Journal of Real-Time Image Processing
DOI 10.1007/s11554-017-0668-5

Ahmed Ellliethy, Gaurav Sharma
University of Rochester

Accelerated Parametric Chamfer Alignment Using a Parallel, Pipelined GPU Realization

Received: 15 July 2016 / Accepted: 16 January 2017

Abstract Parametric chamfer alignment (PChA) is commonly employed for aligning an observed set of points with a corresponding set of reference points. PChA estimates optimal geometric transformation parameters that minimize an objective function formulated as the sum of the squared distances from each transformed observed point to its closest reference point. A distance transform enables efficient computation of the (squared) distances and the objective function minimization is commonly performed via the Levenberg-Marquardt (LM) non-linear least squares iterative optimization algorithm. The point-wise computations of the objective function, gradient, and Hessian approximation required for the LM iterations make PChA computationally demanding for large scale data sets.

We propose an acceleration of the PChA via a parallelized and pipelined realization that is particularly well-suited for large scale data sets and for modern GPU architectures. Specifically, we partition the observed points among the GPU blocks and decompose the expensive LM calculations in correspondence with the GPU's single instruction multiple thread (SIMT) architecture to significantly speed-up this bottleneck step for PChA on large-scale datasets. Additionally, by re-ordering computations, we propose a novel pipelining of the LM algorithm that offers further speed-up by exploiting the low arithmetic latency of the GPU compared with its high global memory access latency. Results obtained on two different platforms for both 2D and 3D large-scale point data sets from our ongoing research demonstrate that the proposed PChA GPU implementation provides a significant speed-up over its single CPU counterpart.

Keywords
Chamfer alignment, pipelining, parametric registration, GPU acceleration.

Ahmed Elliethy and Gaurav Sharma
Dept. of Electrical and Computer Engineering, University of Rochester, Rochester, NY, 14627
E-mail: {ahmed.s.elliethy, gaurav.sharma}@rochester.edu

1 Introduction

Alignment of an observed set (OS) of points with a corresponding reference set (RS) of points is an important task in many computer vision applications such as object localization [1], multi-modality medical data registration [2], stereo matching [3], object modeling [4], range data registration [5, 6, 7], and geo-registration of aerial wide area motion imagery (WAMI) frames [8, 9]. Usually the observed points and the reference points are obtained from different views for the same scene/object and to fuse the useful information contained in both views, an alignment must be performed. The goal of the alignment is to estimate the parameters of the geometric transformation that maps the observed points in the OS to the coordinate system of the reference points in the RS.

Existing point set alignment algorithms can be coarsely divided into two categories: correspondence based alignment and correspondence free alignment. Correspondence based alignment algorithms usually iterate between two steps. The first step estimates the correspondences between the points in the OS and the points in the RS. In the second step, the parameters of the geometric transformation that map each observed point in the OS to its corresponding reference point in the RS are estimated. The estimated correspondences can be either hard assignments as in the iterative closest point algorithm [10, 11], or soft assignments as in the expectation maximization like algorithms [12, 13].

Correspondence free alignment algorithms [14, 15] on the other hand, avoid the correspondence estimation step by modeling the alignment metric based on a different representation of the point sets. For example, the distance transform (DT) [16] is used in [14] to represent the RS, and the alignment is achieved by finding the optimal geometric transformation parameters that minimize an objective function formulated as the sum of the distances between each transformed observed point to its closest point in the RS, where these distances are computed efficiently using the DT. The formulated objective function is commonly minimized via the Levenberg-Marquardt (LM) [17] non-linear least squares iterative optimization algorithm. Motivated by the terminology for the first such algorithms[18], we refer to this technique as "Parametric Chamfer Alignment" (PChA).

Despite the computational efficiency of the DT, the overall computational cost of the PChA grows with the number of points in the OS, because of the *point-wise* computations required in each iteration of the LM algorithm. Specifically, in each iteration, the LM algorithm performs the following steps: (1) computes an estimate of the Hessian matrix and the gradient vector of the objective function at the current estimate of the alignment parameters, (2) solves a linear system of equations defined in terms of the computed Hessian and gradient to estimate the update to the alignment parameters, (3) generates a candidate solution using the estimated parameter update and (4) decides whether or not to update the current estimate to the candidate parameters by evaluating and comparing the objective function at both the candidate and current parameters. The objective function evaluation (Step 4) and the Hessian-gradient calculations (Step 1) have to be computed per observed point, and these steps represent a computational bottleneck for achieving real/near-real time performance for large scale point data sets.

Modern day graphical processing units (GPUs) provide computational hardware that can easily perform billions of general-purpose floating-point operations per second. In this paper, we propose an acceleration of the PChA via a parallelized and pipelined realization on GPU platforms. Specifically, we map the LM calculations to the GPU by partitioning the observed points and decomposing the calculations associated with the objective function, the Hessian matrix, and the gradient vector into independent, per-partition, computations that align well with GPU's single instruction multiple thread (SIMT) architecture and therefore run efficiently on the GPU. Additionally we propose a novel pipelining of the LM algorithm that benefits from the low arithmetic latency compared with the high global memory access latency of the GPU by combining the evaluation of the objective function at the candidate solution (Step 4 above) with the Hessian matrix and the gradient vector calculations evaluated at that candidate solution (Step 1 above). Our pipelined LM allows the gradient-Hessian calculations to be pre-computed relatively inexpensively (in terms of run time) because the gradient-Hessian calculations use the data that is already fetched from GPU global memory in order to evaluate the objective function at the candidate solution.

There are previous efforts for accelerating individual components of the PChA using the GPU. For example, in [19, 20], a GPU based implementation of DT is proposed. Although the significant speed-up achieved by the DT implementation on the GPU compared to its CPU counterpart, the PChA performance is still limited by the running time of the LM algorithm, because the DT is computed only once before performing the iterative and expensive LM algorithm steps. In [21], a GPU based implementation of the LM is proposed that concurrently performs a number of LM model fittings such that each LM model fitting uses a single GPU computational block. However, the shared memory per GPU computational block is limited and therefore this LM implementation is not suitable for our large scale alignment problem. In [22], an OpenGL environment based GPU implementation of the LM algorithm is proposed. Because there is no mechanism available for communication between the computational units (shaders) in the OpenGL environment [23], the implementation uses inefficient multiple texture read and write passes for data sharing, and thus does not completely benefit from the available computational power of modern GPUs.

In this paper, we provide an efficient parallelized and pipelined realization of the PChA on GPU that uses the Compute Unified Device Architecture (CUDA^{TM}) NVIDIA^{TM} GPU programming language and efficiently exploits the GPU's computational cores and the characteristics of the GPU memory hierarchy to achieve a significant performance speed-up compared to the CPU implementation. Our contributions in this paper are a partitioning and decomposition scheme that handles the expensive LM computations efficiently on the GPU and our novel pipelining of the LM algorithm. We demonstrate our GPU based implementation of the PChA on both 2D and 3D point large-scale data sets that are related to ongoing research from our group on WAMI aerial image geo-registration [8, 9] and homologous building analyses using range imaging [7, 24]. In these applications, our GPU based implementation of the PChA executed on two different computer systems shows a significant performance speed-up compared to its single CPU counterpart.

This paper is organized as follows. Section 2 presents a brief overview of the PChA. In Section 3, we describe our GPU based implementation of the PChA, where we discuss our partitioning and decomposition scheme along with our novel pipelining of the LM algorithm. Section 4 describes the GPU and CPU implementations and the hardware configurations that we use for benchmarking. Results and comparison against the CPU-based PChA implementation evaluated on applications from our ongoing research, are presented in Section 5. Section 6 is a discussion of the results along with a discussion of the factors that contribute to variability across different parametric transformations. Section 7 summarizes our concluding remarks.

2 Parametric chamfer alignment

To formulate the PChA, we begin with the following abstract setting. In $\mathbb{R}^n$, we are given a reference set (RS) of N_q data points $\mathbf{Q} = \{\mathbf{q}_j\}_{j=1}^{N_q}$, $\mathbf{q}_j \in \mathbb{R}^n$ and a corresponding observed set (OS) of N_p points $\mathbf{P} = \{\mathbf{p}_i\}_{i=1}^{N_p}$, $\mathbf{p}_i \in \mathbb{R}^n$, where the majority of the observed data points result from observations of the reference data points after a parametric geometric transformation $\mathcal{T}_{\bar{\boldsymbol{\alpha}}} : \mathbb{R}^n \to \mathbb{R}^n$ has been applied where $\bar{\boldsymbol{\alpha}} = [\bar{\alpha_1}, \ldots, \bar{\alpha_L}]^T$ is an L-dimensional vector of the transformation parameters and L depend on the geometric transformation type. This abstraction represents a number of common situations in computer vision and image processing tasks, where the reference and the observed data points are observations of a common scene/object from different viewpoints. Due to differences in sampling and potentially missing observations the number of points need not be identical between the RS and OS. For example, when a single geometric 3D object is observed via range imaging from two different view points related via rotation and translation (Euclidean geometric transformation), $\bar{\boldsymbol{\alpha}}$ would be a 6 parameter vector, i.e., $L = 6$, that corresponds to the three rotation angles and the three translations across the three dimensions.

To align the data points in the OS with the data points in the RS, PChA computes an optimal estimate $\boldsymbol{\alpha}^*$ of the geometric transformation parameters $\bar{\boldsymbol{\alpha}}$ by minimizing the objective function

$$f(\boldsymbol{\alpha}) = \sum_{j=1}^{N_p} \|\mathbf{r}\left(\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)\right)\|^2, \tag{1}$$

where $\mathbf{r}\left(\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)\right)$ represent the residual vector defined as the difference between the transformed location $\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)$ of the j^{th} observed point $\mathbf{p}_j$ under the geometric transformation $\mathcal{T}_{\boldsymbol{\alpha}}$ and the corresponding closest point in the RS $\mathbf{Q}$. Mathematically,

$$\mathbf{r}\left(\boldsymbol{\chi}\right) = \boldsymbol{\mu}_{\mathbf{Q}}\left(\boldsymbol{\chi}\right) - \boldsymbol{\chi}, \tag{2}$$

where

$$\boldsymbol{\mu}_{\mathbf{Q}}\left(\boldsymbol{\chi}\right) = \arg\min_{\mathbf{q}\in\mathbf{Q}} \|\mathbf{q} - \boldsymbol{\chi}\|^2, \tag{3}$$

is the closest point in the RS $\mathbf{Q}$ for a point $\boldsymbol{\chi} \in \mathbb{R}^n$.

The minimization of the objective function (1) is commonly performed via the Levenberg-Marquardt (LM) [17] non-linear least squares iterative optimization algorithm. In each iteration, the LM algorithm estimates the parameter update vector $\boldsymbol{\delta} \in \mathbb{R}^{L\times 1}$ such that the value of the objective function is reduced when moving from $\boldsymbol{\alpha}$ to $\boldsymbol{\alpha} + \boldsymbol{\delta}$, with the parameters converging to a minima of the objective function with the progression of iterations. The parameter update vector $\boldsymbol{\delta}$ is obtained by solving the linear system of equations

$$(\mathbf{H} + \lambda\mathbf{I})\boldsymbol{\delta} = -\mathbf{g}, \tag{4}$$

where λ is a non-negative damping parameter automatically adjusted at each iteration [17] to determine the step size, $\mathbf{I}$ is the identity matrix, and $\mathbf{g} \in \mathbb{R}^{L\times 1}$ is the gradient of $f(\boldsymbol{\alpha})$, computed as

$$\mathbf{g} = \frac{\partial f}{\partial \boldsymbol{\alpha}} = -2\sum_{j=1}^{N_p} \mathbf{J}_j^T \mathbf{r}\left(\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)\right), \tag{5}$$

where $\mathbf{J}_j \in \mathbb{R}^{n\times L}$ is the Jacobian matrix computed at the transformed point $\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)$, computed as

$$\begin{aligned}\mathbf{J}_j = \frac{\partial \mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)}{\partial \boldsymbol{\alpha}} &= \left(\frac{\partial \mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)}{\partial \alpha_1}, \ldots, \frac{\partial \mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)}{\partial \alpha_L}\right) \\ &= \begin{pmatrix} J_{j,1}^1 & J_{j,2}^1 & \cdots & J_{j,L}^1 \\ J_{j,1}^2 & J_{j,2}^2 & \cdots & J_{j,L}^2 \\ \vdots & \vdots & \ddots & \vdots \\ J_{j,1}^n & J_{j,2}^n & \cdots & J_{j,L}^n \end{pmatrix},\end{aligned} \tag{6}$$

and $\mathbf{H} \in \mathbb{R}^{L\times L}$ is the approximation of the Hessian matrix, obtained as

$$\mathbf{H} = \sum_{j=1}^{N_p} \mathbf{J}_j^T \mathbf{J}_j. \tag{7}$$

The explicit expressions for the Jacobian matrix entries $\{J_{j,l}^c : c \in \{1, \ldots, n\}, l \in \{1, \ldots, L\}\}$, for $n = 2$ and for different types of geometric transformations, are provided in Table 1.

Motivating PChA example

To help visualize the steps within the PChA algorithm, we use a motivating example from our recent work on geo-registration of aerial WAMI frames [8, 9]. We achieve the geo-registration of a WAMI frame by (1) detecting vehicle locations in the WAMI frame using vehicular motion and (2) posing the WAMI frame geo-registration as the problem of finding the geometric transformation that aligns the detected vehicle locations with a geo-referenced vector road network. Examples of an aerial WAMI frame and a vector road network are shown in Fig. 1 (a) and (c) respectively. We adopt PChA to efficiently solve the problem of aligning vehicle detections with the vector road network [8], where in this scenario, the OS comprises vehicle detections locations shown as white points in the image shown in Fig. 1 (b), while the RS comprises the locations corresponding to the network of roads shown as white points in Fig. 1 (c).

