2013 IEEE/RSJ International Conference on
Intelligent Robots and Systems (IROS)
November 3-7, 2013. Tokyo, Japan

GPU Accelerated Graph SLAM and Occupancy Voxel Based ICP For Encoder-Free Mobile Robots

Adrian Ratter[†], Claude Sammut[†] and Matthew McGill[†]

Abstract— **Learning a map of an unknown environment and localising a robot in it is a common problem in robotics, with solutions usually requiring an estimate of the robot's motion. In scenarios such as Urban Search and Rescue, motion encoders can be highly inaccurate, and weight and battery requirements often limit computing power. We have developed a GPU based algorithm using Iterative Closest Point position tracking and Graph SLAM that can accurately generate a map of an unknown environment without the need for motion encoders and requiring minimal computational resources. The algorithm is able to correct for drift in the position tracking by rapidly identifying loops and optimising the map. We present a method for refining the existing map when revisiting areas to increase the accuracy of the existing map and bound the run-time to the size of the environment.**

I. INTRODUCTION

In this paper we present a highly accurate, motion encoder free, on-line Graph SLAM algorithm with loop closing capable of running in real time on low cost and low power mobile hardware. As odometry can be highly unreliable in many situations, such as in a disaster site with loose rubble [1], and is subject to drift over time, our algorithm is designed to require no motion encoder information. We instead provide position tracking updates through using laser distance readings as input into an occupancy grid based *Iterative Closest Point* (ICP) algorithm [2]. The algorithm takes full advantage of the highly parallel nature and unique memory model structure of GPUs.

As any position tracking solution will accrue errors over time, a method of loop closing is required to maintain an accurate map. We use the output of our position tracking to generate a series of local maps, which are added as vertices into a graph. Edges, or *constraints*, are labelled with the displacement between local maps, based on the position tracking between adjacent maps or from loop closing, and the certainty of that displacement, based on the gradient of the fit between the local maps. We present a novel three stage algorithm that efficiently detects loop closures with the current local map. Firstly local maps in the correct area are identified using covariance estimates from the constraints. The structural similarity of these maps are compared with the current map using histogram correlations to quickly identify similar maps independent of offset errors. The histogram match is then used as an initial estimate into an ICP alignment. After a loop closure has been identified, we optimise the graph to increase the likelihood of the observations.

[†]School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia. {adrianr, claude, mmcgill} at cse.unsw.edu.au

A drawback with many Graph SLAM methods is that the number of local maps increases over time. Not only does this consume large amounts of memory, but it also slows down graph optimisation and loop closure. We present a method to combine local maps and refine constraints between them when a robot is traversing through already visited areas, instead of creating new local maps.

Recently, GPUs have started to be used for robotics applications, with their high data-throughput and data-parallelism being used to speed up algorithms. However, their highly limited synchronisation and restricted memory model means that performance rapidly decreases with branching threads, random memory access and data-access synchronisation [3]. This is particularly relevant for occupancy grid based ICP algorithms, as there is a need for hundreds of threads to add scan points and modify the grid in parallel without causing data corruption. We propose a method of storing the occupancy grid as a lookup table into an active cells list to enable efficient parallel shifting and modification of the map, while retaining the fast corresponding point search advantage of occupancy grids. This structure is used for both the position tracking and the local map matching. Additionally, we propose a method to optimise the graph in parallel instead of the standard iterative constraint optimisation [4].

II. RELATED WORK

One of the best solutions to mapping on-board resource limited robots was introduced by Kohlbrecher et al. [5]. Their method uses an occupancy grid ICP based position tracker and inertial sensors to generate input for an EKF. While accurate over small scale scenarios and highly computationally efficient, it is not able to close loops in the case of any drift in the position tracking, and it uses motion encoders. An alternate solution presented by Milstein et al. [6] is to use an occupancy grid ICP algorithm as input for a FastSLAM particle filter. However, running the particle filter with enough points to have a high chance of generating an accurate map is computationally expensive, and is therefore performed on an off-board computer.

An alternate SLAM approach, introduced by Gutmann et al. [7] involves combining a series of local maps into a global map. The Hierarchical SLAM algorithm [8] generates a series of local feature based maps, and combines them into an adjacency graph. After loops are closed by detecting similar features, the algorithm applies a global optimisation procedure to correct the global map. A similar solution, that does not have the drawback of using features, instead performs the loop closure detection by histogram correlation

978-1-4673-6357-0/13/$31.00 ©2013 IEEE

of local maps [9]. This allows local maps to be rapidly compared for similarity regardless of alignment errors. We use this idea as part of our three stage loop closure detection. An alternate loop closure strategy presented by Granstrom et al. [10] involves automatic learning of features from laser range scans, but comes at an additional processing expense.

