertask 5. Find eigenvalues and eigenvectors and calculate  $\mathbf{R}$  and  $\vec{t}$  pseudo code.

```

1: function CALC_RT( $\mathbf{Q}$ ,  $\mu_{\mathbf{p0}}$ ,  $\mu_{\mathbf{y}}$ ,  $\mathbf{R}$ ,  $\vec{t}$ )
2:    $\mathbf{W} = \text{eigenValues}(\mathbf{Q})$ ,  $\mathbf{V} = \text{eigenVectors}(\mathbf{Q})$ 
3:    $j = 0$ 
4:    $w_{\max} = \mathbf{W}[0]$ 
5:    $j_{\max} = 0$ 
6:   for each value  $\in \mathbf{W}$  do
7:     if value  $> w_{\max}$  then
8:        $w_{\max} = \text{value}$ 
9:        $j_{\max} = j$ 
10:    end if
11:     $j = j + 1$ 
12:   end for
13:    $\mathbf{q} = \mathbf{V}[j_{\max}]$ 
14:    $\hat{\mathbf{q}} = \text{normalize}(\mathbf{q})$ 
15:    $\mathbf{R} = \text{rotationFromQuaternion}(\hat{\mathbf{q}})$ 
16:    $\vec{t} = \mu_{\mathbf{y}} - \mathbf{R} \cdot \mu_{\mathbf{p0}}$ 
17: end function

```



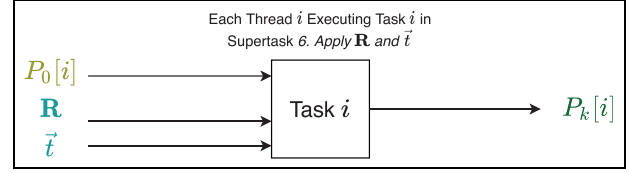
**Figure 7.** Task interaction graph of supertask 5—Find eigenvalues and eigenvectors and calculate  $\mathbf{R}$  and  $\vec{t}$ —outlined in Algorithm 5.

**Algorithm 6.** Supertask 6. Apply  $\mathbf{R}$  and  $\vec{t}$  pseudo code.

```

1: function APPLY_RT( $P_0$ ,  $\mathbf{R}$ ,  $\vec{t}$ ,  $P_k$ )
2:   for each  $\mathbf{p}_0, \mathbf{p}_k \in P_0, P_k$  do
3:      $\mathbf{p}_k = \mathbf{R} \cdot \mathbf{p}_0 + \vec{t}$ 
4:   end for
5: end function

```



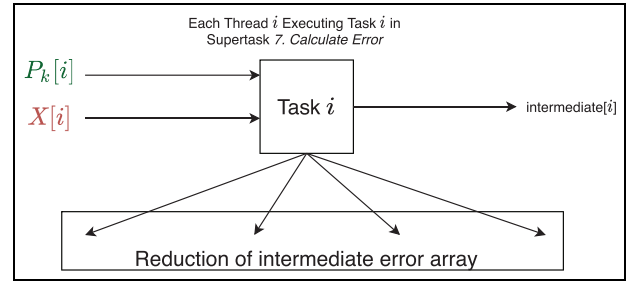
**Figure 8.** Task interaction graph of supertask 6—Apply  $\mathbf{R}$  and  $\vec{t}$ —outlined in Algorithm 6.

**Algorithm 7.** Supertask 7. Calculate error pseudo code.

```

1: function CALC_ERROR( $P_k$ ,  $\mathbf{Y}$ ,  $\text{error}_{rms}$ )
2:    $\text{error}_{rms} = 0$ 
3:   for each  $\mathbf{p}_k, \mathbf{y} \in P_k, \mathbf{Y}$  do
4:      $\text{error}_{rms} = \text{error}_{rms} + |(\mathbf{p}_k - \mathbf{y})|^2$ 
5:   end for
6:    $\text{error}_{rms} = \frac{\text{error}_{rms}}{|P_k|}$ 
7: end function

```



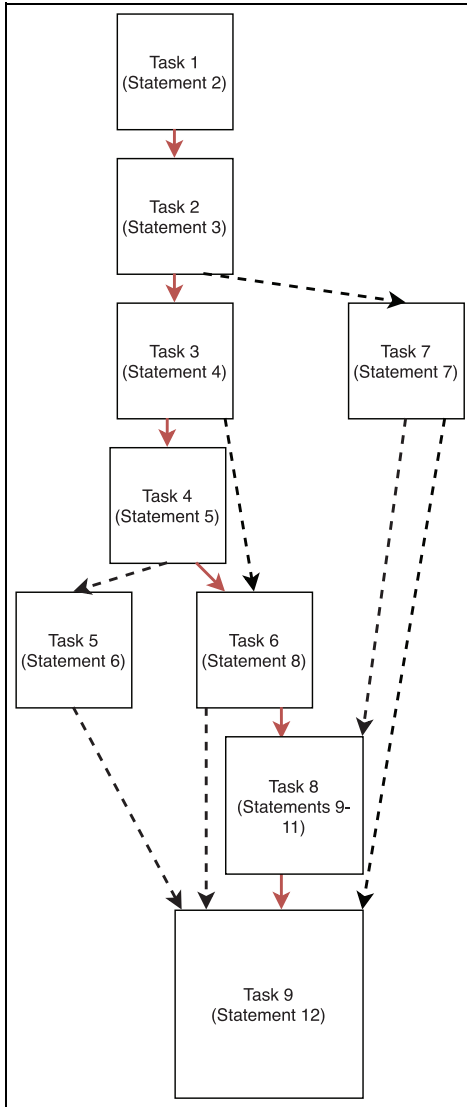
**Figure 9.** Task interaction graph of supertask 7—Calculate error—outlined in Algorithm 7.

supertask. Figure 9 shows the task interaction graph for the tasks within supertask 7—Calculate error. First, the one-to-one mapping portion of this supertask has a degree of concurrency of the number of sensed points. Second, the average degree of concurrency of the binary tree reduction portion of this supertask is again  $\frac{2n-1}{\log_2(n)+1}$ , with  $n$  as the number of sensed points.

The ICP algorithm then iterates through this process until error is lower than a set threshold. The user sets the error threshold to a desired value. The larger the error threshold, the algorithm converges with a lower number of iterations and returns a less accurate estimated pose. The smaller the error threshold, the algorithm converges with a higher number of iterations, but typically yields a pose of increased accuracy.

**3.3.2 Mapping.** Each supertask is statically mapped with respect to mapping the ICP algorithm onto the GPU with





**Figure 10.** Task interaction graph of supertask 4—Calculate  $Q$ —with the critical path in solid red outlined in Algorithm 4.

CUDA. The ICP algorithm is inherently decomposed to reduce thread interaction and idleness within supertasks.

#### Supertask 1. Find correspondences.

This supertask is statically mapped to a kernel<sup>28</sup> launching with  $n$  threads. Again,  $n$  is the number of sensed points. Each thread completes a  $\frac{1}{n}$  part of this task. Algorithm 1 and Figure 4 shows how each thread  $i$  calculates the associated nearest neighbor of  $P_k[i]$  to  $X$ . A mapping based on data partitioning works best for supertask 1—Find correspondences. Because each thread reads  $X$ ,  $X$  is partitioned in a block distribution to take advantage of shared memory and the locality of interaction. This mapping reduces the

read time of  $X$  for each thread. Loading  $X$  into shared memory requires each thread to interact and synchronize after loading  $X$  from global to shared memory. This process reduces the contention on global memory banks and minimizes the size of global data exchange. Finally, this organization completely eliminates interaction between threads and idleness within threads.