The distance transform (DT) [16, 25] efficiently calculates the residual vectors (2) between each vehicle detection point in the OS to its closest road network point in the RS, where these residual vectors are used in the calculations of both the objective function (1) and the gradient vector (5). Specifically, in the implementation we consider, the DT pre-computes and stores as a "look up table" the residual vector $\mathbf{r}\left(\boldsymbol{\chi}\right)$ between any point $\boldsymbol{\chi} \in \Omega$ to its closest point $\boldsymbol{\mu}_{\mathbf{Q}}\left(\boldsymbol{\chi}\right)$ in the RS, where

Transformation type	Parameters	Relation $\mathbf{p}'_j = \mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)$	$J^c_{j,l}$		
			$c=1$	$c=2$	l
Euclidean	$\boldsymbol{\alpha} = [\theta, t_x, t_y]$	$\begin{pmatrix} x'_j \\ y'_j \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_j \\ y_j \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$	$-R^2_\theta(\mathbf{p}_j)$	$R^1_\theta(\mathbf{p}_j)$	1
			1	0	2
			0	1	3
Similarity	$\boldsymbol{\alpha} = [s, \theta, t_x, t_y]$	$\begin{pmatrix} x'_j \\ y'_j \end{pmatrix} = s \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_j \\ y_j \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$	$R^1_\theta(\mathbf{p}_j)$	$R^2_\theta(\mathbf{p}_j)$	1
			$-sR^2_\theta(\mathbf{p}_j)$	$sR^1_\theta(\mathbf{p}_j)$	2
			1	0	3
			0	1	4
Affine	$\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_6]$	$\begin{pmatrix} x'_j \\ y'_j \end{pmatrix} = \begin{pmatrix} \alpha_1 & \alpha_2 \\ \alpha_4 & \alpha_5 \end{pmatrix} \begin{pmatrix} x_j \\ y_j \end{pmatrix} + \begin{pmatrix} \alpha_3 \\ \alpha_6 \end{pmatrix}$	x_j	0	1
			y_j	0	2
			1	0	3
			0	x_j	4
			0	y_j	5
			0	1	6
Projective	$\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_8]$	$x'_j = (\alpha_1 x_j + \alpha_2 y_j + \alpha_3)/w_j$ $y'_j = (\alpha_4 x_j + \alpha_5 y_j + \alpha_6)/w_j$ $w_j = \alpha_7 x_j + \alpha_8 y_j + 1$	x_j/w_j	0	1
			y_j/w_j	0	2
			$1/w_j$	0	3
			0	x_j/w_j	4
			0	y_j/w_j	5
			0	$1/w_j$	6
			$-x'_j x_j/w_j$	$-y'_j x_j/w_j$	7
			$-x'_j y_j/w_j$	$-y'_j y_j/w_j$	8

Table 1: Explicit expressions for the Jacobian matrix entries $\{J^c_{j,l}\}$, for $n = 2$ and for different types of geometric transformations. Note that $\mathbf{p}_j = [x_j, y_j]^T$, $\mathbf{p}'_j = [x'_j, y'_j]^T$, $R^1_\theta(\mathbf{p}_j) = x_j\cos(\theta) - y_j\sin(\theta)$, and $R^2_\theta(\mathbf{p}_j) = x_j\sin(\theta) + y_j\cos(\theta)$.

$\Omega \subset \mathbb{R}^n$ is the support of the RS that defines the spatial extent of the RS. In our scenario, Ω represents all points within the road network image shown in Fig. 1 (c). The residual vectors for some points within Ω are shown in Fig. 1 (d), where the residual vector for a point $\boldsymbol{\chi} \in \Omega$ is represented as an arrow originating from the point and ending at the closest point in the RS. Figure 1 (e) shows the residual vectors overlaid with transformed versions of the OS (vehicle detections) obtained by applying the estimated geometric transformation $\mathcal{T}_{\boldsymbol{\alpha}}$ (at an intermediate iteration). From the distance transform, the residual vector $\mathbf{r}(\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j))$ at a geometrically transformed vehicle detection location $\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)$ is obtained by simple *lookup* at that transformed point location. Thus the DT allows extremely efficient calculation of the objective function value and the gradient vector.

Although the DT is extremely efficient as compared to a brute force minimization in (3), the expensive per-iteration calculations of the LM; such as the point-wise Jacobian matrix (6), the approximation of the Hessian matrix (7), the summation over all the observed points encountered in the objective function (1), and the gradient (5) calculations; are still a challenge in achieving a real time performance for large scale problems, such as the WAMI geo-registration problem considered here. We address this limitation by proposing an accelerated parallelized and pipelined PChA designed for GPU platforms. Our algorithm realization is motivated by the fact that in each LM iteration, the computation of the gradient and the Hessian approximation are point-wise and independent, making PChA an ideal algorithm for parallelized implementation.

The complete parametric chamfer alignment using the LM algorithm is shown in Algorithm 1

3 GPU based implementation of the parametric chamfer alignment

In the following, we first provide a brief description of the GPU hardware and programming models using CUDA$^{\text{TM}}$, then present our proposed GPU based implementation.

3.1 GPU programming using CUDA$^{\text{TM}}$

Although initially dedicated for graphics related tasks, GPUs became a general programmable architecture after

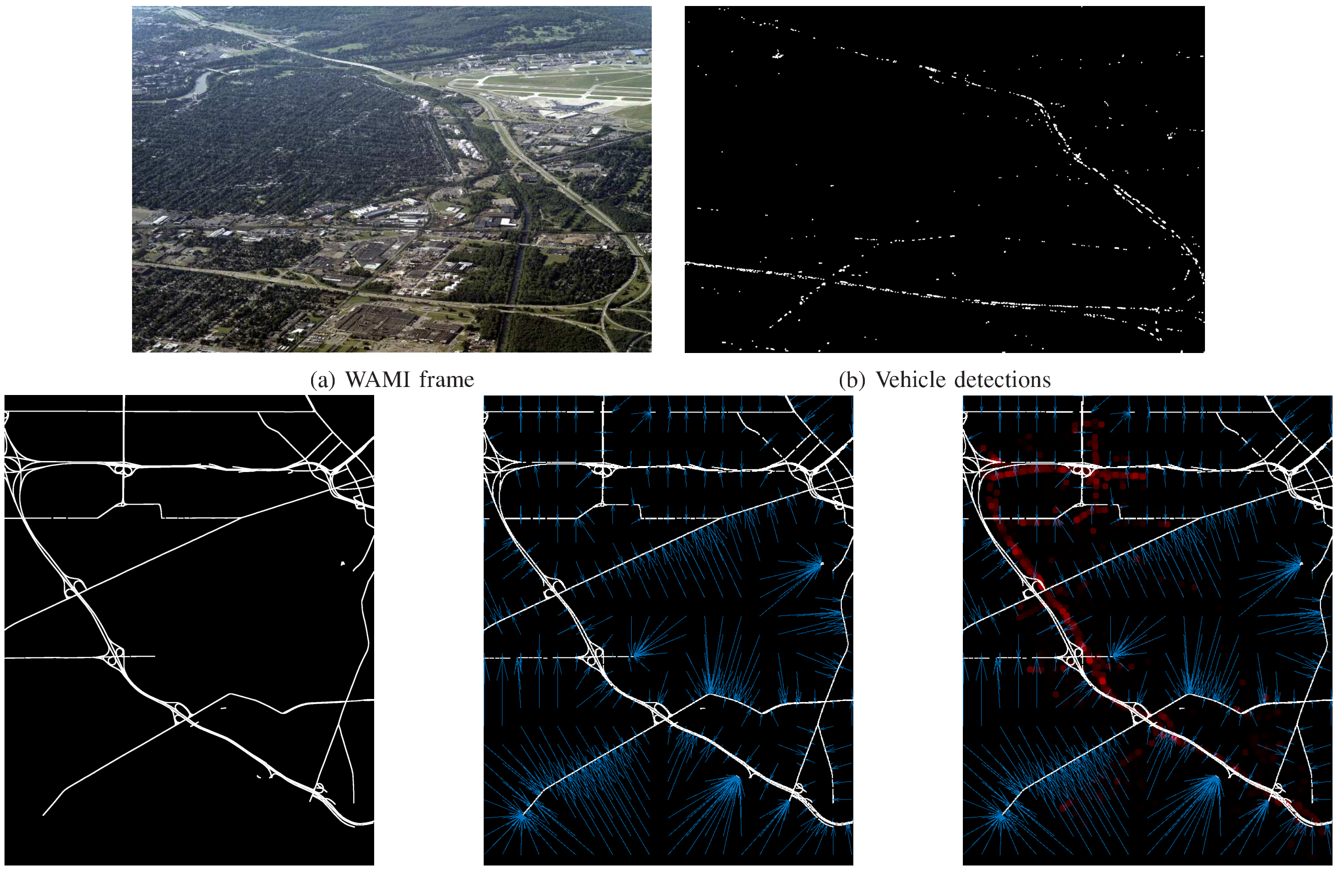


Fig. 1: WAMI frame geo-registration as a motivating example for PChA. The objective is to geo-register the WAMI frame shown in (a) by estimating the parameters for a projective transformation that aligns locations of detected vehicles shown in (b) with the geo-refrenced vector road network shown in (c). In this example, the OS comprises the vehicle detections locations and the RS comprises the locations corresponding to the network of roads shown by white points in (c). The DT is illustrated in (d) by showing computed residual vectors for selected locations in the image. Subfigure (e) shows how the DT is utilized, the geometrically transformed versions of the vehicle detection locations $\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)$, $j = 1, 2, \ldots, N_v$ obtained with a current estimate $\boldsymbol{\alpha}$ of the alignment parameters are superimposed on the DT; the residual vector $\mathbf{r}\left(\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)\right)$ at the geometrically transformed vehicle detection location $\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)$ can be readily obtained by simple lookup of the residuals stored in DT at that location. Thus the DT allows extremely efficient calculation of the objective function value and the gradient vector.

NVIDIA$^{\text{TM}}$ released CUDA$^{\text{TM}}$ in 2006, which allowed a heterogeneous programming model where both CPU and GPU can be used concurrently to run general tasks that may not necessary be graphics related [26]. From the CUDA$^{\text{TM}}$ point of view, the GPU executes tasks called kernels using thousands of parallel threads organized into blocks of grids, where *thread, block, grid* are the CUDA representations of a core, a streaming multiprocessor (multi core processor), and the GPU itself [26]. The best GPU performance is obtained when the kernel is a single instruction multiple thread (SIMT) scheme in which each thread executes the same instruction but operates on different data.

CUDA$^{\text{TM}}$ provides a hierarchy of memory with different types and characteristics [26]. A block-thread can access (1) its local memory and registers, (2) the shared memory of its block which allows the thread to communicate and co-operate efficiently with other threads in the same block, (3) the read only constant memory, and (4) the GPU global memory. The registers of the thread are the fastest level in the hierarchy, but provide very limited storage space. The constant memory is optimized for broadcast of its contents to all threads, while the global memory represents the slowest level in the hierarchy.

3.2 GPU based implementation

Our GPU based implementation of PChA takes advantage of the high-degree of parallelism provided by the

Algorithm 1: Parametric chamfer alignment using the LM algorithm

```
   Input  : Set of observed points $\mathbf{P}$, set of reference
            points $\mathbf{Q}$, and initial parameter $\boldsymbol{\alpha}_0$
   Output: Optimal geometric transformation
            parameters $\boldsymbol{\alpha}^*$
1  Compute $\mathbf{r}(\boldsymbol{\chi})$, $\forall \boldsymbol{\chi} \in \Omega$; //DT calculations
2  $t \leftarrow 0$; $update \leftarrow true$; $e_t \leftarrow f(\boldsymbol{\alpha}_t)$, $iteration \leftarrow 0$;
3  repeat /*LM iterations*/
4      if $update$ then
5          Compute $\mathbf{g}$ and $\mathbf{H}$ using (5) and (7),
             respectively with $\boldsymbol{\alpha}_t$;
6      end
7      Estimate LM update $\boldsymbol{\delta}$ for transformation
         parameter vector using (4);
8      $\boldsymbol{\alpha}_{new} \leftarrow \boldsymbol{\alpha}_t + \boldsymbol{\delta}$; /*Generate candidate solution*/
9      $e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$;
10     if ($e_{new} < e_t$) then /*Check the candidate
         solution*/
           /*Accept the candidate solution*/
11         $t \leftarrow t+1$; $\boldsymbol{\alpha}_t \leftarrow \boldsymbol{\alpha}_{new}$; $e_t \leftarrow e_{new}$;
12         $\lambda \leftarrow \lambda/10$; $update \leftarrow true$;
13     else
           /*Reject the candidate solution*/
14         $\lambda \leftarrow \lambda * 10$; $update \leftarrow false$;
15     end
16     $iteration++$;
17 until ($\|\boldsymbol{\delta}\|_2 \le \epsilon$) or ($iteration > max\ iterations$);
18 $\boldsymbol{\alpha}^* \leftarrow \boldsymbol{\alpha}_t$;
```

GPU to accelerate the computationally demanding and highly parallelizable parts of PChA, while leaving the other less computationally demanding parts for CPU execution[1]. Figure 2 illusrates the partitioning of the algorithm implementation between the CPU and the GPU. The GPU calculates the value of the objective function f defined in (1), the gradient vector $\mathbf{g}$ defined in (5), and the approximation of the Hessian matrix $\mathbf{H}$ defined in (7) that are required by each LM iteration while the CPU calculates the DT and the parameters update vector $\boldsymbol{\delta}$ using (4) with the $\mathbf{H}$ and $\mathbf{g}$ obtained from the GPU computations. As indicated in the introduction, the PChA performance is limited by the execution time of the LM algorithm, because the DT is computed only once before performing the iterative and expensive LM algorithm steps. Furthermore, for most applications, the reference dataset is available in advance, allowing for the DT to be precomputed and stored. Therefore, the focus of the proposed GPU-based implementation is to reduce the execution time indicted by "Execution time" in Fig. 2.