The other major part of Graph SLAM approaches is the method of optimising the graph of local maps to produce a consistent global map. The ATLAS framework [11] applies an expensive global optimisation procedure. Olson et al. [4] applies stochastic gradient descent to reduce errors in the graph, and produce a representation of the global position of the local maps to enable efficient updates. Grisetti et al. [12] extended this method to lessen the number of nodes that need to be updated during optimisation by changing the representation of the global position of local maps into a tree structure. This approach allows the possibility of local maps to be combined when the robot traverses through already visited areas, thereby limiting the complexity to the size of the environment instead of the length of the trajectory. We use a very similar method to optimise the graph, but we relax the optimisation procedure to allow constraints to be optimised in parallel. Mechanisms to lessen the influence of false positive loop closure during the optimisation process, such as introduced by Agarwal et al. [13], could be easily added to our method if required. More complex optimisation techniques, such as by Sunderhauf and Protzel [14], have recently been described and could easily be used as part of our SLAM system, however our results show that the added computational expense is not necessary in many real world environments.

Common drawbacks of many ICP algorithms, such as those in the Point Cloud Library [15], are the computationally intensive search for point correspondences, and that only aligning to the previous scan allows errors in a single scan to cause the algorithm to fail. These issues can be avoided by storing previous scans in an occupancy grid [16] [5]. Storing a history of past scans in an occupancy grid enables a single 2D scan from a tilting laser to be aligned to a 3D grid. The OG-MBICP algorithm [16] uses a metric for finding correspondences that considers rotational and translational movements in a single pass as rotations in the sensor can cause points distant from the sensor to be far away from their corresponding point [17]. We use this algorithm as the basis for our parallel ICP, but our approach could easily be followed using another occupancy grid ICP algorithm.

Recently, GPUs have started to be used for robotics applications, with Olson [18] and Izadi et al. [19] presenting the first mappings of ICP algorithms onto a GPU. The Kinect Fusion algorithm [19] uses ICP to align successive 3D scans to generate a surface map of an area. While the ICP alignment in this algorithm does mean position tracking information can be extracted, it is not able to perform loop closure, and the surface reconstruction aim of the alignment introduces significant computational complexities that are not needed for position tracking. Alternatively, Clipp et al. [20] demonstrated a form of visual based SLAM on a GPU.

III. BACKGROUND

A. GPU Programming

GPUs are based on Single Instruction, Multiple Data (SIMD) architecture. Each GPU has at least one *streaming multiprocessor* (SM), and each streaming multiprocessor has several, typically between 8 and 32, *streaming processor cores* (SP) and on-chip local memory. All the SPs residing on a single SM execute the same instruction at the same time. In the event of branching code, threads residing on a SP but not participating in the branch are paused. As threads on a GPU are extremely lightweight and can be swapped almost instantaneously, GPUs are most efficient when running with hundreds or thousands of threads.

GPUs have two levels of shared memory: local and global. Local memory of a SM resides on-chip and operates at full clock speed, while global memory is significantly slower to access, and is optimised for *coalesced* memory accesses—global memory transactions occur in serialised blocks of 16 or 32 words. If all threads currently executing on a SM request memory from the same block (a coalesced memory access), only one global memory transaction is needed.

As each SP in a SM executes the same instructions at the same time, a high level of synchronisation is available to threads on a SM that are active at the same time. These subgroups of threads, known as a *warp*, are guaranteed by the hardware structure to always be at identical positions in the execution of the kernel. GPUs also provide atomic operations to local and global memory to prevent data corruption, while standard barrier synchronisation is only available to threads residing on the same SM.

B. Occupancy Grid Metric Based ICP

ICP is an iterative algorithm that searches for correspondences between scans to find the alignment of the new set of points $S_{new} = \{p'_i\}$ in the existing environment [2]. An alignment in three dimensions is defined as $q = (x, y, z, \theta)$, assuming the pitch and roll of the robot is known a priori, such as from an attitude sensor. The algorithm works by finding the nearest existing point p_i (normally the closest point on the line between successive points is used (p_i, p_{i+1})) to each new point p'_i and then finding the alignment, q_{min}, that minimises the mean squared error between p_i and the transformation of p'_i by the alignment, as shown in (Eq. 1). All points in S_{new} are transformed by q and the process is repeated until q converges.