#### Supertask 2. Calculate centers of mass.

This supertask is statically mapped to a reduction kernel, launching with  $\frac{n}{2}$  threads. This calculates a single center of mass. This kernel is executed twice and the instances can be run in parallel to calculate the centers of mass of  $P_0$  and  $Y$ . However, the center of mass of  $P_0$  is only calculated on the first ( $k=0$  th) iteration of the accelerated ICP algorithm. In this kernel, threads in each block reduce the data associated with that thread block to a block sum. Then, each block sum is summed and the final sum is divided by the number of elements of the array. This task leverages a block distribution and the utilization of shared memory.

#### Supertask 3. Calculate $\Sigma_{px}$ .

This supertask is first statically mapped to a kernel, launched with  $n$  threads. Each thread calculates each dot product of the  $P_k$  and  $Y$  pair. This is then stored in nine intermediate arrays. These intermediate arrays are summed in reduction kernels launched with  $\frac{n}{2}$  threads, which can be run in parallel. These are then averaged into  $\Sigma_{px}$ . This mapping substantially reduces interaction and idleness between threads in the supertask.

#### Supertask 4. Calculate $Q$ .

This supertask is statically mapped to a kernel that spawns 16 warps of 32 threads. To differentiate the execution of statements, the kernel contains multiple branch statements. However, this introduces the problem of branch divergence among threads in the same warp.<sup>29</sup> To fix this problem, each statement that can be executed in parallel is assigned to a warp that executes independently with an independent instruction scheduler. In addition, threads use shared memory to interact. In Algorithm 4 and Figure 10, statements 2 and 3 are mapped to warp 1. Once those finish and synchronize, warp 1 executes statements 4 and 5. Concurrently, warp 2 executes statement 7. After both those end and synchronize, statement 6 executes on warp 1 and statement 8 on warp 2. After those finish and synchronize, warp 1 executes statements 9–11. Finally after those finish and synchronize, all 16 warps execute statement 12. This supertask does not map efficiently, but using shared memory for interaction reduces the communication overhead. Unavoidably, because a single warp contains 32



threads, each warp may have idle threads throughout the supertask.

Supertask 5. Find eigenvalues and eigenvectors and calculate  $\mathbf{R}$  and  $\vec{t}$ .

The eigenvalues and eigenvectors calculation portion of this supertask is statically mapped to a *cuSolver*<sup>23</sup> kernel launched with  $\mathbf{Q}$ . After this, the remainder of the supertask is statically mapped to a single kernel with nine warps of 32 threads each. The quaternion,  $\hat{q}$ , is calculated by warps 0–3 and placed into shared memory. After synchronization, warps 0–8 assign  $\mathbf{R}$ . Then after synchronization, warps 0–2 assign  $\vec{t}$ . This supertask also does not map efficiently because some threads in the warps are idle. However, shared memory interaction is used for executing threads.

6. Apply  $\mathbf{R}$  and  $\vec{t}$ .

This supertask is statically mapped to a kernel, launching with  $n$  threads. A mapping based on data partitioning is leveraged for this supertask. Each thread  $i$  inside the kernel calculates its own  $P_k[i]$  based on its  $\mathbf{R}$ ,  $\vec{t}$ , and  $P_0[i]$ . Besides memory bank contention, this mapping prohibits thread interaction and thread idleness.

7. Calculate error.

The supertask is statically mapped to a kernel, launching with  $n$  threads. A mapping based on data partitioning is leveraged for this supertask. In addition, a block distribution is used for shared memory interactions between threads during the reduction phase of this supertask. Each thread  $i$  inside the kernel calculates an intermediate error based on  $P_k[i]$  and  $X[i]$ . These intermediate errors are placed into shared memory and then summed in a reduction kernel. This mapping reduces thread interaction and idleness.

**3.3.3 Supertask breakdowns.** All tasks in a single ICP iteration are static in nature. It is known which tasks must be executed to complete each iteration. What is not known, however, is how many iterations must be executed to converge to the error threshold. Because error is not computed until the end of an iteration, the number of iterations executed in the ICP algorithm is dynamic in nature. The iterative nature of the algorithm leads to dynamic task generation. The ICP algorithm has non-uniform tasks. Supertask 1, Find correspondences, is the most computationally expensive supertask and can account for over 90% of the total ICP run time.<sup>6</sup>

A discussion follows about each task required in each supertask, because each supertask executes in sequence. In the ICP algorithm, supertask interactions are irregular,<sup>17</sup> static,<sup>17</sup> and one-way.<sup>17</sup> However, each supertask has threads that exhibit inter-task interactions that are regular.<sup>17</sup>

Supertask 1. Find correspondences.

This supertask is split perfectly between each thread. Each thread executes an equal workload to find the nearest neighbor associated with their thread index. Because of this partitioning, this supertask is split into uniform tasks between the threads. This supertask does have a large data size associated with it. Each thread requires access to  $X$ , a single member of  $P_k$ , and a write in  $Y$ .

In this supertask, the interaction is static and  $P_k$  and  $X$  are read-only.  $Y$  is read-write. Because threads do not interact besides memory bank conflicts, interactions are regular.

Supertask 2. Calculate centers of mass.

This supertask is split into a multitude of non-uniform tasks. In the first portion, the task of calculating both centers of mass is mapped uniformly among threads. However, inside calculating each center of mass, the distribution of work is non-uniform among threads because it is an all-to-one reduction. The all-to-one reduction is mapped such that at the final level of the reduction, a single thread reports the sum and all other threads are idle. The data size associated with this supertask is also large as it includes  $P_k$  and  $Y$ .

In this supertask,  $P_k$  and  $Y$  are read-only. The centers of mass intermediate array reduction is read-write. Because this supertask includes all-to-one reductions, it inherently includes thread interaction.

Supertask 3. Calculate  $\Sigma_{\text{px}}$ .

In this supertask, the dot product is mapped to threads uniformly in a kernel, and the reduction is mapped to threads non-uniformly. The data size associated with this supertask is also large, as it includes  $P_0$  and  $Y$ .

In this supertask, the interaction is static and  $P_0$  and  $Y$  are read-only. The intermediate arrays are read-write. The reduction aspect of this supertask inherently includes thread interaction.

Supertask 4. Calculate  $\mathbf{Q}$ .

This supertask is mapped non-uniformly to threads in a kernel. There are many mini-tasks executing in this supertask that each differ in run time. The data size for this

supertask is small as it only inputs  $\Sigma_{px}$  and the centers of mass, and only outputs  $\mathbf{Q}$ , a  $4 \times 4$  matrix.

In this supertask, the interaction is static and  $\Sigma_{px}$  and the centers of mass are read-only.  $\mathbf{Q}$  matrix is read-write. Interactions between threads are regular, but require finesse to mask with helpful computations. Interactions between threads are timed and coordinated to ensure thread synchronizations minimize thread idleness.

Supertask 5. Find eigenvalues and eigenvectors and calculate  $\mathbf{R}$  and  $\vec{t}$ .

This supertask has minimal workload in the accelerated ICP algorithm. Its data size is small as it only takes in a  $4 \times 4$  matrix and outputs a  $3 \times 3$  matrix  $\mathbf{R}$  and a  $3 \times 1$  vector  $\vec{t}$ .

In this supertask, the interaction is static and  $\mathbf{Q}$  read-only.  $\mathbf{R}$  and  $\vec{t}$  are read-write. Interactions between threads are regular, but require finesse to mask. Again, interactions between threads are timed and coordinated to ensure thread synchronizations minimize thread idleness.

Supertask 6. Apply  $\mathbf{R}$  and  $\vec{t}$ .

This supertask is mapped uniformly to threads in a kernel. Each thread  $i$  applies  $\mathbf{R}$  and  $\vec{t}$  to  $P_0[i]$  and assigns it to  $P_K[i]$ . The data size for this supertask is small as each thread only takes in a  $3 \times 3$  matrix and a  $3 \times 1$  vector and outputs to a single member in  $P_K$ .

In this supertask, the interaction is static and  $P_0$ ,  $\mathbf{R}$ , and  $\vec{t}$  are read-only.  $P_K$  is read-write. Because threads do not interact besides memory bank conflicts, interactions are regular.