In the following, we first explain how we map the calculations of the objective function f, the approximation of the Hessian matrix $\mathbf{H}$, and the gradient vector $\mathbf{g}$ to the GPU by partitioning the observed points and decomposing these calculations across different GPU blocks. Then we describe our novel pipelining of the LM algorithm.

[1] Later in this paper, in Fig. 11, we provide results from profiling a single CPU implementation of PChA that empirically demonstrates that the components of PChA that we select for GPU implementation represent a substantial part of the execution time for the complete single implementation.

3.2.1 Partitioning and decomposition

To handle the computations of the objective function f, the approximation of the Hessian matrix $\mathbf{H}$, and the gradient vector $\mathbf{g}$ efficiently on the GPU, we partition the N_p points in $\mathbf{P}$ into equal sized N_b partitions, where each partition contains N_t points[2]. These partitions are distributed across the GPU blocks such that each block is responsible for computing the objective function, the approximation of the Hessian matrix, and the gradient vector for only the points assigned to that block. Within a single block, each observed point is assigned to a single block-thread for performing required calculations associated with the point.

To express this partitioning scheme mathematically, we can re-write the objective function in (1) as

$$f = \underbrace{\sum_{b=1}^{N_b}}_{\text{Grid}} \underbrace{\sum_{j \in B(b)}}_{\text{Block}} \underbrace{r_j}_{\text{Thread computes the per-point residual}} , \tag{8}$$

where $r_j = \|\mathbf{r}\left(\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)\right)\|^2$ and $B(b)$ is the set that contains the indices of the observed points that are assigned to block b. Similarly, the approximation of the Hessian matrix (7) can be written as

$$\mathbf{H} = \underbrace{\sum_{b=1}^{N_b}}_{\text{Grid}} \underbrace{\sum_{j \in B(b)}}_{\text{Block}} \underbrace{\sum_{c=1}^{n} \begin{pmatrix} \left(J^c_{j,1}\right)^2 & J^c_{j,1}J^c_{j,2} & \cdots & J^c_{j,1}J^c_{j,L} \\ J^c_{j,1}J^c_{j,2} & \left(J^c_{j,2}\right)^2 & \cdots & J^c_{j,2}J^c_{j,L} \\ \vdots & \vdots & \ddots & \vdots \\ J^c_{j,1}J^c_{j,L} & J^c_{j,2}J^c_{j,L} & \cdots & \left(J^c_{j,L}\right)^2 \end{pmatrix}}_{\text{Thread computes per-point Hessian approx. } \mathbf{H}_j} , \tag{9}$$

and the gradient vector (5) can be written as

$$\mathbf{g} = -2 \underbrace{\sum_{b=1}^{N_b}}_{\text{Grid}} \underbrace{\sum_{j \in B(b)}}_{\text{Block}} \underbrace{\mathbf{J}_j^T \mathbf{r}\left(\mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p}_j)\right)}_{\text{Thread computes the per-point gradient } \mathbf{g}_j} . \tag{10}$$

The decompositions of the computations of f, $\mathbf{H}$, and $\mathbf{g}$ as shown in (8), (9), and (10), respectively, align well with the SIMT architecture and enable these computations to be efficiently performed on the GPU. Specifically, each block-thread operates on a different input observed point and performs *independent* and *similar* calculations to compute the per-point residual value r_j,

[2] We augment the points in the OS to ensure $N_p = N_b N_t$.

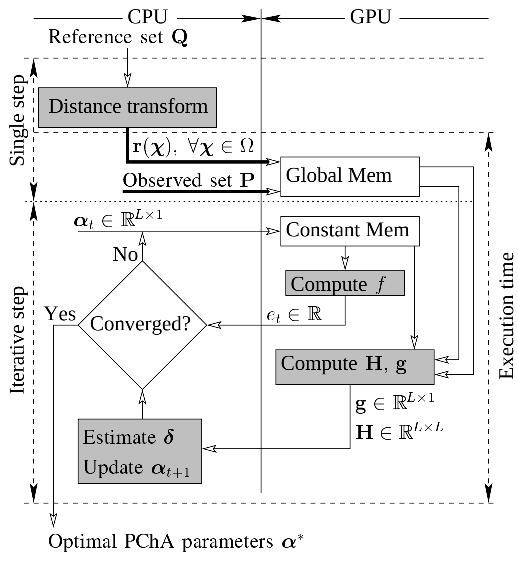

CPU
GPU
Reference set $\mathbf{Q}$
Single step
Distance transform
$\mathbf{r}(\boldsymbol{\chi}),\ \forall \boldsymbol{\chi} \in \Omega$
Observed set $\mathbf{P}$
Global Mem
$\boldsymbol{\alpha}_t \in \mathbb{R}^{L\times 1}$
Constant Mem
No
Compute f
Yes
Converged?
$e_t \in \mathbb{R}$
Iterative step
Compute $\mathbf{H}$, g
Execution time
$\mathbf{g} \in \mathbb{R}^{L\times 1}$
$\mathbf{H} \in \mathbb{R}^{L\times L}$
Estimate $\boldsymbol{\delta}$
Update $\boldsymbol{\alpha}_{t+1}$
Optimal PChA parameters $\boldsymbol{\alpha}^*$

Fig. 2: Schematic diagram that illustrates the decomposition of PChA into different computational tasks and indicates the amount of data transfer between CPU and GPU. The figure shows that there is a large amount of data transfer but only in the first single step. For the next iterative steps, the amount of data transfer is very small. Specifically, L single precision numbers copied from CPU to GPU and $L^2 + L + 1$ single precision numbers copied from GPU to CPU.

the per-point Hessian approximation $\mathbf{H}_j$, and the per-point gradient vector $\mathbf{g}_j$. Then, the calculated r_j, $\mathbf{H}_j$, and $\mathbf{g}_j$ are summed up; first within each block to obtain the per-block sums in (8), (9), and (10), then within the grid to obtain the final f, $\mathbf{H}$, and $\mathbf{g}$ in (8), (9), and (10), respectively.

The approximation of the per-point Hessian matrix $\mathbf{H}_j$ is completely determined by a smaller subset of its elements. Specifically, we exploit the frequent binary (0-1) values of the Jacobian matrix entries $\{J^c_{j,l}\}$ and the frequent similarity between them across different c and l as shown in Table 1, to reduce the computations required to determine $\mathbf{H}_j$, where for each transformation type, we define the vector $\mathbf{h}_j$ that contains the Hessian approximation matrix elements that are computed by GPU and which determine the per-point Hessian approximation $\mathbf{H}_j$. Moreover, we define the vector $\boldsymbol{\rho}_j$ that contains the Jacobian matrix elements required to compute the vector $\mathbf{h}_j$. Table 2 lists $\mathbf{h}_j$ and $\boldsymbol{\rho}_j$ for different transformation types.

Our partitioning and decomposition scheme is illustrated in Fig. 3. Each block-thread computes the scalar r_j, and the vectors $\mathbf{h}_j$ and $\mathbf{g}_j$ for a single observed point $\mathbf{p}_j$, and stores the computed elements in the shared memory of the thread block. Furthermore, a reduction summation operation [27] is carried out to obtain the sum of r_j, the sum of $\mathbf{h}_j$, and the sum of $\mathbf{g}_j$ for all observed points assigned to that block, i.e., for $j \in B(b)$ in (8), (9), and (10), where the resultant per-block sums $\sum\limits_{j\in B(b)} r_j$, $\sum\limits_{j\in B(b)} \mathbf{h}_j$, and $\sum\limits_{j\in B(b)} \mathbf{g}_j$ are stored in the global memory. To obtain the final per-grid sums $\left(\sum\limits_{b=1}^{N_b}\right)$ in (8), (9), and (10), of the previously stored per-block sums in the global memory, the GPU performs the final reduction summation operations, synchronized by one or more successive CPU reduction summation kernel launches. Because there is no synchronization mechanism across different GPU blocks, we employ the CPU to synchronize these final reduction summation operations to obtain the per-grid sums.

Figure 3 shows our proposed utilization of the different types of memory in the GPU. We efficiently utilize the different types of GPU memory according to specific memory characteristics to obtain the best performance. We use the constant memory to store the geometric transformation (matrix) $\mathcal{T}_{\boldsymbol{\alpha}}$ because of its small size and because all threads require the common value of $\mathcal{T}_{\boldsymbol{\alpha}}$. The constant memory is ideally suited for this purpose as it is highly optimized for broadcast [26]. Moreover, we store r_j, $\mathbf{h}_j$, and $\mathbf{g}_j$ that are calculated by each block-thread in the shared memory of that block for the further reduction summation operation to obtain the per-block sums. The block shared memory is accessible by all block-threads and has very low latency [26], therefore it is an ideal choice for the reduction summation operation which requires significant communication between threads in the same block to calculate the per-block sums $\sum\limits_{j\in B(b)} r_j$, $\sum\limits_{j\in B(b)} \mathbf{h}_j$, and $\sum\limits_{j\in B(b)} \mathbf{g}_j$ in (8), (9), and (10), respectively.

3.2.2 Pipelined LM

As shown in Algorithm 1, the traditional LM algorithm generates the candidate solution $\boldsymbol{\alpha}_{new}$ (Step 8), then evaluates the value of the objective function at that candidate solution (Step 9). If the candidate solution $\boldsymbol{\alpha}_{new}$ has a lower objective function value than the current solution $\boldsymbol{\alpha}_t$, the traditional LM algorithm accepts the candidate solution and updates the current solution to that candidate solution (Step 11), then the process is continued to the computation of the approximation of the Hessian matrix and the gradient vector at the updated current solution. The sequence of the traditional LM steps implies that the algorithm performs the expensive Hessian and gradient calculations only for a solution that is better than the current one.

The partitioning and decomposition scheme discussed previously handles the computations of the objective function f, the approximate of Hessian matrix $\mathbf{H}$, and the gradient vector $\mathbf{g}$ efficiently on the GPU. However, GPU based implementation of the LM algorithm with the same

Transformation type	Required Hessian and Jacobian elements
Euclidean	$\mathbf{H} = \sum_{j=1}^{N_p} \begin{pmatrix} (J^1_{j,1})^2 + (J^2_{j,1})^2 & J^1_{j,1} & J^2_{j,1} \\ J^1_{j,1} & 1 & 0 \\ J^2_{j,1} & 0 & 1 \end{pmatrix}$ $\mathbf{h}_j = \left[(J^1_{j,1})^2 + (J^2_{j,1})^2, J^1_{j,1}, J^2_{j,1}\right]$ $\boldsymbol{\rho}_j = [J^1_{j,1}, J^2_{j,1}]$
Similarity	$\mathbf{H} = \sum_{j=1}^{N_p} \begin{pmatrix} (J^1_{j,1})^2 + (J^2_{j,1})^2 & 0 & J^1_{j,1} & J^2_{j,1} \\ 0 & s^2\left((J^1_{j,1})^2 + (J^2_{j,1})^2\right) & -sJ^2_{j,1} & sJ^1_{j,1} \\ J^1_{j,1} & -sJ^2_{j,1} & 1 & 0 \\ J^2_{j,1} & sJ^1_{j,1} & 0 & 1 \end{pmatrix}$ $\mathbf{h}_j = \left[(J^1_{j,1})^2 + (J^2_{j,1})^2, J^1_{j,1}, J^2_{j,1}\right]$ $\boldsymbol{\rho}_j = [J^1_{j,1}, J^2_{j,1}]$
Affine	$\mathbf{H} = \sum_{j=1}^{N_p} \begin{pmatrix} x_j^2 & x_jy_j & x_j & 0 & 0 & 0 \\ x_jy_j & y_j^2 & y_j & 0 & 0 & 0 \\ x_j & y_j & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_j^2 & x_jy_j & x_j \\ 0 & 0 & 0 & x_jy_j & y_j^2 & y_j \\ 0 & 0 & 0 & x_j & y_j & 1 \end{pmatrix}$ $\mathbf{h}_j = [x_j^2, x_jy_j, x_j, y_j^2, y_j]$ $\boldsymbol{\rho}_j = [J^1_{j,1}, J^1_{j,2}, J^2_{j,4}, J^2_{j,5}]$
Projective	$\mathbf{H} = \sum_{j=1}^{N_p} \begin{pmatrix} {x'}^2_j & {x'}_j{y'}_j & {x'}_j & 0 & 0 & 0 & -{x'}^3_j & -{x'}^2_j{y'}_j \\ {x'}_j{y'}_j & {y'}^2_j & {y'}_j & 0 & 0 & 0 & -{x'}^2_j{y'}_j & -x'_j{y'}^2_j \\ {x'}_j & {y'}_j & 1 & 0 & 0 & 0 & -{x'}^2_j & -x'_j{y'}_j \\ 0 & 0 & 0 & {x'}^2_j & {x'}_j{y'}_j & {x'}_j & -{x'}^2_jy'_j & -{x'}_j{y'}^2_j \\ 0 & 0 & 0 & {x'}_j{y'}_j & {y'}^2_j & {y'}_j & -{x'}_j{y'}^2_j & -{y'}^3_j \\ 0 & 0 & 0 & {x'}_j & {y'}_j & 1 & -{x'}_jy'_j & -{y'}^2_j \\ -{x'}^3_j & -{x'}^2_j{y'}_j & -{x'}^2_j & -{x'}^2_jy'_j & -{x'}_j{y'}^2_j & -{x'}_jy'_j & {x'}^2_j({x'}^2_j + {y'}^2_j) & {x'}_j{y'}_j({x'}^2_j + {y'}^2_j) \\ -{x'}^2_j{y'}_j & -x'_j{y'}^2_j & -x'_j{y'}_j & -{x'}_j{y'}^2_j & -{y'}^3_j & -{y'}^2_j & {x'}_j{y'}_j({x'}^2_j + {y'}^2_j) & {y'}^2_j({x'}^2_j + {y'}^2_j) \end{pmatrix}$ $\mathbf{h}_j = [{x'}^2_j, {x'}_j{y'}_j, {x'}_j, {y'}^2_j, {y'}_j, -{x'}^3_j, -{x'}^2_j{y'}_j, -x'_j{y'}^2_j, -{y'}^3_j, {x'}^2_j({x'}^2_j + {y'}^2_j), {x'}_j{y'}_j({x'}^2_j + {y'}^2_j), {y'}^2_j({x'}^2_j + {y'}^2_j)]$ $\boldsymbol{\rho}_j = [J^1_{j,1}, J^1_{j,2}, J^1_{j,7}, J^1_{j,8}, J^2_{j,4}, J^2_{j,5}, J^2_{j,7}, J^2_{j,8}]$ (see Appendix A for details)