$$E_{dist}(q) = \sum_{i=1}^{n} d(p_i, q(p'_i))^2 \quad (1)$$

MBICP uses a metric to allow the translation and orientation of successive scans to be determined in one pass [17]. The metric defines a constant, L, that controls the relative importance of angular displacement against linear displacement.

$$\|q\| = \sqrt{x^2 + y^2 + z^2 + L^2\theta^2} \quad (2)$$

The distance between two points, p_1 and p_2 is defined as:

$$d(p_1, p_2) = min\{\|q\| \textit{ such that } q(p_1) = p_2\} \quad (3)$$

Therefore, the ICP algorithm consists of searching for corresponding points for each new point p'_i by solving (Eq. 3) for each point. Once all the corresponding points have been found, the alignment q_{min} that minimises (Eq. 1) can be found. All full derivation of the Metric Based ICP algorithm can be found in [16].

The Occupancy Grid Metric Based ICP (OG-MBICP) algorithm [16] stores previously observed points inside an occupancy grid, so that the process of finding corresponding points is simplified to searching for observed points in nearby cells in the grid to the new point. No accuracy is lost in using the occupancy grid as the alignment is still made to the stored points and not the grid itself.

C. Graph SLAM

A typical formulation of the graph SLAM problem consists of a series of nodes, representing local maps, that are connected by edges, representing an observed displacement from one node to another node. The edges in the graph, normally referred to as *constraints*, are generated from movement between nodes (the change in displacement from our position tracking algorithm between adjacent local maps), or from loop closing. The aim of graph SLAM algorithms is to find a configuration of the nodes that maximises the likelihood of the observations. We use the definitions provided by Grisetti et al. [12] to describe the problem:

- δ_{ba} is a constraint from node a to node b. It is an observation of node b from node a.
- Ω_{ba} is an information matrix describing the uncertainty of the constraint δ_{ba}.
- P_a is the current global position of local map a.
- x_a is a parameterisation of the position of node a for the current configuration. We define it to be $x_a = P_a - P_{parent(a)}$, where the parent of node a is the local map previously visited by the robot. $\mathbf{x}$ describes the vector of these parameters for the current configuration.
- $f_{ba}(\mathbf{x})$ is a function that calculates the expected observation of node b from node a given the current graph configuration.
- $e_{ba}(\mathbf{x})$ is the current error in the constraint δ_{ba} and is given by $e_{ba}(\mathbf{x}) = f_{ba}(\mathbf{x}) - \delta_{ba}$.
- $r_{ba}(\mathbf{x})$ is the residual in the constraint, and is defined as $r_{ba}(\mathbf{x}) = -e_{ba}(\mathbf{x})$.
- $\mathbf{H}$ is the Hessian matrix of the graph, and represents the curvature of the error function. This can be approximated as $\mathbf{H} \simeq \sum_{\langle b,a \rangle} J_{ba}\Omega_{ba}J_{ba}^T$, where the Jacobian, J_{ba}, is $J_{ba} = \sum_{i=a+1}^{b} \iota_i$. Here, $\iota_i = (0_0 \quad 0_1 \ldots 0_{i-1} \quad I_i \quad 0_{i+1} \ldots 0_N)$, where 0 is a 3 by 3 matrix of zeros and I is the 3 by 3 identity matrix.

If the observations are assumed to be independent, the negative log likelihood of a configuration $\mathbf{x}$ is given by:

$$F(\mathbf{x}) = \sum_{\langle b,a \rangle \epsilon \mathbf{C}} r_{ba}(\mathbf{x})^T \Omega_{ba} r_{ba}(\mathbf{x}) \tag{4}$$

where $\mathbf{C}$ is the set of pairs of indices that are connected by a constraint. The aim of the graph optimisation step is to find the configuration $\mathbf{x*}$ that minimises (Eq. 4).

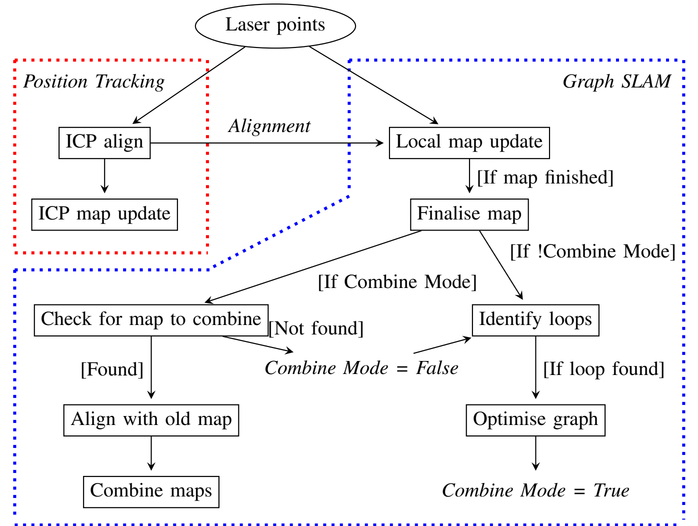

Fig. 1: Overview of our SLAM system.