Supertask 7. Calculate error.

This supertask is mapped non-uniformly to threads in a kernel as the last portion of finding error is an all-to-one reduction. The data size for this supertask is large as it inputs both  $P_k$  and  $X$  but only outputs a single error value.

In this supertask, the interaction is static and  $P_k$  and  $X$  are read-only. The reported error is read-write. Interactions between threads are again regular. The reduction aspect of this supertask has the highest thread interaction.

**3.3.4. Processing models.** The way the accelerated ICP algorithm is decomposed into tasks is very similar to the pipeline model.<sup>17</sup> However, this pipeline is not being preemptively filled until the error is found. For this reason, the accelerated ICP algorithm employs the manager-worker model,<sup>17</sup> where the CPU is the manager and the GPU is the worker. After each iteration, the CPU receives

the error from the GPU and dynamically decides whether to run another iteration. Additional worker GPUs can be added to the model. Because the NVIDIA RTX 3080 GPU has 8704 CUDA cores, each CUDA core can be considered a worker. As stated previously, in this research, only one GPU is considered. However, as the following describes, each supertask within the ICP algorithm exhibits its own individual model.

Supertasks 1 and 6, Find correspondences and Apply  $\mathbf{R}$  and  $\vec{t}$ , respectively, exhibit the data-parallel model.<sup>17</sup> Figure 4 shows how each thread  $i$  is performing the same set of code to calculate the nearest neighbor of  $X$  for their respective  $P_k[i]$ . Figure 8 shows this same idea.

Supertask 2, Calculate centers of mass, exhibits a task graph model<sup>17</sup> and a data-parallel model. Threads run in parallel based on data, but are also dependent on the reduction process between threads in shared memory space.

A task graph model describes supertasks 3—Calculate  $\Sigma_{px}$ , 4—Calculate  $\mathbf{Q}$ , 5.1—Find eigenvalues and eigenvectors, and 5.2—Calculate  $\mathbf{R}$  and  $\vec{t}$ . These supertasks have many dependencies between threads. However, these supertasks allow for some data-level parallelism between threads.

Supertask 7, Calculate error, exhibits the data-parallel model. Each thread  $i$  within the kernel is performing the code on its own  $P_0[i]$ . This supertask also exhibits a data-parallel model where each thread calculates the error associated with each point between  $X$  and  $P_k$ . However, the reduction portion of this task is more similar to a task graph model in which a multitude of threads reduce down to a single thread at the task completion.

### 3.4. Optimization

**3.4.1. Find correspondences.** The nearest neighbor kernel executing supertask 1, Find correspondences, utilizes the privatization optimization. Stratton et al.<sup>30</sup> explain this optimization. All threads inside each thread block read the same global memory many times. To avoid this, this global memory is replicated among all thread blocks through privatization. From Stratton: “Privatization is the transformation of taking some data that was once common or shared among parallel tasks and duplicating it such that different parallel tasks have a private copy on which to operate.”<sup>30</sup> In this manner, global memory reads are reduced and transformed into shared memory reads. The timing hit from a shared memory access is less than a global memory access. This optimization reduces the run time of the nearest neighbor kernel in the accelerated ICP algorithm implementation.

**3.4.2. Reduction.** The all-to-one reduction<sup>17</sup> kernel, which uses a binary tree-type reduction, is executed 12 times

throughout the accelerated ICP algorithm. For this reason, optimization of this kernel is of utmost importance. The implementation of the reduction kernel is based on Ritcher's reduction kernel optimizations ideas.<sup>31</sup>

First, each thread in the reduction kernel reads two elements from global memory, adds them together, and assigns them to shared memory in the thread block. Thread blocks then perform the remaining reduction in shared memory. This method reduces future memory access overheads in the reduction process. Threads out of bounds of the array assign a zero into shared memory. In addition shared memory bank conflicts are reduced through special indexing. From Ritcher: "Zero padding reduces thread divergence by allowing all the threads to participate in calculations. Shared memory indexing was implemented in a way to reduce shared memory bank conflicts and also reducing thread divergence."<sup>31</sup> The final 32 threads of the reduction can reduce without memory bank conflicts by taking advantage of warp properties on the GPU. Ritcher et al.<sup>31</sup> also explains unrolling any additional loops left over in the reduction to achieve maximum performance. This reduces the instruction count of the kernel by removing the loop check and loop iterator. Ritcher's ideas exemplify many CUDA optimization techniques that have been applied to the reduction kernel and other kernels in the accelerated ICP algorithm.

### 3.5. Algorithm analysis

**3.5.1. Thread communication.** All-to-one reductions frequently appear in the design of this accelerated ICP. Reductions occur inside supertasks 2—Calculate centers of mass, 3—Calculate  $\Sigma_{px}$ , and 7—Calculate error. With respect to the GPU, because these communication patterns are implemented with threads communicating with each other, utilizing shared thread block memory presents opportunities for optimization. However, a reduction is a challenging communication pattern because of the bottleneck of a single thread reporting the result at the finish of each supertask.

For this research, the assumption is made that every multiprocessor on the RTX 3080 is involved in the all-to-one reduction. This research exploits several aspects of GPU architecture in the implementation of all-to-one reductions, including utilizing shared-memory among thread blocks, taking advantage of thread behavior within warps, and optimizing cache hits. From Grama, the cost for the all-to-one reduction is:  $T(p) = (t_s + t_w m) \log(p)$ .<sup>17</sup>  $T$  represents the time to run an all-to-one reduction with  $t_s$  as the startup time or overhead,  $t_w$  as the transfer time or bandwidth,  $m$  as the size of the message, and  $p$  as the number of processors or elements in the reduction.<sup>17</sup> This can be mapped to the RTX 3080 GPU. Again, the RTX 3080 has 8704 CUDA cores and a memory bandwidth of

760.3 GB/s.<sup>32,33</sup> These can be substituted for a more accurate cost:

$$T(8704) = \left( t_s + \frac{1}{760.3 \text{ GB/s}} m \right) \log(8704).$$

To get a precisely accurate  $t_s$ , a profiling would need to be conducted. For this analysis, it is assumed that  $t_s$  is greater than GPU clock rate of 1440 MHz. This gives us our final equation of:

$$T(8704) = \left( \frac{1}{1440 \text{ MHz}} + \frac{1}{760.3 \text{ GB/s}} m \right) \log(8704)$$

for a lower-bound execution run time.

**3.5.2. Interaction overheads.** In supertask 1, Find correspondences, an interaction overhead is the contention for  $X$ . Each thread may need all members of  $X$ . To reduce this overhead, each thread block can initially load a copy of  $X$  into shared memory. Then threads can read from shared memory inside the thread block throughout the rest of the kernel. This approach reduces the global reads of each  $X$  member to only the number of blocks. This method is also called privatization<sup>30</sup> or the replication of data to take advantage of a "private copy"<sup>30</sup> among threads.

In supertask 2, Calculate centers of mass, interaction overheads are the synchronization of threads during the reduction process as well as contention for the input array. The contention is reduced again replicating the input array into shared memory inside the thread blocks and also using memory accesses with striding. The synchronization overhead between threads can be reduced by taking advantage of warp synchronization among threads in the same warp.

In supertask 4, Calculate  $\mathbf{Q}$ , interaction overheads include idleness while waiting for data exchange and contention for data. The contention is reduced by loading thread data to shared memory. The data-exchange overhead is masked with useful computations.

In supertask 5, Find eigenvalues and eigenvectors and calculate  $\mathbf{R}$  and  $\vec{l}$ , interaction overheads also include data contention and waiting for data exchange. For instance, many threads require the max eigenvalue. Because of this, each thread calculates the max eigenvalue. This method replaces interactions with computations. In addition, shared memory is used to store the quaternion, eigenvalues, eigenvectors, and centers of mass. This technique reduces data contention in global memory banks.