Table 2: The Hessian matrix approximation $\mathbf{H}_j$, the elements $\mathbf{h}_j$ from which $\mathbf{H}_j$ is computed, and associated Jacobian matrix elements $\boldsymbol{\rho}_j$ used to compute the vector $\mathbf{h}_j$. Rows of the table delineated by lines list these for commonly used Euclidean, similarity, affine, and projective 2D transformations.

sequence of steps ignores the high latency of the GPU global memory. Specifically, a block-thread fetches an observed point $\mathbf{p}_j$ and the residual vector $\mathbf{r}\left(\mathcal{T}_{\boldsymbol{\alpha}_{new}}(\mathbf{p}_j)\right)$ from the global memory in order to evaluate the objective function $f(\boldsymbol{\alpha}_{new})$ at the candidate solution $\boldsymbol{\alpha}_{new}$. The same data is fetched again for performing the gradient-Hessian calculations at the candidate solution if the LM algorithm accepts the candidate solution. Thus the expensive global memory access is performed twice for fetching the same data.

Taking into account the highly parallel architecture of the GPU that has a very low arithmetic latency compared with the high global memory access latency, we propose a pipelined LM implementation that re-orders the steps of the traditional LM. Specifically, our pipelined LM evaluates the Hessian matrix and the gradient vector at the candidate solution as it computes the objective function value associated with the candidate solution $f(\boldsymbol{\alpha}_{new})$. Our pipelined LM allows relatively inexpensive pre-computation of the Hessian-gradient calculations that are associated with the candidate solution, *without waiting* to decide whether or not the candidate solution $\boldsymbol{\alpha}_{new}$ is better than the current one $\boldsymbol{\alpha}_t$. Because of the high global memory access latency of the GPU compared with the arithmetic latency, once the required data is fetched from the GPU global memory, the Hessian-gradient calculations are preferably performed when evaluating $f(\boldsymbol{\alpha}_{new})$. In other words, the time re-

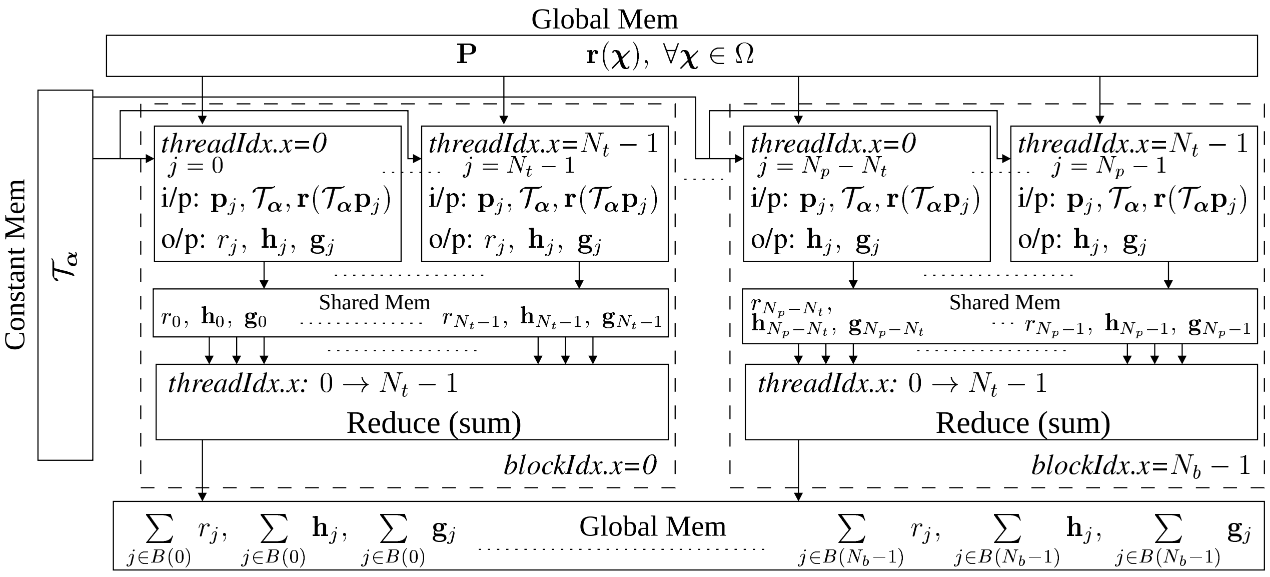

Global Mem
$\mathbf{P}$
$\mathbf{r}(\boldsymbol{\chi}),\ \forall\boldsymbol{\chi}\in\Omega$
Constant Mem
$\mathcal{T}_{\boldsymbol{\alpha}}$
threadIdx.x=0
$j=0$
i/p: $\mathbf{p}_j, \mathcal{T}_{\boldsymbol{\alpha}}, \mathbf{r}(\mathcal{T}_{\boldsymbol{\alpha}}\mathbf{p}_j)$
o/p: $r_j,\ \mathbf{h}_j,\ \mathbf{g}_j$
threadIdx.x=N_t-1
$j=N_t-1$
threadIdx.x=0
$j=N_p-N_t$
o/p: $\mathbf{h}_j,\ \mathbf{g}_j$
threadIdx.x=N_t-1
$j=N_p-1$
Shared Mem
$r_0,\ \mathbf{h}_0,\ \mathbf{g}_0$
$r_{N_t-1},\ \mathbf{h}_{N_t-1},\ \mathbf{g}_{N_t-1}$
$r_{N_p-N_t},\ \mathbf{h}_{N_p-N_t},\ \mathbf{g}_{N_p-N_t}$
$r_{N_p-1},\ \mathbf{h}_{N_p-1},\ \mathbf{g}_{N_p-1}$
threadIdx.x: $0\rightarrow N_t-1$
Reduce (sum)
blockIdx.x=0
blockIdx.x=N_b-1
$\sum_{j\in B(0)} r_j,\ \sum_{j\in B(0)}\mathbf{h}_j,\ \sum_{j\in B(0)}\mathbf{g}_j$
$\sum_{j\in B(N_b-1)} r_j,\ \sum_{j\in B(N_b-1)}\mathbf{h}_j,\ \sum_{j\in B(N_b-1)}\mathbf{g}_j$

Fig. 3: Schematic diagram illustrating the partitioning of the observed points across GPU blocks and the decomposition of the objective function, the approximation of the Hessian matrix, and the gradient vector calculations according to the CUDA$^{\text{TM}}$ programming model. Each block-thread computes r_j, $\mathbf{h}_j$, and $\mathbf{g}_j$ for a single observed point $\mathbf{p}_j$ and stores the output in the shared memory of the block. Then a reduction summation operation is carried out to obtain the sum of r_j, the sum of $\mathbf{h}_j$, and the sum of $\mathbf{g}_j$ for all observed points within that block, where the resultant per-block sums are stored in the global memory for further per-grid summations.

quired for Hessian matrix approximation and the gradient vector calculations is significantly reduced if these evaluations are combined with the objective function evaluation, because the data needed to compute them is already fetched from the global memory in order to evaluate the objective function value associated with the candidate solution. Figure 4 illustrates the timed execution for a trace of few steps for both the traditional and our pipelined LM implementations on the GPU, where the Hessian-gradient calculations, and the objective function evaluation are performed separately by two GPU kernels in the traditional LM implementation and combined into a single kernel in the pipelined LM implementation.

Our GPU kernel that combines the Hessian-gradient calculations with the objective function evaluation is shown in Algorithm 2 and the overall pipelined LM is shown[3] in Algorithm 3. Our pipelining of the LM calculates the Hessian matrix approximation and gradient vector at the candidate solution $\boldsymbol{\alpha}_{new}$ concurrently with the calculations of the objective function at $\boldsymbol{\alpha}_{new}$. If $\boldsymbol{\alpha}_{new}$ proves to be better than the current solution $\boldsymbol{\alpha}_t$, the pipelined LM immediately uses the pre-computed Hessian approximation and gradient vector to compute the parameter update vector $\boldsymbol{\delta}$ and generates a new candidate solution. Otherwise, the pipelined LM disregards the pre-computed gradient and Hessian approximation, losing the small amount of time that was dedicated for the Hessian-gradient pre-calculations. Thus the pipelined LM provides a faster execution time than the traditional LM on the GPU because the pipelined LM performs the expensive global memory access only once for the combined objective function evaluation and the Hessian-gradient

Algorithm 2: GradientHessianResidualReduce kernel

```
1  $j \leftarrow threadIdx.x + blockIdx.x \times blockDim.x$;
2  Fetch $\mathbf{p}_j$ from global memory and $\mathcal{T}_{\boldsymbol{\alpha}}$ from constant memory;
3  Compute $\mathbf{p}'_j \leftarrow \mathcal{T}_{\boldsymbol{\alpha}}\mathbf{p}_j$;
4  if $\mathbf{p}'_j \in \Omega$ then
5      Fetch $\mathbf{r}(\mathbf{p}'_j)$ from global memory;
       Residual calculations:
6      Compute $r_j = \|\mathbf{r}\left(\mathbf{p}'_j\right)\|^2$;
       Gradient and Hessian calculations:
7      Compute $\boldsymbol{\rho}_j$ for $\mathbf{p}_j$ using the formulae from Table 1;
8      Compute $\mathbf{h}_j$ using $\boldsymbol{\rho}_j$;
9      Compute $\mathbf{g}_j$ from (10) using $\boldsymbol{\rho}_j$ and $\mathbf{r}(\mathbf{p}'_j)$ ;
10     Store $\mathbf{h}_j$, $\mathbf{g}_j$, and $r_j$ in the shared memory;
11 end
12 __syncthreads();
   Reduce (sum):
13 Reduce : compute the per-block sums $\sum_{j\in B(blockIdx.x)}\mathbf{h}_j$, $\sum_{j\in B(blockIdx.x)}\mathbf{g}_j$, and $\sum_{j\in B(blockIdx.x)} r_j$;
14 Store the per-block sums in the global memory;
```

[3] For clarity, we omit the final per-grid reduction summation operations from Algorithm 3 and assume that the kernel in Algorithm 2 will return $\{\mathbf{g},\ \mathbf{H},\ f\}$.

calculations while the traditional LM performs the same expensive global memory access twice.

Algorithm 3: Parametric chamfer alignment using our proposed pipelined LM algorithm

```
   Input  : Set of observed points $\mathbf{P}$, set of reference
            points $\mathbf{Q}$, and initial parameter $\boldsymbol{\alpha}_0$
   Output: Geometric transformation parameter $\boldsymbol{\alpha}^*$
1  Compute $\mathbf{r}(\boldsymbol{\chi})$, $\forall\boldsymbol{\chi} \in \Omega$; //DT calculations
2  $t \leftarrow 0$; $iteration \leftarrow 0$;
3  $\{\mathbf{g},\ \mathbf{H},\ e_t\} \leftarrow$ GradientHessianResidual
     $<<< N_b, N_t >>>$ $(\boldsymbol{\alpha}_t)$;
4  repeat /*LM iterations*/
5      Estimate LM update $\boldsymbol{\delta}$ for transformation
         parameter vector using (4);
6      $\boldsymbol{\alpha}_{new} \leftarrow \boldsymbol{\alpha}_t + \boldsymbol{\delta}$; /*Generate candidate solution*/
7      $\{\mathbf{g}_{new},\ \mathbf{H}_{new},\ e_{new}\} \leftarrow$ GradientHessianResidual
         $<<< N_b, N_t >>>$ $(\boldsymbol{\alpha}_{new})$;
8      if $(e_{new} < e_t)$ then /*Check the candidate
         solution*/
           /*Accept the candidate solution*/
9          $t \leftarrow t+1$; $\boldsymbol{\alpha}_t \leftarrow \boldsymbol{\alpha}_{new}$; $e_t \leftarrow e_{new}$;
10         $\mathbf{g} \leftarrow \mathbf{g}_{new}$; $\mathbf{H} \leftarrow \mathbf{H}_{new}$;
11         $\lambda \leftarrow \lambda/10$;
12     else
           /*Reject the candidate solution*/
13         $\lambda \leftarrow \lambda * 10$;
14     end
15     $iteration++$;
16 until $(\|\boldsymbol{\delta}\|_2 \leq \epsilon)$ or $(iteration > max\ iterations)$;
17 $\boldsymbol{\alpha}^* \leftarrow \boldsymbol{\alpha}_t$;
```

4 GPU/CPU Implementations and Configurations for Benchmarks

To illustrate the benefit of the proposed GPU realization and to highlight how the individual elements of our proposed scheme contribute to the overall performance gains, we consider several alternative GPU and CPU implementations for our benchmarks. The GPU based implementations are:

– **PGPU**: The complete proposed GPU based implementation of PChA that uses the parallelization and the pipelined LM.
– **TGPU**: A GPU based implementation of PChA that uses the proposed parallelization but with the traditional, as opposed to the pipelined, LM. A comparison against this specifically allows us to evaluate the benefit of pipelining.
– **GGPU**: an (inefficient) GPU based implementation of PChA intended to demonsrate the benefit of the partitioning used in PGPU across the GPU memory hierarchy. Specifically, GGPU only uses the GPU global memory and does not use the shared memory (but does use the pipelining for LM). GGPU uses the global memory to (a) store the geometric transformation $\mathcal{T}_{\boldsymbol{\alpha}}$ (for which PGPU uses the constant memory) and (b) to store r_j, $\mathbf{h}_j$, and $\mathbf{g}_j$ that are calculated by each block-thread (for which PGPU uses the shared memory).