D. Histogram Matching

As laser points are added to the current local map, we use them to generate orientation, projection and entropy histograms, using the method described by Bosse and Zlot [9]. These histograms are used as part of our loop closure algorithm to quickly compare the similarity of two local maps, regardless of their initial offsets. The orientation histograms for each map can be used to compute the difference in rotation between two local maps by performing a correlation, irrespective of any translational differences. They are calculated by adding the normal of each point n_i (calculated from the inverse gradient of the line joining p_i and p_{i+1}) to the histogram.

The projection histograms can give the translational offset once the rotational offset has been calculated. A projection histogram is calculated for each bin, θ_p, of the orientation histogram, by orthogonally projecting the laser points, $(p_{i,x}, p_{i,y})$ onto the line with the angle θ_p, and adding the point to the bin, d_i, it is projected onto. This is given by: $d_i = p_{i,x}\cos(\theta_p) + p_{i,y}\sin(\theta_p)$. When being added to the histogram, the points are weighted according to their surface orientation to enhance the peaks (a surface nearly parallel to a projection line gives little information, so the points in it have a small weight), given by: $n_{i,x}\cos(\theta_p) + n_{i,y}\sin(\theta_p)$.

The final histogram created is the entropy histogram, which measures the uniformity of the projection histograms. A projection histogram with evenly distributed points will have a large entropy, while histograms with only a small number of bins used will have a small entropy. Bosse and Zlot [9] found that entropy sequences worked more accurately in unstructured environments than the orientation histogram, and a full derivation of the entropy histograms is available in [9]. All histograms are normalised while a local map is being finalised, to ensure the correlation of any histograms can never exceed one.

IV. APPROACH

Our SLAM algorithm consists of two major components: a 3D occupancy grid ICP algorithm provides position updates, while a 2D Graph SLAM algorithm generates an array of local maps as the robot moves around its environment. Once a local map has finished being created, it is added into a graph, and compared against the other local maps for similarity. If a similar local map is found, a loop closing constraint is added, and the graph is optimised to increase the likelihood of all the observations. After a loop closure, if the robot starts traversing areas of the environment it has already visited, the new local maps are combined with the previous local map of the area. An overview of this algorithm is shown in Fig. 1. All stages are designed for use on a GPU to maximise the efficiency of the algorithm.

A. Occupancy Grid ICP

Algorithm 1 Psuedocode of our parallel ICP algorithm

```
1: function PARALLEL ICP(og, ogPoints, points, q_init)
2:     q ← q_init
3:     repeat
4:         for all p in points do
5:             p_q ← (transform p by q)
6:             search og near p_q to minimise Eq. 3
7:         end for
8:         q ← q + (q_min from Eq. 1)
9:     until converged
10:    clear og using ogPoints
11:    if position tracking then
12:        shift ogPoints by q
13:        add ogPoints to og
14:        add points to og and ogPoints
15:    end if
16: end function
```

ICP is used in our algorithm to align successive laser scans for position tracking, and as part of the test for similarity to detect loop closures. As it needs to run frequently, it is important that it be both as fast and as accurate as possible. We designed our occupancy grid ICP algorithm to make it efficient when deployed on a GPU, with a particular focus on a flexible map structure to enable it to be modified, shifted, and have points added and removed from it by hundreds of threads without causing memory corruption.

Our algorithm solves this problem by splitting the map in two parts—a list of active two dimensional cells, and a two dimensional grid map of the area around the robot storing indexes into the active cells list. Each active cell contains a list of sub-cells for the z dimension, and each sub-cell stores points that were previously observed in it. Active cells also store their position in the grid map, a total count of how many times the cell has been observed and when the cell was last observed when the algorithm is being used for position tracking. Information about active cells, such as the observation count, are stored as arrays containing that

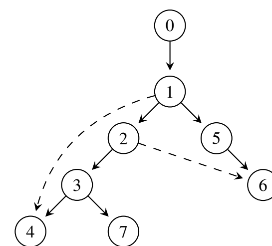

Fig. 2: An example graph structure. Solid lines represent constraints from position tracking, and dashed lines represent constraints from loop closing.

information for all active cells to ensure coalesced memory access for threads performing actions on active cells.