For supertask 6, Apply  $\mathbf{R}$  and  $\vec{l}$ , an interaction overhead is the contention for  $P_0$ ,  $\mathbf{R}$ , and  $\vec{l}$ .  $P_0$ ,  $\mathbf{R}$ , and  $\vec{l}$  are replicated among each thread block to reduce global memory bank contention.

In supertask 7, Calculate error, an interaction overhead is the contention for  $P_k$  and  $X$  during the reduction portion. This can be mitigated by performing the reduction in shared memory and also utilizing striding in memory accesses.

**3.5.3. Isoefficiency function.** When building the isoefficiency function, it is assumed that  $m \geq p$  and  $n \geq p$ . Here  $m$  is the number of truth points,  $n$  is the number of sensed points, and  $p$  is the number of processors. Because the accelerated ICP implementation and a serial ICP implementation execute an identical number of iterations, only a single ICP iteration is considered in the run time comparison for simplification purposes.

The parallel run time of supertask 1, Find correspondences, is as follows:

$$\frac{n \log_2(m)}{p} \quad (1)$$

The parallel run time of supertask 2, Calculate centers of mass, is as follows:

$$\frac{2n}{p} \log_2(n) \quad (2)$$

The parallel run time of supertask 3, Calculate  $\Sigma_{px}$ , is:  $\frac{n}{p} + \frac{9n}{p} \log_2(n)$ . Assuming  $p$  is significantly greater than 16, the parallel run time of the portion containing the supertasks 4 and 5, Calculate  $\mathbf{Q}$ , and Find eigenvalues and eigenvectors and calculate  $\mathbf{R}$  and  $\vec{t}$ , respectively, is a constant  $C$  because the maximum degree of concurrency is 16. However, for the serial implementation, this needs to be a run time of  $16C$ . The supertask 6, Apply  $\mathbf{R}$  and  $\vec{t}$  has a parallel run time of  $\frac{n}{p}$ . Finally, the supertask 7, Calculate error, has a parallel run time of  $\frac{n}{p} \log_2(n)$ .

This leads to a parallel run time of:  $\frac{n \log_2(m)}{p} + \frac{2n}{p} + \frac{12n \log_2(n)}{p} + C$ , which also leads to a serial run time of:  $n \log_2(m) + 2n + 12n \log_2(n) + 16C$ . However, an ideal serial algorithm does a single reduction in  $n$ . This makes our serial run time now as follows:

$$n \log_2(m) + 2n + 12n + 16Cn \log_2(m) + 14n + 16C.$$

For the following analysis, it is assumed the run time  $C$  adds is negligible compared to  $n \log_2(m)$ ,  $n$ , and  $\log_2(n)$  weighted terms. From Gustafson,<sup>34</sup> the expected speedup by analysis is the serial run time divided by the parallel run time. With  $C = 0$ , this gives a speedup of:

$$\frac{p(n \log_2(m) + 14n)}{n \log_2(m) + 2n + 12n \log_2(n)} \quad (3)$$

Using the isoefficiency equation,  $\frac{\text{speedup}}{p}$ ,<sup>34</sup> this gives a function of  $\frac{(n \log_2(m) + 14n)}{n \log_2(m) + 2n + 12n \log_2(n)}$ . Assuming

like-weighted terms scale similarly with increased point sizes, there is an extra  $n \log_2(n)$  term in the denominator. This term is a remnant of the serial reduction's increased efficiency compared to the parallel reduction.

**3.5.4. Scalability.** From the speedup function previously stated in Equation (3), an increased number of sensed and truth points allows for an increased number of processors to contribute. Consequently, larger point clouds generally achieve a larger speedup. In addition, increasing the number of processors past the number of sensed points produces diminishing performance results. The reason for this effect is that the maximum degree of concurrency mapped is equal to the number of sensed points, as found in many supertasks similar to supertask 1, Find correspondences.

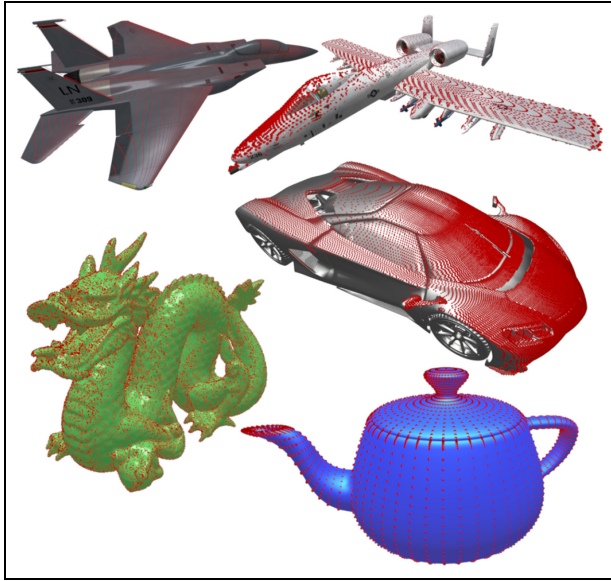
The accelerated ICP algorithm scales efficiently with the number of processors, assuming processors is less than the number of sensed and truth points. The number of sensed points has the largest effect on the run time and speedup of the algorithm. Increasing both the number of processors and the number of points achieves a justified speedup. For instance, a setup with 5k processors and 30k truth and sensed points is expected to achieve a speedup of  $739.02 \times$ .

## 4. Results

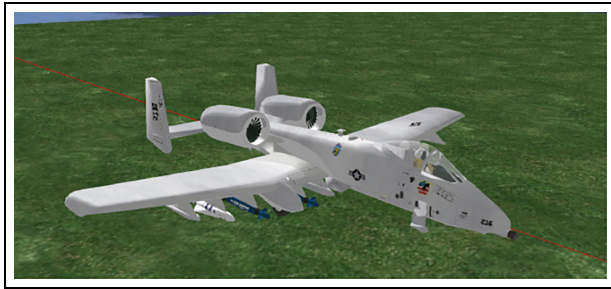
### 4.1. Experimental design

To compare speed, the accelerated ICP GPU implementation is tested against a ICP CPU implementation and another project's ICP GPU implementation. To verify accuracy, our Accelerated ICP GPU implementation is compared against the Point Cloud Library (PCL).<sup>35</sup> Each implementation is tested with a plethora of different models, truth point sets, and sensed point sets. To ensure a fair comparison, the point sets are identical for the tests between each implementation, respectively.

In this paper, our accelerated ICP GPU algorithm implements the PNNOPr,<sup>6</sup> the novel pairwise Euclidean distance nearest neighbor matching algorithm, which utilizes the Delaunay triangulation of the target 3D point set. Once the Delaunay triangulation for a reference point set has been generated, the nearest neighbor of a query point is found by traversing the edges of the Delaunay graph. The algorithm utilizes a greedy approach where at each node of the Delaunay graph, if a connected node is closer than the current visiting node to the query point, the algorithm proceeds to check the nodes connected to the new current visiting mode. This process continues until there are no closer nodes to the query point connected to the current visiting node. Anderson<sup>36</sup> shows how this nearest neighbor algorithm is superior. We invite the reader to watch a video overview of the work detailed in this paper.



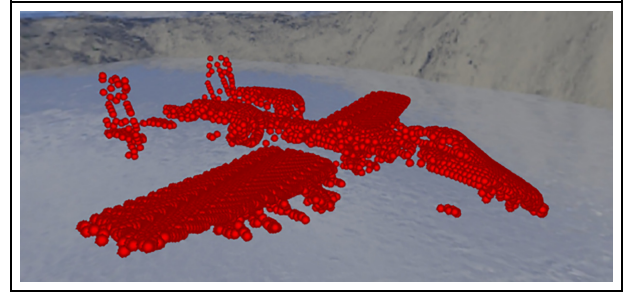
**Figure 11.** Aircraft B (F15), Aircraft A (A10), Sports Car, Dragon, and Teapot models.