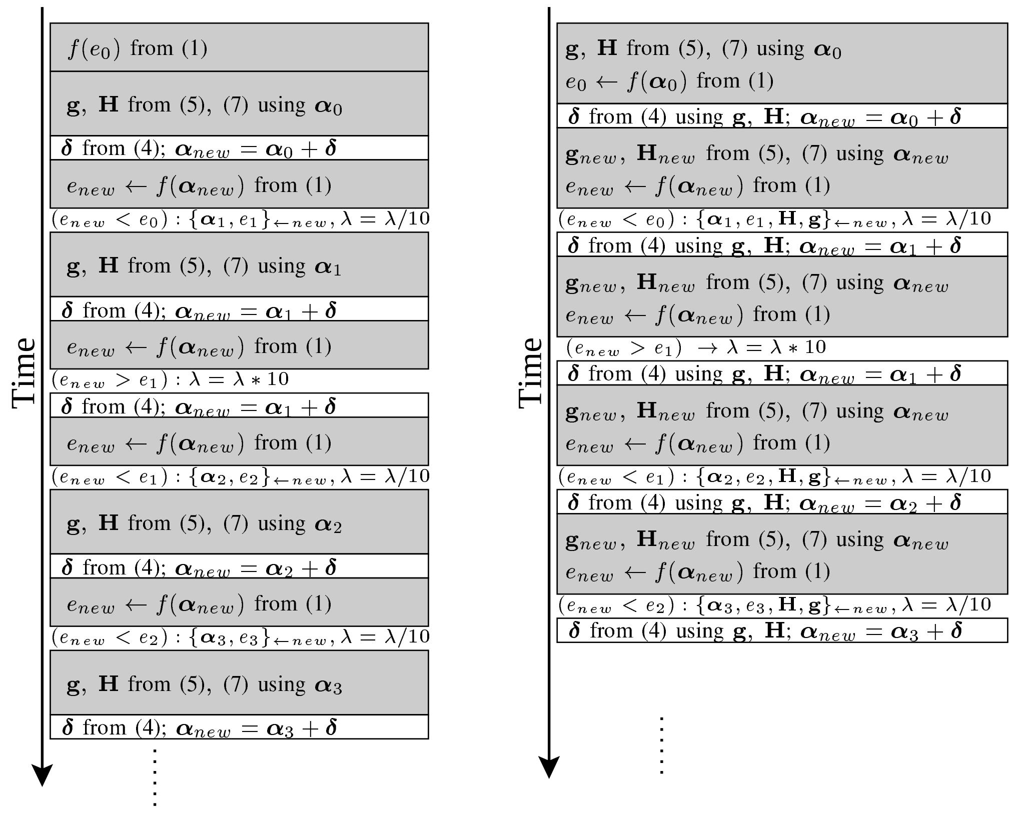
In addition to the GPU based implementations, for benchmarking, we also provide a parallelized CPU acceleration for the PChA. Specifically, we use OpenMP [28] to parallelize the expensive per-point computations of the objective function, gradient and Hessian approximation, which are required for the LM iterations in PChA. That is, the operations shown on the right half in Fig. 2 under the GPU heading are instead parallelized on the CPU using OpenMP. Algorithms 4 and 5 summarize, in pseudo-code with included OpenMP compiler directive constructs, the OpenMP based acceleration for the calculation of the objective function and of the gradient and Hessian approximation, respectively. We denote the OpenMP based CPU implementation as **CPU(n)**, where n indicates the number of threads used when executing the implementation. For example, **CPU(8)** denotes the OpenMP based CPU implementation that uses 8 threads. Also, **CPU(1)** denotes a CPU implementation that uses only single thread[4], which is obtained by using the OpenMP command "omp_set_num_threads(1)". All the GPU and CPU implementations utilize the reduced Hessian and Jacobian computations provided in Table 2.

Algorithm 4: OpenMP based acceleration for the computation of the objective function f

```
1  $e \leftarrow 0$;
   #pragma omp for reduction (+:$e$)
2  for $j$ = 1 to $N_p$ do
3      Compute $\mathbf{p}'_j \leftarrow \mathcal{T}_{\boldsymbol{\alpha}}\mathbf{p}_j$;
4      if $\mathbf{p}'_j \in \Omega$ then
5          $e$ += $\|\mathbf{r}\left(\mathbf{p}'_j\right)\|^2$;
6      end
7  end
8  return $e$;
```

All implementations were in C++ with OpenMP 3.0 and CUDA$^{\text{TM}}$ 7.0. As indicated by "Execution time" annotation in Fig. 2, reported execution times for an implementation represent the time from completion of the DT computation to the convergence of the LM. Thus, the execution time is measured for the complete PChA algorithm excluding the computation time for the DT. Specifically, this implies that, any reported execution time for a GPU based implementation **includes** the time for all data transfer between CPU memory and GPU memory. Also, the speed-up factor of an implementation

[4] The **CPU(1)** implementation offers performance close to but not identical to that of a CPU implementation that is obtained by completely eliminating the OpenMP compiler directives from the code.


Time
$f(e_0)$ from (1)
$\mathbf{g}$, $\mathbf{H}$ from (5), (7) using $\boldsymbol{\alpha}_0$
$\boldsymbol{\delta}$ from (4); $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_0 + \boldsymbol{\delta}$
$e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$ from (1)
$(e_{new} < e_0) : \{\boldsymbol{\alpha}_1, e_1\}_{\leftarrow new}, \lambda = \lambda/10$
$\mathbf{g}$, $\mathbf{H}$ from (5), (7) using $\boldsymbol{\alpha}_1$
$\boldsymbol{\delta}$ from (4); $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_1 + \boldsymbol{\delta}$
$e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$ from (1)
$(e_{new} > e_1) : \lambda = \lambda * 10$
$\boldsymbol{\delta}$ from (4); $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_1 + \boldsymbol{\delta}$
$e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$ from (1)
$(e_{new} < e_1) : \{\boldsymbol{\alpha}_2, e_2\}_{\leftarrow new}, \lambda = \lambda/10$
$\mathbf{g}$, $\mathbf{H}$ from (5), (7) using $\boldsymbol{\alpha}_2$
$\boldsymbol{\delta}$ from (4); $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_2 + \boldsymbol{\delta}$
$e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$ from (1)
$(e_{new} < e_2) : \{\boldsymbol{\alpha}_3, e_3\}_{\leftarrow new}, \lambda = \lambda/10$
$\mathbf{g}$, $\mathbf{H}$ from (5), (7) using $\boldsymbol{\alpha}_3$
$\boldsymbol{\delta}$ from (4); $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_3 + \boldsymbol{\delta}$
Time
$\mathbf{g}$, $\mathbf{H}$ from (5), (7) using $\boldsymbol{\alpha}_0$
$e_0 \leftarrow f(\boldsymbol{\alpha}_0)$ from (1)
$\boldsymbol{\delta}$ from (4) using $\mathbf{g}$, $\mathbf{H}$; $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_0 + \boldsymbol{\delta}$
$\mathbf{g}_{new}$, $\mathbf{H}_{new}$ from (5), (7) using $\boldsymbol{\alpha}_{new}$
$e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$ from (1)
$(e_{new} < e_0) : \{\boldsymbol{\alpha}_1, e_1, \mathbf{H}, \mathbf{g}\}_{\leftarrow new}, \lambda = \lambda/10$
$\boldsymbol{\delta}$ from (4) using $\mathbf{g}$, $\mathbf{H}$; $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_1 + \boldsymbol{\delta}$
$\mathbf{g}_{new}$, $\mathbf{H}_{new}$ from (5), (7) using $\boldsymbol{\alpha}_{new}$
$e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$ from (1)
$(e_{new} > e_1) \rightarrow \lambda = \lambda * 10$
$\boldsymbol{\delta}$ from (4) using $\mathbf{g}$, $\mathbf{H}$; $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_1 + \boldsymbol{\delta}$
$\mathbf{g}_{new}$, $\mathbf{H}_{new}$ from (5), (7) using $\boldsymbol{\alpha}_{new}$
$e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$ from (1)
$(e_{new} < e_1) : \{\boldsymbol{\alpha}_2, e_2, \mathbf{H}, \mathbf{g}\}_{\leftarrow new}, \lambda = \lambda/10$
$\boldsymbol{\delta}$ from (4) using $\mathbf{g}$, $\mathbf{H}$; $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_2 + \boldsymbol{\delta}$
$\mathbf{g}_{new}$, $\mathbf{H}_{new}$ from (5), (7) using $\boldsymbol{\alpha}_{new}$
$e_{new} \leftarrow f(\boldsymbol{\alpha}_{new})$ from (1)
$(e_{new} < e_2) : \{\boldsymbol{\alpha}_3, e_3, \mathbf{H}, \mathbf{g}\}_{\leftarrow new}, \lambda = \lambda/10$
$\boldsymbol{\delta}$ from (4) using $\mathbf{g}$, $\mathbf{H}$; $\boldsymbol{\alpha}_{new} = \boldsymbol{\alpha}_3 + \boldsymbol{\delta}$

(a) Traditional LM (b) Our pipelined LM

Fig. 4: Execution timeline that shows a trace for few steps of GPU based implementations of: (a) the traditional LM algorithm and (b) the proposed pipelined LM algorithm. The traditional LM algorithm computes the gradient and Hessian only if the candidate solution $\boldsymbol{\alpha}_{new}$ is accepted and **after** the acceptance. On the other hand, our pipelined LM algorithm preemptively computes the Hessian matrix approximation and the gradient vector for the candidate solution $\boldsymbol{\alpha}_{new}$ concurrently with the evaluation of the objective function value at the candidate solution $\boldsymbol{\alpha}_{new}$. Because the data needed for the Hessian-gradient calculations is already fetched from the high latency global memory for the evaluation of the objective function, the gradient-Hessian calculations take little additional time and provide very significant savings when the candidate solution is accepted and only a small overhead of wasted time when the candidate solution is rejected.

a over an implementation b is computed as the ratio of the execution time for b to the execution time for a.

The different GPU and CPU based implementations were benchmarked on two different computer systems. The first system, which we refer to as “MC1”, is a desktop computer with the Ubuntu 14.04.4 LTS operating system. MC1 had 32 GB of main memory installed and an Intel® Core™ i7-4790 CPU with 8 cores, 8 MB cache, operating at 3.60 GHz. MC1 was equipped with an NVIDIA™ GeForce GTX™ 760 graphics card that comprises a GPU based on the Kepler™ GK104 core architecture with 2 GB of physical memory and 1152 CUDA cores operating at 1033 MHz. The second system, which we refer to as “MC2”, is a compute node in a linux compute cluster [29] with the Red Hat Enterprise Linux Server release 6.6 (Santiago) operating system. MC2 is configured to have 16 GB of main memory and an Intel® Xeon® E5-2695 v2 CPU with 12 cores, 30 MB cache, operating at 2.40 GHz. MC2 is equipped with an NVIDIA™ Tesla™ K20X graphics processing accelerator that comprises a GPU based on the Kepler™ GK110 core architecture with 6 GB of physical memory and 2688 CUDA cores operating at 732 MHz. For the GPU based implementations, the number of threads in each block for both MC1 and MC2 was set to 128. This number was obtained empirically by varying the number of threads in each block and determining the value that resulted in the smallest execution time.

5 Application Examples

In this section, we demonstrate the efficiency of our PChA GPU based implementation on 2D/3D large scale point

Algorithm 5: OpenMP based acceleration for the computation of the gradient $\mathbf{g}$ and the Hessian approximation $\mathbf{H}$

```
1  $\mathbf{H} \leftarrow \mathbf{0}$; $\mathbf{g} \leftarrow \mathbf{0}$;
   #pragma omp for reduction (+:$\mathbf{H}, \mathbf{g}$)
2  for $j$ = 1 to $N_p$ do
3      Compute $\mathbf{p}'_j \leftarrow \mathcal{T}_{\alpha} \mathbf{p}_j$;
4      if $\mathbf{p}'_j \in \Omega$ then
5          Compute $\boldsymbol{\rho}_j$ for $\mathbf{p}_j$ using the formulae from
             Table 1;
6          Compute $\mathbf{H}_j$ using $\boldsymbol{\rho}_j$;
7          Compute $\mathbf{g}_j$ from (10) using $\boldsymbol{\rho}_j$ and $\mathbf{r}(\mathbf{p}'_j)$ ;
8          $\mathbf{H}$ += $\mathbf{H}_j$;
9          $\mathbf{g}$ += $\mathbf{g}_j$;
10     end
11 end
12 return $\mathbf{H}, \mathbf{g}$;
```

set alignment problems that are related to our ongoing research on WAMI geo-registration [8, 9] and analyses of homologous buildings using range imaging [7, 24].