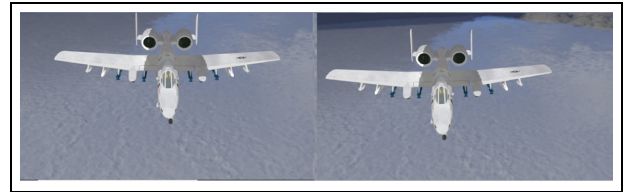
**Figure 12.** Virtual A10 Thunderbolt sensed object used to create the truth model.

Using the previous ICP iteration's nearest neighbor is an efficient way to leverage a known close starting point during the Delaunay graph traversal. However, this strategy cannot be implemented for the first ICP iteration. For this reason, on the first ICP iteration, PNNOPT traverses a k-d tree to the first leaflet and uses this as the starting point for the traversal of the Delaunay graph. All subsequent ICP iterations are started from the previous ICP iteration's nearest neighbor. In contrast, the Previous Nearest Neighbor Delaunay Walk (PNN) does not implement this optimization strategy for the first ICP iteration.

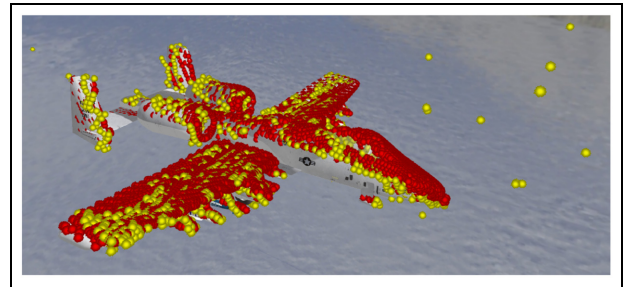
The objects being sensed are an F15 aircraft, A10 aircraft, sports car, dragon, and a teapot, all shown in Figure 11. These objects are each represented by a truth point dataset at two different fidelities, ranging from 4k to 63k points. Figures 12 and 13 show an A10 Aircraft and the associated 5053 point truth model. The specific fidelity used is closest



**Figure 13.** Virtual A10 Thunderbolt red 5053 point truth model generated from the sensed object.



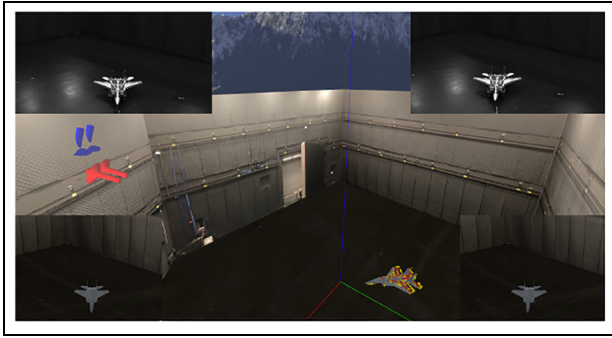
**Figure 14.** Left and right virtual stereo cameras images showing sensed object.



**Figure 15.** ICP running on yellow sensed points and red truth model points.

to the number of sensed points. This also ensured that the number of truth points are greater than or equal to the number of sensed points. The sensed points are generated by the OpenCV C++ library<sup>37</sup> from images of two calibrated virtual stereo cameras through stereo block matching. Figure 14 shows the left and right images of the stereo cameras. Figure 15 shows the accelerated ICP algorithm registering the yellow sensed points generated from stereo block matching from these images.

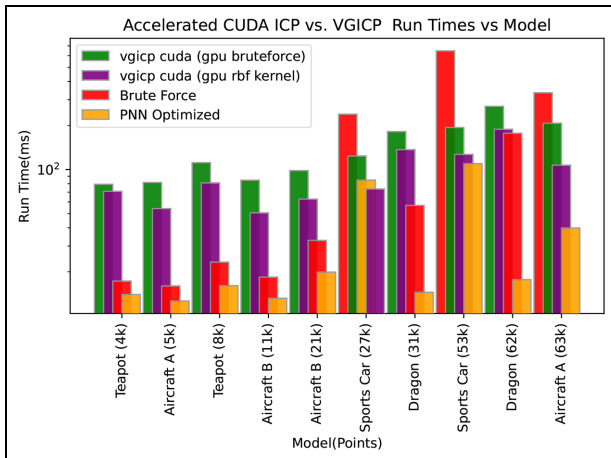
Virtual cameras with parameters and pose set to correspond with their real counterparts captured the 3D environment. Figure 16 shows the virtual images in the lower left and right and the real images in the upper left and right, the sensed points in yellow, and the reference points in red. In the middle left of Figure 16, the red tubes depict



**Figure 16.** Virtual Motion Capture Laboratory mimicking the real Motion Capture Laboratory.



**Figure 17.** Real Motion Capture Laboratory.



**Figure 18.** Accelerated GPU ICP and VGICP run-time comparison.

the location and view direction of the stereo cameras in the real environment and virtual environment—the real

and virtual stereo cameras are co-incident and share the same coordinate frame with respect to the truth motion capture system. Figure 17 shows the real Motion Capture Laboratory.

Next, the accelerated ICP GPU is tested against two competitor GPU ICP algorithms. These competitor algorithms implement the voxelized generalized (VGICP) and the radial basis function (RBF) ICP algorithms<sup>38</sup> on the GPU. Both these algorithms replace the nearest neighbor search with voxelization.

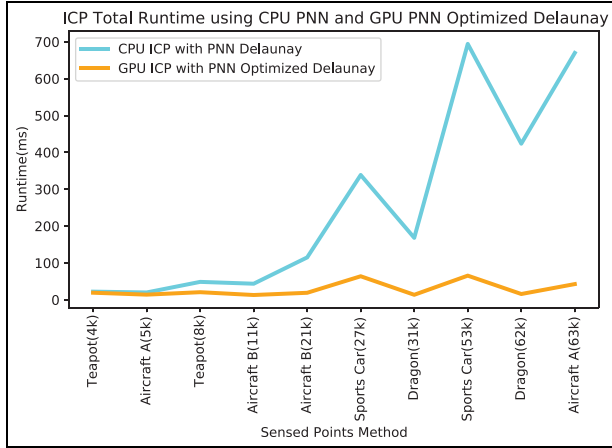
NVIDIA's profiling tool *nvprof*<sup>39</sup> is used to measure kernel timings on the GPU. In addition, the C++ timing library of *std::chrono*<sup>40</sup> is used to time each implementation's overall wall clock time. This wall clock time is reported as an average of all the runs for the particular point sets. Each ICP implementation runs continuously for 100 full convergences with an error threshold of  $1E - 6$ . It is noted, the implementations have identical outputs but differ in speed. The average ICP wall clock time is reported. The profiler *nvprof* also reports the percentage portion each kernel contributes to the total accelerated ICP run time. This tool can be used to gain knowledge on the effect the number of points has on the GPU kernel run times.

## 4.2. Performance

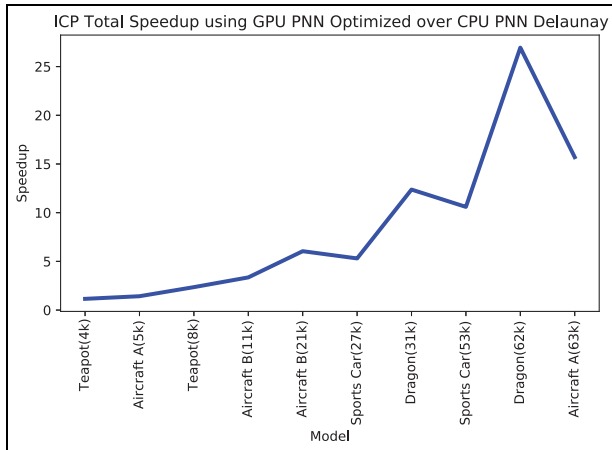
Figure 18 shows the run times of the accelerated ICP using brute force and PNNOPT nearest neighbor against competitors' ICP GPU implementations on all the models while varying the number of truth and sensed points. Our accelerated ICP is substantially faster on all the models, except the Sports Car 27k model. The point cloud topology of the Sports Car model may lead to ICP converging in more iterations than other models, thus a slower run time. The Sports Car model does not have prominent structure away from the center of mass of the model, like the wing tips, vertical stabilizers, and nose cone of aircraft models. In contrast, the Dragon model does have prominent structure away from the center of mass, which may lead to ICP converging in fewer number of iterations. The competitors' VGICP is based on an estimation, which could explain the faster run time on the Sports Car 27k model. However, our PNNOPT ICP was still faster on the Sports Car 53k model. Finally, the point cloud topology of the model would also affect the topology of the Delaunay graph. This would, in turn, affect the number of nodes walked during the PNNOPT nearest neighbor, supertask 1, Find correspondences.

Figure 19 shows a graph of the GPU versus CPU implementation run times. As the number of points increases, the GPU vastly outperforms the CPU. The GPU's run time is stable, while the CPU's run time is increasing at a greater rate.





**Figure 19.** Graph of ICP total run time on the CPU using PNN<sup>4</sup> Delaunay versus GPU using PNNOPT nearest neighbor for supertask I, Find correspondences.



**Figure 20.** Graph of GPU speedup over CPU versus number of sensed points.

Interestingly, with lower point totals, the CPU and GPU run times are close. This similarity can be attributed to the processor the CPU implementation is using having a faster clock rate than the GPU processors. In addition, the GPU has more overheads with memory transfers, kernel invocations, and thread synchronizations. However, these hardware differences and overheads become negligible with larger point totals.

Figure 20 shows the GPU speedup over the CPU with respect to the number of truth and sensed points. The GPU only starts with less than a  $1 \times$  speedup but ends with over a  $25 \times$  speedup. For the CPU implementation, increasing the truth model point total beyond 21,000 points results in an unreasonable run time for real-time applications. However, we still ran the CPU ICP for all the models to get a complete GPU versus CPU run-time comparison. A higher speedup may be achieved with higher point totals and additional optimizations to the nearest neighbor and reduction kernels.

Table 1 shows the portion of ICP total run time the CUDA kernels contributed as well as the run time of those kernels with respect to the number of points. The nearest neighbor kernel starts with 42.47% and ends with 47.26% of the total time. The reduction kernel starts with 37.76% and ends with 34.61% of the total time. As shown here, the nearest neighbor kernel is contributing the largest portion of the GPU run time. This portion is closely followed by the reduction kernel. An interesting observation is as the truth and sensed point sizes become equivalent, the portion the nearest neighbor kernel takes is less than when the point differentials are greater. This decrease can be attributed to increased memory contention for truth points during the matching process. Finally, it can be seen that future optimizations to the accelerated ICP implementation should be focused on the nearest neighbor and reduction kernels.

**Table 1.** This table shows the nearest neighbor and reduction CUDA kernels percentage portion of ICP total run time on the GPU versus number of truth and sensed points.

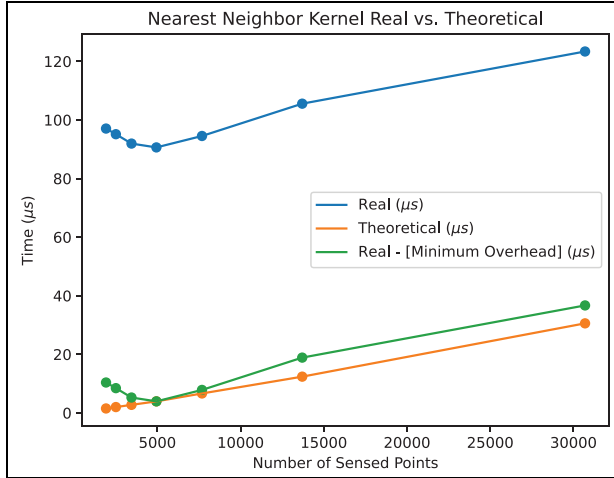
Number of truth points ( $m$ )	Number of sensed points ( $n$ )	Nearest neighbor portion of ICP(%)	Nearest neighbor average run time ( $\mu s$ )	Reduction portion of ICP(%)	Reduction in average run time ( $\mu s$ )
<b>5053</b>	<b>1915</b>	42.47	97.064	37.76	2.696
<b>5053</b>	<b>2507</b>	42.08	95.139	38.19	2.698
<b>5053</b>	<b>3445</b>	41.28	91.938	38.85	2.703
<b>5053</b>	<b>4941</b>	40.74	90.630	38.90	2.704
<b>10,106</b>	<b>7681</b>	41.95	94.530	38.41	2.704
<b>15,159</b>	<b>13,703</b>	44.34	105.560	36.56	2.719
<b>40,424</b>	<b>30,696</b>	47.26	123.370	34.61	2.823

CUDA: compute unified device architecture; ICP: iterative closest point; GPU: graphics processing unit.

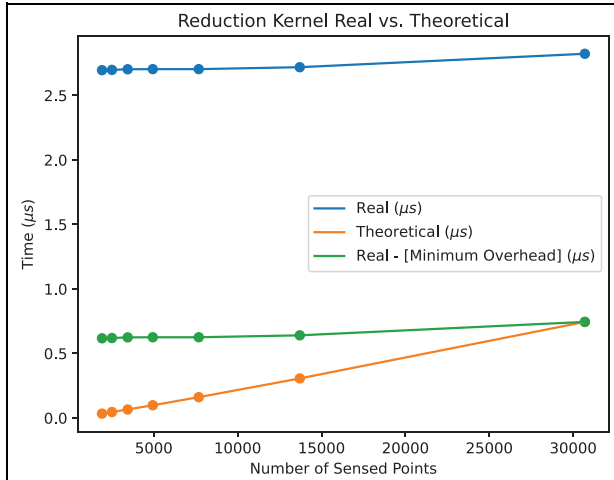
The PNNOPT nearest neighbor was used for supertask I, Find correspondences. The CUDA kernel run times were reported by *nvprof*.

The bolded values represent the independent values on the table, the number of truth and sensed points. They are shaded red for the truth points and yellow for the sensed points to correspond with the same color scheme on figures 13 to 16.





**Figure 21.** Graph of nearest neighbor kernel real and theoretical run times versus number of sensed points.



**Figure 22.** Graph of reduction kernel real and theoretical run times versus number of sensed points.

Figure 21 shows the nearest neighbor kernel's real versus theoretical run times. The theoretical run time was calculated from Equation (1) and divided by the RTX 3080's clock rate of 1440 MHz. In addition, using the compiled parallel thread execution (PTX)<sup>23</sup> file, an estimated instruction count of 400 was used in this calculation. Figure 22 shows the reduction kernel's real versus theoretical run times. The theoretical run time was calculated from Equation (2) and divided by the RTX 3080's clock rate of 1440 MHz and by 2 to account for a single reduction. Again, using the compiled PTX file, an estimated instruction count of 10 was used in this calculation. For both kernels, the overhead was calculated from real run time minus theoretical run time. Then, the minimum overhead was used to shift the real run times.

The theoretical run times estimate a lower-bound execution run time. They show an accurate trend, but do not account for additional overhead factors. Overhead factors that are contributing to the real run times are kernel launch, memory transfers, and warp divergence. For kernels with a smaller workload, the kernel launch is the largest overhead contributing to run time.<sup>41</sup> For kernel runs with smaller point totals, we expect kernel launch time to be the largest overhead. As point totals increase and execution becomes the largest overhead, we expect theoretical and real run time to converge.

When accounting for overhead in the nearest neighbor kernel, the trend of real run times closely matches the trend of the theoretical run times. However, the first few points do not match this trend. The reason for this could be because the RTX 3080's 8704 cores were not saturated until 8704 or more points. The data point with 5053 sensed points is close to core saturation. In addition, non-deterministic instruction and warp scheduling could factor into this anomaly. Again, the theoretical run time serves as a lower bound for real run time.