5.1 WAMI real time geo-registration

In our previous work [8, 9], as shown in the motivating example in Section 2, we employed the PChA to geo-register a WAMI frame by finding the unknown projective geometric transformation that aligns the detected vehicle locations contained in that WAMI frame with a geo-referenced vector road map (using only a CPU based implementation that did not use any of the optimizations reported here, including those in Table 2.). We estimate the unknown projective geometric transformation using PChA and we evaluate the geo-registration execution time on MC1 for the PGPU, CPU(1), CPU(4), and CPU(8) implementations on a WAMI test set. Our WAMI test set is composed of 60 frames chosen randomly from the CorvusEye WAMI data set that is captured by the CorvusEye system [30] at 2 frames per second. In Fig. 5, we plot the execution time (in ms) for the PGPU, CPU(1), CPU(4), and CPU(8) implementations applied on each frame from the WAMI test set, where a logarithmic scale is used for the execution time axis. Our PGPU implementation achieves real time performance on the WAMI test set and provides an average speed-up of 6.9× compared with the single CPU implementation CPU(1), where the average is computed over the 60 frames. Additionally, for some frames, even the CPU(8) implementation cannot estimate the projective geometric transformation in real time, while our PGPU implementation achieve real time performance for all frames.

To assess how the performance of the different PChA implementations varies with differing vehicle counts in a WAMI frame, we use our actual datasets to synthetically generate additional vehicle detections locations. Specifically, we first use our accurate geo-registration scheme proposed in [9] to align the WAMI frame with a vector road network, then we add a varying number of vehicle detections randomly at the locations coinciding with the road network in the aligned WAMI frame. Then, we apply a different number of Euclidean, similarity, affine, and projective parametric transformations on the obtained vehicle detections to generate the OS where we randomly generate the transformation parameters associated with each transformation type within tolerances typically seen for the meta data based geo-registration in our actual datasets. In our semi-synthetic WAMI experiment, we first generate 100 random synthetic geometric transformations for each transformation type (Euclidean, similarity, affine, and projective). Then, we use the GPU based implementations PGPU, TGPU, GGPU, in addition to the single CPU implementation (CPU(1)), to estimate the geometric transformation parameters and we measure the execution time for each case on both MC1 and MC2. For each transformation type, we calculate the average and the standard deviation of the execution time for the 100 random transformations generated for that transformation type (within established tolerance limits). Figures 6 (a), (b), (c), (d) plot the calculated average execution time (in ms) against the number of points N_p in the OS for the Euclidean, similarity, affine, and projective transformation types, respectively.

From Fig. 6, we can conclude that our PGPU implementation shows significantly better performance (smaller execution time) compared with the single CPU implementation for both MC1 and MC2. The improvement comes from the efficient partitioning of the observed points and the decomposition of the LM calculations across the different GPU blocks to exploit the powerful highly parallel architecture of the GPU. Among the GPU based implementations, the PGPU implementation shows the best performance highlighting the benefits of the pipelining and of the effective usage of the GPU. Specifically, compared with the TGPU implementation, the PGPU implementation saves a constant amount of time per-iteration, when the new candidate solution is accepted. The saved amount is the time required by a block-thread to access the global memory for fetching the data for performing the gradient-Hessian calculations as discussed in Section 3.2.2. Because the time saving is constant per iteration, the speed-up of PGPU over TGPU is always a constant factor, as can be seen the plots in Fig. 6. The GGPU implementation has the worst performance among all GPU based implementations because it utilizes the high latency global memory for all computations. This highlights the importance of efficiently utilizing the different memory types of the GPU.

Under the same settings of the semi-synthetic WAMI experiment, the plots in Fig. 7 show the average execution time of the GPU and CPU based implementations for a dataset with 1 million observed points for different transformation types. We use a logarithmic scale for the

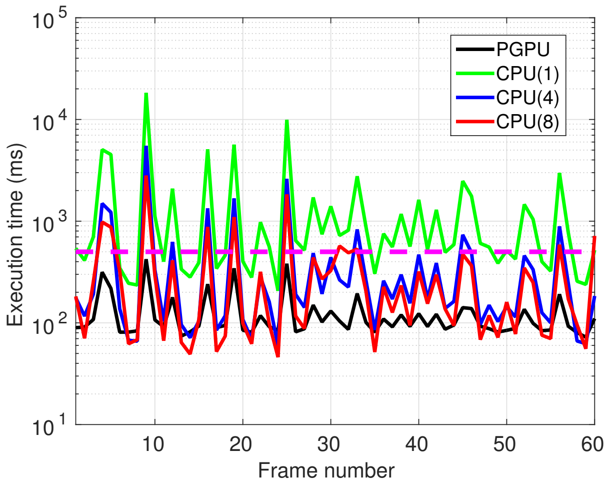


Fig. 5: A plot shows the execution time (in ms) for the PGPU, CPU(1), CPU(4), and CPU(8) implementations applied on 60 WAMI frames and tested on MC1, where a logarithmic scale is used for the execution time axis. Our PGPU implementation achieves the real time performance (registration within the WAMI inter-frame duration of 500 ms).

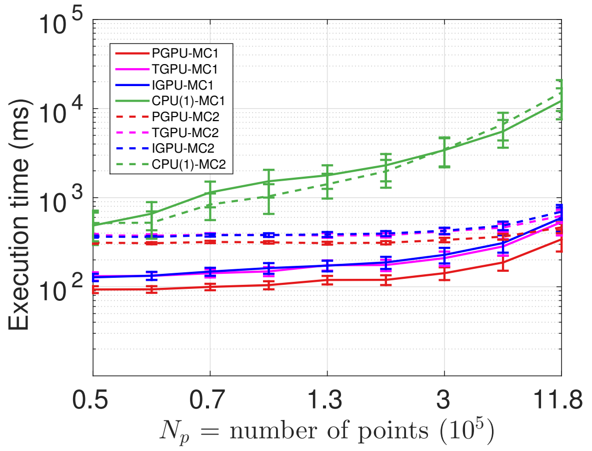
execution time axis and show the execution time (in ms) on MC1 and MC2 in Fig. 7 (a) and (b), respectively. Among the implementations benchmarked, the PGPU implementation has the best performance across the different transformation types and for both MC1 and MC2. For example, on MC1, the PGPU achieves 20× - 26× speed-up over CPU(1) and also achieves a speed-up of between 1.3× - 1.5× over the TGPU implementation due to the pipelining of the LM. The CPU parallel implementation shows better performance than the single CPU implementation and the performance of the CPU parallel implementation is enhanced by increasing the number of threads used. However, using threads more than the number of cores of the CPU is not useful for the parallel CPU implementation. For example, on MC1, the performance of CPU(8) is better than CPU(12), while on MC2, the performance of CPU(12) is better. Because the CPU of MC1 has 8 physical cores while the CPU of MC2 has 12 physical cores, this suggests that the best performance of the parallel CPU implementation is obtained when the number of used threads matches the number of cores of the CPU, and adding more threads is not useful. In contrast, the GPU based implementations show better performance than the CPU parallel implementation because of the large number of threads that can run concurrently on the GPU due to its large number of available computational cores compared with the CPU. Also, the performance of GPU implementations on MC2 is slightly better than the performance on MC1 because the GPU of MC2 has larger number of cores than the GPU of MC1.

To assess how the performance of the different PChA implementations varies according to the number of iterations of the LM, we forced the LM to perform a specific number of iterations by setting the condition ($e_{new} < e_t$) to be true (Line 10 in Algorithm 1 and Line 8 in Algorithm 3) until the desired iteration count was reached. The average and standard deviation of the execution time was evaluated for varying numbers of data points N_p and iterations for each transformation type (Euclidean, similarity, affine, and projective) over 100 randomly generated geometric transformations. This experiment was conducted on MC1 for the PGPU, CPU(1), and CPU(8) implementations. Fig. 8 (a), (b), (c), (d) show plots of the average execution time against N_p for 10, 50, 100, and 200 iterations, respectively. The execution time includes the time required for the large amount of initial data transfer to the GPU global memory (shown in the first single step in Fig. 2) and this component is represented in all plots in Fig. 8 by a thick black line. PGPU has a smaller execution time than the CPU(1) implementation for all of the plots in Fig. 8. Additionally, for a sufficiently large number of iterations or data points, the PGPU implementation has a better performance than the CPU(8) implementation. When the number of data points or iterations is small, the execution time for the PGPU implementation is dominated by the overhead associated with the time required for the initial data transfer and CPU(8) implementation has a lower execution time in these settings.

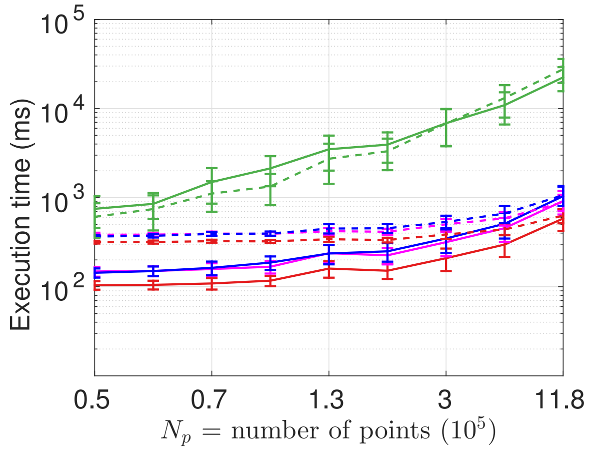
5.2 Analysis of homologous architectural buildings

In our ongoing analyses of homologous architectural buildings [7, 24], we are interested in identifying differences between homologous historical buildings constructed to a common template. We are given a reference 3D model created from engineering drawings and corresponding observed 3D range data captured for several historical buildings using a lidar scanner, and our goal is to identify the differences between the observed buildings' range data and the reference model. Toward this goal, we employ PChA to align the 3D range data with the 3D reference model. Figure 9 shows an example, with the 3D reference model in (a), the observed range data in (b), and the alignment result with the observed range data overlaid with the reference model in (c). The alignment is obtained by finding the unknown 3D similarity transformation that aligns the observed building 3D range data with the 3D reference model. In this alignment example, we use the PGPU and the CPU(1) implementations to align the observed 3D range data that contains 1140703 points and report results obtained on MC1. The alignment execution time of the PGPU and the CPU(1) implementations are 790 ms and 11850 ms, respectively.

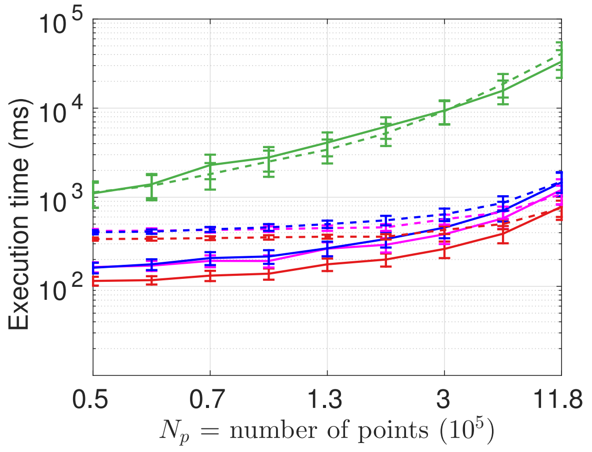
As in Section 5.1 for the 2D case, we assessed the impact of the number N_p of observed 3D range data



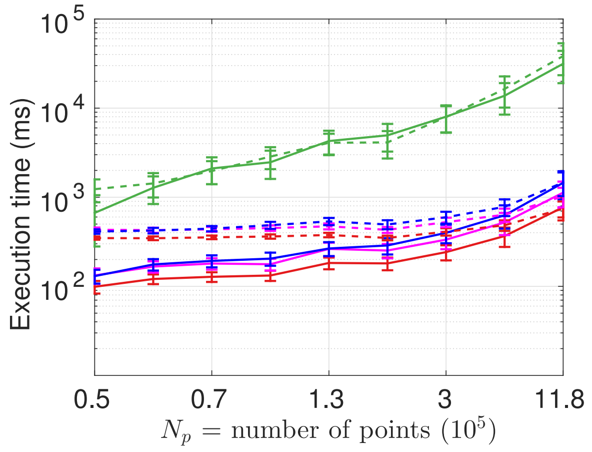
(a) Euclidean



(b) Similarity



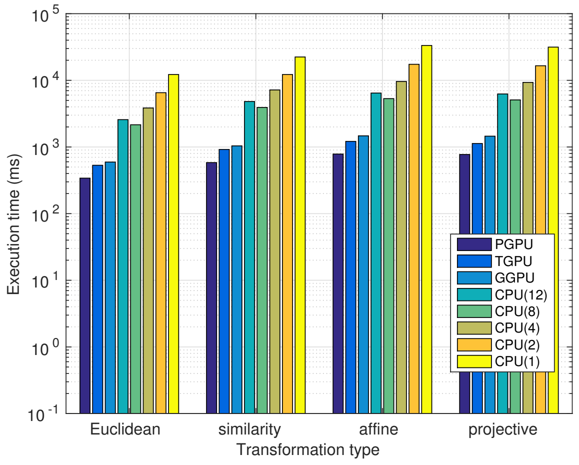
(c) Affine



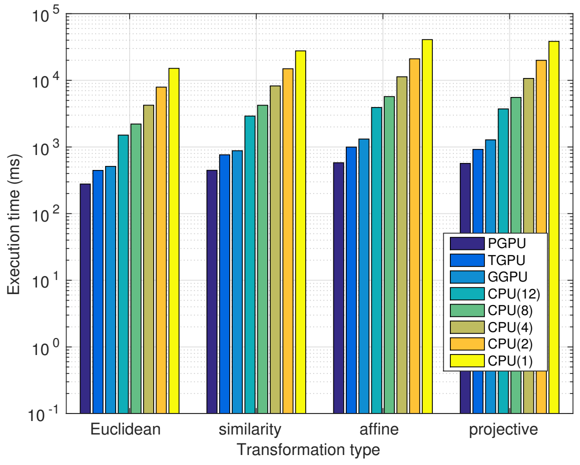
(d) Projective

Fig. 6: Average execution time for aligning the 2D semi-synthetic WAMI datasets with a corresponding roadmap as a function of the number of observed points for (a) Euclidean, (b) similarity, (c) affine, and (d) projective transformation types. In this experiment motivated by our WAMI to roadmap alignment problem, we use observed datasets (OS) with a varying number of vehicle detections generated by applying a synthetic random transformation to synthetic detections generated on the roads. An alignment transformation is then estimated by PChA using the GPU based implementations PGPU, TGPU, GGPU, in addition to the single CPU implementation (CPU(1)) on both MC1 and MC2. Reported values correspond to averages of the execution time over 100 runs corresponding to 100 different synthetic transformations. Standard deviations are shown as error bars and the execution time for an implementation on MC1 and MC2 is shown in the same color but with different line styles (solid lines for MC1 and dashed lines for MC2).