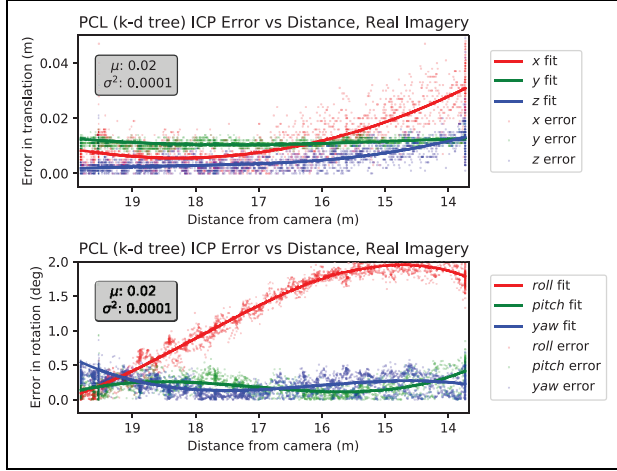
In the reduction kernel, the trend of real run times closely matches the trend of theoretical run times when accounting for overhead. The first few data points do not follow the theoretical trend as much as the data points with higher point totals. The first few data points from the nearest neighbor kernel show a similar anomaly. The same reasons for this anomaly hold true here. Higher point totals verify the theoretical trend lower bound of real run time.

### 4.3. Accuracy

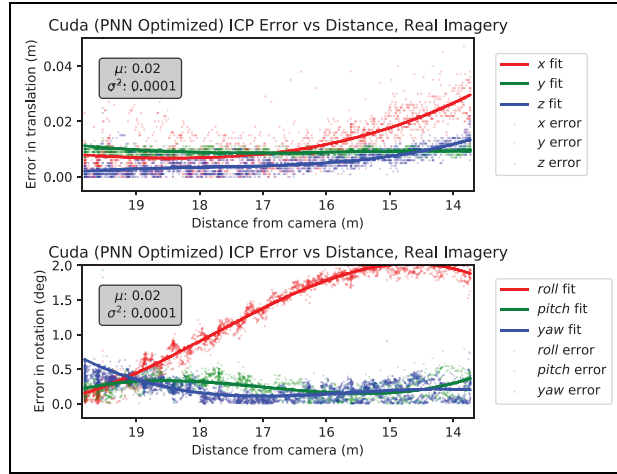
As seen in Figures 23 and 24, when registering points generated from real images, our accelerated GPU ICP and PCL implementations perform with the same accuracy. Our implementation and PCL report less than a magnitude of 2.4 centimeters in positional error and a magnitude of 2 degrees of rotational error. Table 2 shows the positional and rotational mean magnitude error of our accelerated ICP using brute force and PNNOPT nearest neighbors against competitors' ICP GPU implementations on real images. Our implementation is substantially more accurate.

## 5. Conclusion

In this study, the ICP algorithm was rephrased to a massively parallel algorithm and implemented and mapped to an NVIDIA RTX 3080 GPU. The algorithm, implementation, and parallelization approaches for each step of the algorithm were discussed. Our accelerated ICP GPU algorithm was compared against competitors' ICP GPU implementations. In addition, the wall clock time and speedup with respect to the CPU implementation were compared to



**Figure 23.** Errors in translation and rotation using the PCL ICP on real imagery.



**Figure 24.** Errors in translation and rotation using the accelerated ICP on real imagery.

achieve over a  $25 \times$  speedup. In addition, the specific portion and run time of the CUDA kernels were profiled. Finally, an algorithmic analysis and comparison was conducted of the parallel algorithm between theoretical and real run times.

## 6. Future work

Future work will run experiments with higher point totals and also match the number of truth and sensed points. This would give insight into how the algorithm performs in ideal conditions and show if theoretical and real run times completely converge. In addition, a study on ICP convergence and run-time performance in relation to model

**Table 2.** This table shows the Translation (m) and Rotation (deg) magnitude errors of our CUDA ICP versus the CUDA VGICP on the GPU.

ICP algorithm	Aircraft A 63k		Aircraft A 5k		Dragon 31k		Dragon 62k		Aircraft B 11k		Aircraft B 21k		Sports Car 27k		Sports Car 53k		Teapot 4k		Teapot 8k	
	ICP brute force	ICP PNNOpt	VGICP brute force	VGICP rbf kernel	ICP brute force	ICP PNNOpt	VGICP brute force	VGICP rbf kernel	ICP brute force	ICP PNNOpt	VGICP brute force	VGICP rbf kernel	ICP brute force	ICP PNNOpt	VGICP brute force	VGICP rbf kernel	ICP brute force	ICP PNNOpt	VGICP brute force	VGICP rbf kernel
Translation errors (m)	0.000	0.000	0.038	0.014	0.000	0.000	0.016	0.076	0.000	0.000	0.033	0.045	0.000	0.000	0.001	0.001	0.000	0.000	0.000	0.000
Rotation errors (deg)	0.000	0.000	0.001	0.003	0.000	0.000	0.000	1.743	0.000	0.000	0.020	0.019	0.001	0.001	0.002	0.002	0.000	0.000	0.000	0.000
	0.003	0.003	0.004	0.004	1.001	0.977	0.000	0.020	0.016	0.000	0.000	0.020	0.077	0.311	0.079	0.008	0.006	0.006	0.002	0.002

CUDA: compute unified device architecture; ICP: iterative closest point; VGICP: voxelized generalized ICP; GPU: graphics processing unit; PNNOpt: previous nearest neighbor optimized Delaunay walk.

topology should be conducted. It is planned to run the experiments with different CPUs and GPUs. This includes an application using the NVLink interconnect with multiple GPUs. In addition, more optimizations to various kernels are planned to increase the performance of the accelerated ICP algorithm. Finally, an ideal comparison between the serial and parallel algorithms would compare identical processors for the serial and parallel implementations, respectively.



### Acknowledgments

The authors wish to thank Dan Schreiter, and the rest of AFRL/RQ Aerospace Systems Directorate for their support and feedback throughout this experiment.

### Funding

The author(s) received no financial support for the research, authorship, and/or publication of this article.

### ORCID iDs

Ryan M Raettig  <https://orcid.org/0000-0002-4809-5382>  
 Scott L Nykl  <https://orcid.org/0000-0002-4271-2132>