points on the different implementations by using synthetically generated OS of different sizes. For each of the Euclidean, similarity, and affine parametric transformation types, we randomly generated 100 different values for the associated transformation parameters (within pre-determined tolerance) and obtain corresponding synthetic OS of different sizes. For each of the generated OS datasets, the PGPU, the TGPU, and the CPU(1) implementations were used to estimate the transform parameters and the average execution time and standard deviation were computed for each value of N_p over the 100 realizations. The experiment was independently conducted on both MC1 and MC2. Fig. 10 (a), (b) and (c) show plots of the average execution time against N_p for the Euclidean, similarity, and affine parametric transformations, respectively, on the observed range data to generate the OS where we randomly generate the transformation parameters (within pre-determined tolerance) associated with each transformation type. Again, the PGPU implementation shows significantly better performance (smaller execution time) compared with the single CPU implementation. As in Section 5.1, the PGPU also has a nearly constant computational speed-up factor over the TGPU implementation.



(a) MC1



(b) MC2

Fig. 7: Average execution time of the GPU and CPU based implementations for a dataset with 1 million observed points and estimated for different transformation types. We use a logarithmic scale for the execution time axis and show the execution time (in ms) on MC1 and MC2 in (a) and (b), respectively.

6 Discussion

The two main ingredients contributing to the observed speed up for the proposed implementation in this paper are the parallelization and the pipelining. In this section, we characterize the improvements provided by each ingredient, using empirical profiling for the first and simplified analysis for the second.

To assess the relative contribution of the parallelized components to the overall computations required for the PChA, we empirically profiled the average execution time for each component of the PChA. As in Section 5.1, a semi-synthetic dataset was used for this purpose with $N_p = 1$ million data points and 1000 realizations for each geometric transformation type (Euclidean, similarity, affine, and projective) obtained by using synthetically generated random parameters. For each transformation type, average execution times were computed for each component of the PChA using the single CPU implementation CPU(1) on MC2. The results summarized in the bar graphs in Fig. 11 show that the execution times for the computation of f, $\mathbf{H}$, and $\mathbf{g}$, which are parallelized in the GPU implementation presented in this paper are much larger than the execution time for the computation of $\boldsymbol{\delta}$, which is not parallelized here. The parallelization in the implementation presented therefore effectively targets the computationally most demanding components of PChA. For reference, the time required for the computation of the DT[5] is also included in Fig. 11, although, as we already noted, most applications do not require the DT to be computed in real-time.

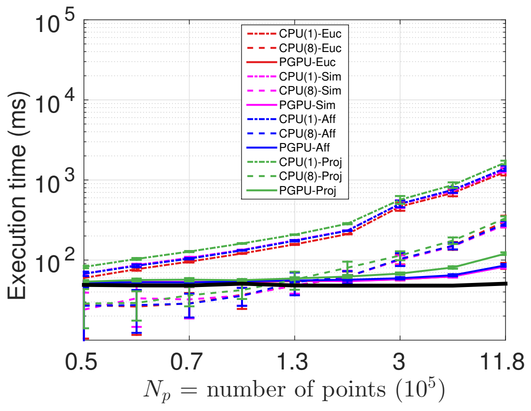
As noted in Section 3.2.2, the proposed pipelined LM implementation saves a significant amount of time when the candidate solution is accepted but causes a small overhead when the candidate solution is rejected. We quantify the speed-up of the PGPU implementation over the TGPU implementation as the percentage of candidates accepted ranges from 100 through 0. For the TGPU implementation, we use the NVIDIA$^{\text{TM}}$ Profiler to measure the execution time T_f for the GPU kernel that calculates the objective function at the candidate solution, and the execution time T_{gh} for the GPU kernel that calculates the gradient and the Hessian approximation at the candidate solution. Similarly, for the PGPU implementation, we measure the execution time T_{fgh} for the GPU kernel that calculates the objective function, the gradient, and the Hessian approximation. The speed-up S of PGPU over TGPU can then be estimated as

$$S = \frac{A(T_{gh} + T_f) + (100 - A)T_f}{100 T_{fgh}}, \tag{11}$$

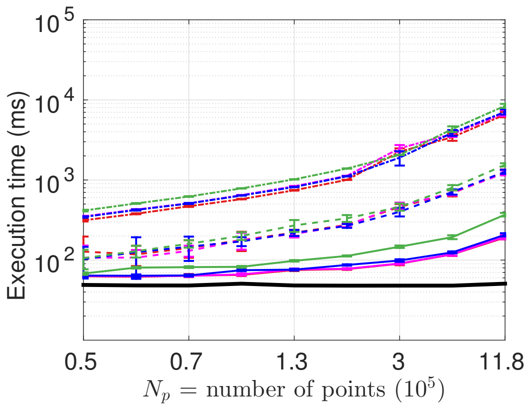
where $0 \leq A \leq 100$ is the percentage of the candidates accepted. Figure 12 shows the speed-up factor for Euclidean, similarity, affine, and projective transformation types as the acceptance percentage A varies from 100 through 0.

We draw three important conclusions from Fig. 12. First, S is maximum when $A = 100$, and starts to decrease with the decrease of A. Because the PGPU causes a small overhead when the candidate solution is rejected, the overhead is accumulated with the decrease in A and reduces the speed-up factor. Second, for a specific transformation, S is inversely related to the number of computations required to estimate the parameters of the transformation, for the same value of A. For example, for same value of A, S is higher for the Euclidean transformation than for other transformation types. Because the calculations for the Euclidean case are simpler than other

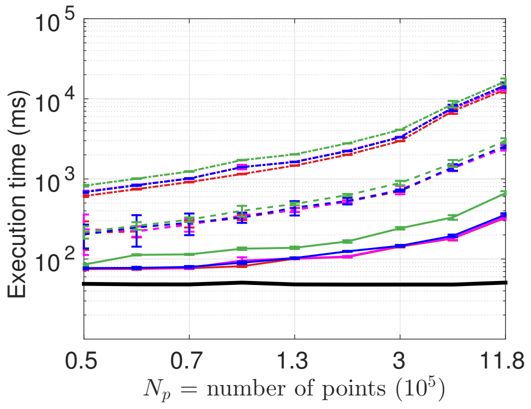
[5] The DT is calculated using the method in [25].



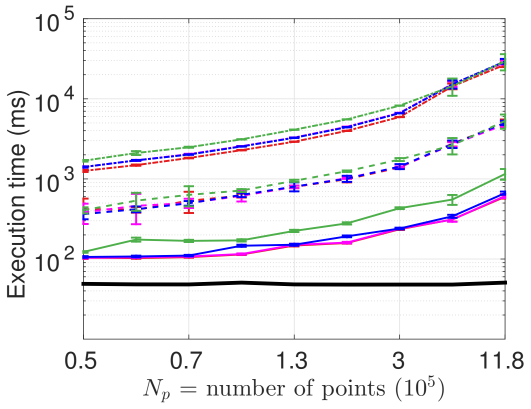
(a) 10 iterations



(b) 50 iterations

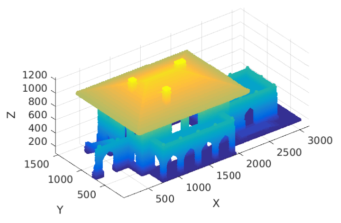


(c) 100 iterations

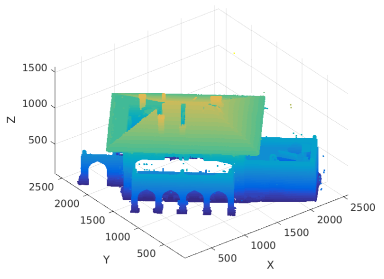


(d) 200 iterations

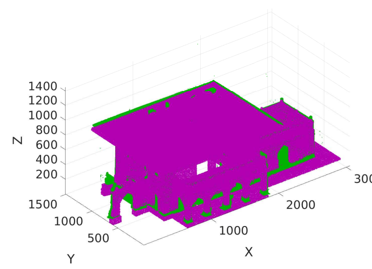
Fig. 8: Average execution time for aligning the 2D semi-synthetic WAMI datasets with a corresponding roadmap as a function of the number of observed points, obtained using the PGPU, CPU(1), and CPU(8) implementations, and executed for (a) 10, (b) 50, (c) 100, and (d) 200 iterations. In all plots, we report the average execution time over 100 runs and the standard deviations are shown as error bars. The thick black line represents the average time required for the initial large data transfer to the GPU global memory. As shown in the plots, the PGPU always has a better performance than the CPU(1). Additionally, the PGPU has a better performance than the CPU(8) when the PGPU is either executed for a sufficiently large number of iterations or applied for a large number of points. Otherwise, the CPU(8) has better performance because the execution time of the PGPU in this case is dominated by the overhead associated with the time required for the initial data transfer.



(a) Reference 3D model (RS)

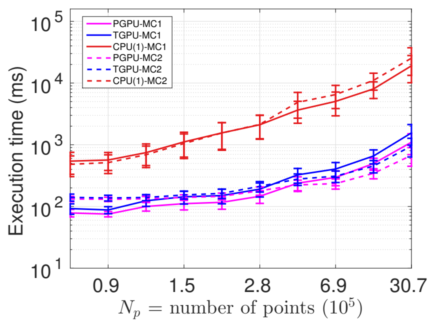


(b) Observed 3D range data (OS)

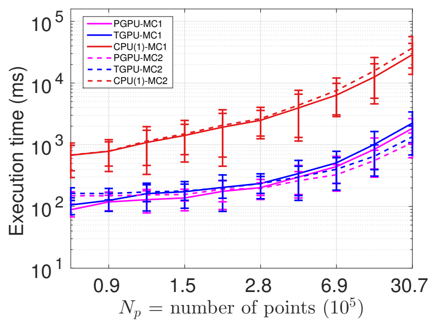


(c) Alignment result

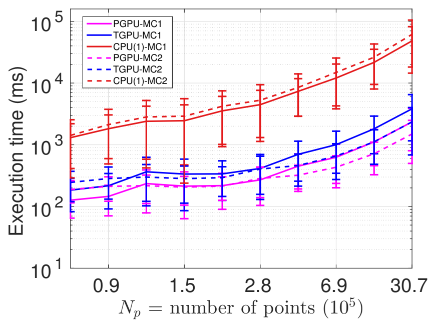
Fig. 9: Example of PChA applied for aligning an observed 3D range point dataset for a building to a corresponding 3D reference model: (a) 3D reference model, (b) the range data for a corresponding building, and (c) the alignment result visualized as an overlay of the observed building 3D range data (shown in green) with the reference 3D model (shown in magenta) for the estimated optimal alignment parameters.



(a) Euclidean



(b) Similarity



(c) Affine

Fig. 10: Average execution time for aligning the 3D semi-synthetic range-image dataset for a building with a corresponding model as a function of the number of observed points for: (a) Euclidean, (b) similarity, and (c) affine transformation types. In this semi-synthetic experiment, we use observed range datasets (OS) with different number of points generated from the reference range dataset (RS) by applying a synthetically generated transformation. An alignment transformation is then estimated by PChA using the PGPU, the TGPU, and the CPU(1) implementations. Reported values correspond to averages of the execution time over 100 runs corresponding to 100 different synthetic transformations. Standard deviations are shown as error bars and the execution time for an implementation on MC1 and MC2 is shown in the same color but with different line styles (solid lines for MC1 and dashed lines for MC2).

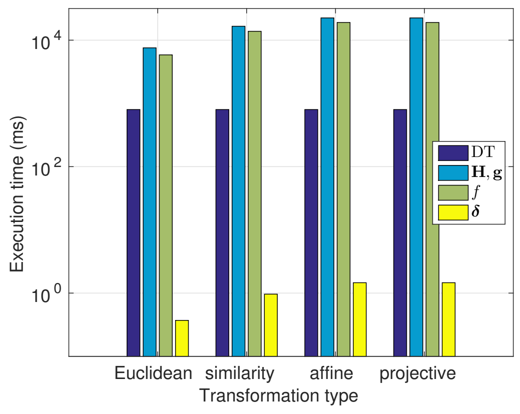


Fig. 11: PChA computational profile: The bar graphs show the average execution time for each component of the PChA for the single CPU implementation CPU(1) running on MC2 for solving a 2D WAMI semi-synthetic dataset with 1 million points. Averages are computed over 1000 runs. The plots confirm that the parallelized in the GPU implementation presented in this paper targets the computationally most demanding components of PChA (computation of f, $\mathbf{H}$, and $\mathbf{g}$), whereas the contribution of the non-parallelized component (computation of $\boldsymbol{\delta}$) to the overall execution time is several orders of magnitude smaller than that of the other components.

transformation types, T_{fgh} is closer to T_f in the Euclidean case than for other transformation types, which translates to a higher speed-up for the same value of A. Finally, $S > 1$ for Euclidean, similarity, affine, and projective transformation types, when $A > 24\%$, $A > 27\%$, $A > 40\%$, and $A > 80\%$, respectively. To improve performance in practice, our implementation performs parameter estimation using the hierarchy of transformation types in order of increasing generality and number of computations required for estimating the parameters. For example, to estimate projective transformation parameters, we proceed in four stages estimating in sequence parameters for Euclidean, similarity, affine, and then projective transformations, where at each stage the parameters are initialized using the best estimates for the preceding simpler transformation. This hierarchical parameter estimation procedure ensures good initialization for the more complex transformation types and results in a higher effective speed-up factor because of both increased acceptance percentage and fewer iterations for the complex transformations.

7 Conclusion

In this paper, we propose a parallel and pipelined realization of parametric chamfer alignment (PChA) for GPU implementation. The proposed parallelization achieves a significant speed-up by partitioning the OS points across different GPU blocks and decomposing the expensive per-point gradient and Hessian computations required for the LM iterations in PChA in correspondence with the GPU's single instruction multiple thread (SIMT) architecture. Additional speed-up is obtained by the proposed pipelining of the LM algorithm. The pipelining exploits the GPU's low arithmetic latency compared with high global memory access latency by precomputing gradient and Hessian at candidate solution preemptively even before a decision to accept or reject the candidate is made. The preemptive computation eliminates

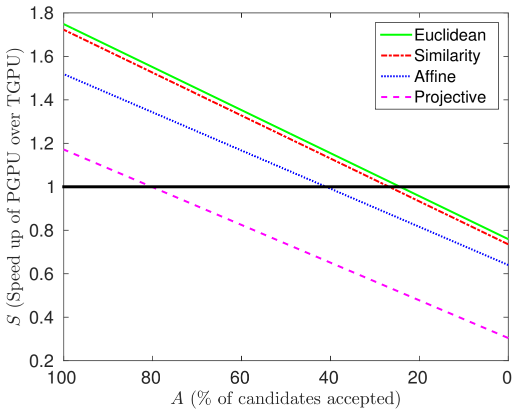


Fig. 12: The speed-up of the PGPU implementation over the TGPU implementation as the percentage of candidates accepted ranges from 100 through 0, for Euclidean, similarity, affine, and projective transformation types.

the need for fetching data a second time from high latency GPU global memory and offers a speed-up in typical application scenarios. Results obtained on two different computer systems for both large scale 2D and 3D point data sets from our ongoing research demonstrate that our PChA GPU implementation offers a very significant speed-up over its single CPU counterpart. Specifically, the speed-up allows us to achieve real-time PChA based alignment of vector roadmaps to wide area motion imagery (WAMI).

A Compositional approach for projective transformation estimation

The Jacobian matrix elements $\{J^c_{j,l}\}$ associated with the projective transformation in Table 1 require a division operation per-element, which is computationally expensive. To simplify the Jacobian calculations, we adopt a compositional approach [31] that eliminates the division operations and also enables further simplifications. The compositional approach for our 2D point set alignment by projective transformation proceeds as follows:

– The projective transformation defined by the current estimate $\boldsymbol{\alpha}_t$ of the parameters is applied to each OS point $\mathbf{p}_j$ to obtain a corresponding warped point $\mathbf{p'}_j = \mathcal{T}_{\boldsymbol{\alpha}_t}(\mathbf{p}_j)$.
– Each LM iteration, then estimates the incremental parameter update $\boldsymbol{\delta}$ that minimizes

$$f(\boldsymbol{\delta}) = \sum_{j=1}^{N_p} \left\| \mathbf{r}\left(\mathcal{T}_{(\boldsymbol{\alpha}^I+\boldsymbol{\delta})}\left(\mathbf{p'}_j\right)\right)\right\|^2, \tag{12}$$

where $\boldsymbol{\alpha}^I = [1,0,0,0,1,0,0,0]$ is the parameter vector that corresponds to the identity transformation, i.e., $\mathcal{T}_{\boldsymbol{\alpha}^I}\left(\mathbf{p'}_j\right) = \mathbf{p'}_j$.
– The updated projective transformation is obtained as

$$\mathcal{T}_{\boldsymbol{\alpha}_{t+1}} = \mathcal{T}_{\boldsymbol{\alpha}_t} \circ \mathcal{T}_{\boldsymbol{\delta}}, \tag{13}$$

where $\circ$ denotes composition, or equivalently multiplication of the corresponding matrix representations.

Considerable simplification of the Jacobian matrix calculation is obtained because the calculation is performed at $\boldsymbol{\alpha}^I$, where the term w_j in Table 1 becomes unity, eliminating the need for division operations. Specifically, the Jacobian matrix $\mathbf{J}_j$ at the transformed point $\mathbf{p'}_j \equiv \mathcal{T}_{\boldsymbol{\alpha}^I}(\mathbf{p'}_j)$, is computed as

$$\mathbf{J}_j = \frac{\partial \mathcal{T}_{\boldsymbol{\alpha}}(\mathbf{p'}_j)}{\partial \boldsymbol{\alpha}}\bigg|_{\boldsymbol{\alpha}=\boldsymbol{\alpha}^I} = \begin{pmatrix} x' & y' & 1 & 0 & 0 & 0 & -x'^2 & -x'y' \\ 0 & 0 & 0 & x' & y' & 1 & -x'y' & -y'^2 \end{pmatrix}. \tag{14}$$

The Hessian matrix approximation elements are shown in Table 2, where additional simplifications are also incorporated.

Acknowledgement

We thank Bernard Brower of Harris Corporation for making available the CorvusEye [30] WAMI datasets used for demonstrating PChA on real-world 2D datasets. We also thank our colleagues from the Architectural Bio-metrics project for providing the 3D datasets of building models and lidar scans that are used in our evaluation. We also thank the Center for Integrated Research Computing (CIRC), University of Rochester, for providing access to computational resources for this research.

References

1. Ming-Yu Liu, O. Tuzel, A. Veeraraghavan, and R. Chellappa. Fast directional chamfer matching. In *IEEE Intl. Conf. Comp. Vision, and Pattern Recog.*, pages 1696–1703, June 2010.
2. Hongjian Jiang, Kerrie S Holton, and Richard A Robb. Image registration of multimodality 3-D medical images by chamfer matching. In *SPIE/IS&T 1992 Symposium on Electronic Imaging: Science and Technology*, pages 356–366. International Society for Optics and Photonics, 1992.
3. Yu-Tseh Chi, SM Nejhum Shahed, Jeffrey Ho, and Ming-Hsuan Yang. Higher dimensional affine registration and vision applications. In *Proc. European Conf. Computer Vision*, pages 256–269. Springer, 2008.
4. Faysal Boughorbel, Muharrem Mercimek, Andreas Koschan, and Mongi Abidi. A new method for the registration of three-dimensional point-sets: The Gaussian fields framework. *Comp. Vis. and Image Understanding.*, 28(1):124 – 137, 2010.
5. Adrien Gressin, Clment Mallet, Jrme Demantk, and Nicolas David. Towards 3D lidar point cloud registration improvement using optimal neighborhood knowledge. *Journal of Photogrammetry and Remote Sensing*, 79:240 – 251, 2013.
6. Martin Danelljan, Giulia Meneghetti, Fahad Shahbaz Khan, and Michael Felsberg. A probabilistic framework for color-based point set registration. In *IEEE Intl. Conf. Comp. Vision, and Pattern Recog.*, pages 1818–1826, June 2016.
7. L. Ding, A. Elliethy, E. Freedenberg, S. A. Wolf-Johnson, J. Romphf, P. Christensen, and G. Sharma. Comparative analysis of homologous buildings using range imaging. In *IEEE Intl. Conf. Image Proc.*, pages 4378–4382, Sept 2016.
8. Ahmed Elliethy and Gaurav Sharma. Vector road map registration to oblique wide area motion imagery by exploiting vehicles movements. In *IS&T Electronic Imaging: Video Surveillance and Transportation Imaging Applications*, pages VSTIA–520.1–8, San Francisco, California, 2016. URL

http://ist.publisher.ingentaconnect.com/contentone/ist/ei/2016/00002016/00000003/art00008.

9. Ahmed Elliethy and Gaurav Sharma. Automatic registration of vector road maps with wide area motion imagery by exploiting vehicle detections. *IEEE Trans. Image Proc.*, 25(11):5304 – 5315, November 2016. doi: 10.1109/TIP.2016.2601265.
10. P. J. Besl and H. D. McKay. A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intel.*, 14(2):239–256, Feb 1992.
11. Zhengyou Zhang. Iterative point matching for registration of free-form curves and surfaces. *Intl. J. Computer Vision*, 13(2):119–152, 1994.
12. A. Myronenko and X. Song. Point set registration: Coherent point drift. *IEEE Trans. Pattern Anal. Mach. Intel.*, 32(12):2262–2275, Dec 2010.
13. M. Sofka, G. Yang, and C. V. Stewart. Simultaneous covariance driven correspondence (CDC) and transformation estimation in the expectation maximization framework. In *IEEE Intl. Conf. Comp. Vision, and Pattern Recog.*, pages 1–8, June 2007.
14. Andrew W Fitzgibbon. Robust registration of 2D and 3D point sets. *Image and Vision Computing*, 21(1314): 1145 – 1153, 2003.
15. M. Rouhani and A. D. Sappa. Correspondence free registration through a point-to-model distance minimization. In *IEEE Intl. Conf. Comp. Vision.*, pages 2150–2157, Nov 2011.
16. Gunilla Borgefors. Distance transformations in digital images. *Comp. Vis., Graphics and Image Proc.*, 34(3): 344–371, June 1986.
17. J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.
18. H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *Proc. Int. Joint Conf. Artificial Intell.*, pages 659–663, 1977.
19. C. Sigg, R. Peikert, and M. Gross. Signed distance transform using graphics hardware. In *IEEE Visualization*, pages 83–90, Oct 2003.
20. Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 83–90, New York, NY, USA, 2010. ACM.
21. Xiang Zhu and Dianwen Zhang. Efficient parallel Levenberg-Marquardt model fitting towards real-time automated parametric imaging microscopy. *PloS one*, 8(10):e76665, 2013.
22. Bo Li, Alistair A Young, and Brett R Cowan. GPU accelerated non-rigid registration for the evaluation of cardiac function. In *Medical Image Computing and Computer-Assisted Intervention*, pages 880–887. Springer, 2008.
23. R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos. Comparing CUDA and OpenGL implementations for a Jacobi iteration. In *IEEE Intl. Conf. High Performance Computing Simulation*, pages 22–32, June 2009.
24. Architectural Biometrics Project. https://architecturalbiometrics.com/.
25. Pedro Felzenszwalb and Daniel Huttenlocher. Distance transforms of sampled functions. Technical Report TR2004-1963, Cornell University, 2004. URL https://ecommons.cornell.edu/handle/1813/5663.
26. David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.
27. Mark Harris. Optimizing parallel reduction in CUDA. 2007. NVIDIA Developer Technology.
28. The OpenMP API specification for parallel programming. http://www.openmp.org/.
29. University of Rochester, BlueHive Cluster. https://info.circ.rochester.edu/BlueHive/System_Overview.html.
30. CorvusEye$^{\text{TM}}$1500 Data Sheet. http://www.exelisinc.com/solutions/corvuseye1500/Documents/CorvusEye500DataSheetAUG14.pdf.
31. Richard Szeliski and Heung-Yeung Shum. Creating full view panoramic image mosaics and environment maps. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 251–258, 1997.

Ahmed Elliethy received the B.Sc. degree (excellent with honors) in computer engineering and the M.Sc. degree in electrical engineering from the Military Technical College, Cairo, Egypt, in 2003, and 2010, respectively. He is currently pursuing his Ph.D. degree at University of Rochester. His research interests are computer vision and security, specifically, multiple object tracking, optical flow, and media forensics.

Gaurav Sharma is a professor at the University of Rochester in the Department of Electrical and Computer Engineering, in the Depart- ment of Computer Science and in the Department of Biostatistics and Computational Biology. From 2008 to 2010, he served as the Director for the Center for Emerging and Innovative Sciences (CEIS), a New York state funded center for promoting joint universityindustry research and technology development, which is housed at the University of Rochester. He received the BE degree in electronics and communication engineering from Indian Institute of Technology Roorkee (formerly Univ. of Roorkee), India, in 1990; the ME degree in electrical communication engineering from the Indian Institute of Science, Bangalore, India, in 1992; and the MS degree in applied mathematics and Ph.D. degree in electrical and computer engineering from North Carolina State University, Raleigh, in 1995 and 1996, respectively. From August 1996 through August 2003, he was with Xerox Research and Technology, in Webster, NY, initially as a Member of Research Staff and subsequently at the position of Principal Scientist. Dr. Sharmas research interests include image processing and computer vision, color science and imaging, multi- media security and watermarking, and bioinformatics. He is the editor of the Color Imaging Handbook, published by CRC press in 2003. He is a fellow of the IEEE, of SPIE, and of the Society of Imaging Science and Technology (IS&T) and a member of Sigma Xi.