### References

- Konolige K. Small vision systems: hardware and implementation. In: Shirai Y and Hirose S (eds) *Robotics research*. London: Springer, 1998. pp. 203–212.
- Myronenko A and Song X. Point set registration: coherent point drift. *IEEE Trans Pattern Anal Mach Intell* 2010; 32: 2262–2275.
- Stamper RL, Lieberman MF and Drake MV. Visual field theory and methods. In: Stamper RL, Lieberman MF and Drake MV (eds) *Becker-Shaffer's Diagnosis and Therapy of the Glaucomas*. 8th ed. Edinburgh: Mosby, pp. 91–97.
- Raettig RM. *Accelerating point set registration for automated aerial refueling*. Air force institute of technology Wright-Patterson, AFB OH, 2021.
- Besl PJ and McKay ND. Method for registration of 3-d shapes. In: *Sensor fusion IV: control paradigms and data structures*, vol. 1611, Boston, MA, 14–15 November 1991, pp. 586–606. Bellingham, WA: International Society for Optics and Photonics.
- Anderson JD, Taylor CN and Wischgoll T. Delaunay walk for fast nearest neighbor: accelerating correspondence matching for ICP. *Mach Vis Appl* 2022; 33: 31.
- Park SY and Subbarao M. An accurate and fast point-to-plane-registration technique. *Pattern Recognit Lett* 2003; 24: 2967–2976.
- Blais G and Levine MD. Registering multiview range data to create 3d computer objects. *IEEE Trans Pattern Anal Mach Intell* 1995; 17: 820–824.
- Chen Y and Medioni G. Object modeling by registration of multiple range images. In: *IEEE international conference on robotics and automation*, Sacramento, CA, 9–11 April 1991.
- Cover T and Hart P. Nearest neighbor pattern classification. *IEEE Trans Inform Theory* 1967; 13: 21–27.
- Garcia V, Debreuve E and Barlaud M. Fast k nearest neighbor search using GPU. In: *IEEE computer society conference on computer vision and pattern recognition workshops*, Anchorage, AK, 23–28 June 2008, pp. 1–6. New York: IEEE.
- Friedman JH, Bentley JL and Finkel RA. An algorithm for finding best matches in logarithmic expected time. *ACM Trans Math Softw* 1977; 3: 209–226.
- Sproull RF. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica* 1991; 6: 579–589.
- Shibata T, Kato T and Wada T. K-d decision tree: an accelerated and memory efficient nearest neighbor classifier. In: *Third IEEE international conference on data mining*, Melbourne, FL, 22–22 November 2003, pp. 641–644. New York: IEEE.
- Robinson J, Piekenbrock M, Burchett L, et al. Parallelized iterative closest point for autonomous aerial refueling. In: *International symposium on visual computing*, Las Vegas, NV, 12–14 December 2016, pp. 593–602. Cham: Springer.
- Langis C, Greenspan M and Godin G. The parallel iterative closest point algorithm. In: *Proceedings third international conference on 3-D digital imaging and modeling*, Quebec City, QC, Canada, 28 May 2001–1 June 2001, pp. 195–202. New York: IEEE.
- Gram A, Kumar V, Gupta A, et al. *Introduction to Parallel Computing*. London: Pearson Education, 2003.
- Rahman MM, Galanakou P and Kalantzis G. A fast GPU point-cloud registration algorithm. In: *IEEE/ACIS international conference on software engineering, artificial intelligence, networking and parallel/distributed computing (SNPD)*, Busan, South Korea, 27–29 June 2018.
- Golub GH and Reinsch C. Singular value decomposition and least squares solutions. *Numer Math* 1970; 14: 403–420.
- Tamaki T, Abe M, Raytchev B, et al. Softassign and EM-ICP on GPU. In: *First international conference on networking and computing*, Higashi, Japan, 17–19 November 2010.
- Granger S and Pennec X. Multi-scale EM-ICP: a fast and robust approach for surface registration. In: *European conference on computer vision*, Copenhagen, Denmark, 28–31 May 2002, pp. 418–432. Berlin: Springer.
- NVIDIA. NVIDIA CUDA compute unified device architecture programming guide (Version 1.0). [http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA\\_CUDA\\_Programming\\_Guide\\_1.0.pdf](http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf) (accessed 25 June 2020).
- NVIDIA. CUDA toolkit documentation (V11.0.171). <https://docs.nvidia.com/cuda/>
- Oden L. Lessons learned from comparing C-CUDA and python-numba for GPU-computing. In: *28th euromicro international conference on parallel, distributed and network-based processing (PDP)*, Västerås, Sweden, 11–13 March 2020, pp. 216–223. New York: IEEE.
- NVIDIA. Compare current and previous GeForce series of graphics cards. <https://www.nvidia.com/en-us/geforce/graphics-cards/compare/?section=compare-specs> (accessed 13 June 2022).
- Database TG. NVIDIA GeForce RTX 3080 specs. <https://www.techpowerup.com/gpu-specs/geforce-rtx-3080.c3621> (accessed 13 June 2022).

27. NVIDIA. NVIDIA ampere GA102 GPU architecture. <https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf> (accessed 13 June 2022).
28. Awatramani M, Zambreno J and Rover D. Increasing GPU throughput using Kernel interleaved thread block scheduling. In: *IEEE 31st international conference on computer design (ICCD)*, Asheville, NC, 6–9 October 2013, pp. 503–506. New York: IEEE.
29. Fung WWL, Sham I, Yuan G, et al. Dynamic warp formation and scheduling for efficient GPU control flow. In: *40th Annual IEEE/ACM international symposium on microarchitecture (MICRO 2007)*, Chicago, IL, 1–5 December 2007, pp. 407–420. New York: IEEE.
30. Stratton JA, Anssari N, Rodrigues C, et al. Optimization and architecture effects on GPU computing workload performance. In: *Innovative parallel computing (InPar)*, San Jose, CA, 13–14 May 2012, pp. 1–10. New York: IEEE.
31. Richter E, Raettig R, Mack J, et al. Accelerated shadow detection and removal method. In: *2019 IEEE/ACS 16th international conference on computer systems and applications*, Abu Dhabi, United Arab Emirates, 3–7 November 2019, pp. 1–8. New York: IEEE.
32. NVIDIA. GeForce RTX 30 series graphics card overview. <https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/> (accessed 29 June 2022).
33. TechPowerUp. GPU database–NVIDIA GeForce RTX 3080 specs. <https://www.techpowerup.com/gpu-specs/geforce-rtx-3080.c3621> (accessed 3 November 2022).
34. Gustafson JL. Reevaluating Amdahl's law. *Commun ACM* 1988; 31: 532–533.
35. Rusu RB and Cousins S. 3D is here: Point Cloud Library (PCL). In *IEEE international conference on robotics and automation*, Shanghai, China, 9–13 May 2011.
36. Anderson J. Delaunay walk for fast nearest neighbor matching, 2021. <https://youtu.be/wl6MiRx-5Ik> (accessed 14 April 2021).
37. Kaehler A and Bradski G. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV library*. Sebastopol, CA: O'Reilly Media, Inc., 2016.
38. Koide K, Yokozuka M, Oishi S, et al. Voxelized GICP for fast and accurate 3d point cloud registration. *EasyChair Preprint No. 2703*, 2020.
39. NVIDIA. Profiler: CUDA toolkit documentation. <https://docs.nvidia.com/cuda/profiler-users-guide/index.html> (accessed 27 June 2020).
40. Josuttis NM. *The C++ Standard Library: A Tutorial and Reference*. Boston, MA: Addison-Wesley Professional, 2012.
41. Zhang L, Wahib M and Matsuoka S. Understanding the overheads of launching CUDA Kernels. In: *ICPP19*, 5–8 August 2019, Kyoto, Japan.

## Author biographies

**Ryan M. Raettig** is a PhD student at the Air Force Institute of Technology (AFIT). His areas of interest are parallel computing, computer vision, and alternative navigation. He received his MS degree in Computer Engineering at AFIT in 2021, Distinguished Graduate, and a BS degree in Electrical & Computer Engineering at the University of Arizona in 2019, summa cum laude.

**James D. Anderson** received his Master's degree in computer engineering in 2018 from Wright State University in Dayton, Ohio, and is working on his PhD at the same institution. His research work is in collaboration with researchers at the Air Force Institute of Technology (AFIT) on simulating and analyzing flights involving automated aerial refueling. His research involves real-time visualization and image processing, as well as virtual and augmented reality environments.

**Scott L. Nykl** is an Associate Professor of Computer Science at the Air Force Institute of Technology. His areas of interest are real-time 3D computer graphics, computer vision, sensor fusion, parallel processing, interactive virtual worlds, and computer networking. He authored a 3D visualization engine and created avionics-related visualizations earning several national awards including mention in *Forbes*'s "The Greatest Young Inventors In America." He received his BS degree in Software Engineering at UW-Platteville in 2006 and his MS and PhD degrees in Computer Science from Ohio University in 2008 and 2013, summa cum laude.

**Laurence D. Merkle** received his BS degree in computers and systems engineering from Rensselaer Polytechnic Institute, Troy, NY, USA, in 1987, and the MSCE and PhD degrees in computer engineering from the Air Force Institute of Technology (AFIT), Wright-Patterson Air Force Base, OH, USA, in 1992 and 1996, respectively. He is currently an Assistant Professor of Computer Science with the AFIT. His research interests include quantum computing, evolutionary computation, and computer science education.