Once an alignment q has been found for the new points S_{new}, the active cells list is used to quickly clear the occupancy grid. If the algorithm is being used for position tracking, the grid map is then shifted by the alignment so the robot remains centred in the grid, the new points are added to the map, and active cells that haven't been observed recently are deleted. This process is shown in Algorithm 1.

By far the most computationally expensive part of ICP is finding the point in the occupancy grid that minimises (Eq.3) for each laser point. In our algorithm, the work to find a single corresponding point is split over all threads in a warp. At the start, each thread is given a different cell in the grid map to process, such that the 32 closest cells to the laser point in the grid map are each allocated a thread. Each thread then calculates d from (Eq. 3) for the centre of the three sub-cells nearest in height to the laser point if the sub-cell is occupied, and selects the sub-cell with the lowest d. The threads in the warp then choose the best four matching sub-cells. All this can be performed without any synchronisation as all threads involved are in the same warp. Finally, the threads calculate d for each of the points stored in the closest sub-cells, and chose the best match.

B. Building the Graph

As the robot moves around, laser scans and their ICP alignment from the position tracking are added to the current local map in the graph SLAM algorithm. The local map stores the movement of the robot (from the ICP alignment) since the local map was created, and transforms the laser points by the total movement so that all laser points are relative to the position of the robot at the start of the map. This allows the location of a local map to be described by the position of the robot when the map was created.

A new local map is created after the robot has moved a certain distance inside the current local map. This distance should be small enough such that the unavoidable drift from slight errors in the position tracking are insignificant, but large enough to keep the number of local maps needed to describe the environment as small as possible. In our office environment, we found that a movement of around 2m gave the best results, and example local maps are shown in Figs. 3d and 3e. Each local map stores its current global position,

which is calculated from the global position of the parent local map plus the displacement between the parent local map and the new local map. This displacement is stored in the new local map as the constraint δ_{ba}. Normally, the parent of a new local map is the current local map, so the displacement is the robot's movement inside the current local map. If the current local map closed a loop, the parent of the new local map is set to the local map that matched the current map. Also, if the current local map was combined with an older map, the parent is set to the older map. This creates a tree of local maps, where each node is joined by the constraints with their parents. An example of this structure is shown in Fig. 2, where node 1 is the parent of node 5 as node 1 closed a loop with node 4, and node 3 is the parent of node 7 as the local map created immediately before node 7 was combined into node 3.

When the current local map is finalised, the information matrix Ω_{ba} between it and its parent is calculated by considering the gradient of the current local map around matching points in the parent local map. Firstly, points in the parent local map, p'_i, are transformed by the parent constraint (the displacement between the two maps), $\kappa = (x, y, \theta)$, so that the points become:

$$T_i(\kappa) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} p'_{i,x} + x \\ p'_{i,y} + y \end{pmatrix} \quad (5)$$

As the points in the current local map are still stored in an occupancy grid, we find the closest matching point to each T_i in the occupancy grid, p_i, using the same metric as ICP. This operation can be written according to the function $h_i(\kappa) = M(T_i(\kappa))$. Differentiating this according to the chain rule yields an expression for the gradient of the match:

$$h'_i(\kappa) = \nabla M(T_i(\kappa)) \frac{\partial T_i(\kappa)}{\partial \kappa} \quad (6)$$

A Hessian information matrix of the point match can be constructed by $H_i = h'_i(\kappa)^T h'_i(\kappa)$, giving the overall information matrix:

$$H = \frac{1}{N} \sum_{i=1}^{N} \left(\nabla M(T_i(\kappa)) \frac{\partial T_i(\kappa)}{\partial \kappa} \right)^T \left(\nabla M(T_i(\kappa)) \frac{\partial T_i(\kappa)}{\partial \kappa} \right) \quad (7)$$

When the local map is being built, the next laser point detected, p_{i+1}, to each laser point p_i is stored in the occupancy grid along with the laser point itself. The gradient of the map around the matching point p_i, $\nabla M(T_i(\kappa))$, can therefore be approximated as the gradient of the line joining p_i and p_{i+1}.

C. Identification of Loops

When a robot returns to an area that it has previously observed, it is necessary to recognise the similarity in the current local map to the one originally created for the area so that a loop closing constraint can be added to the graph. Methods to do this have to be fast, as there can be many local maps storing thousands of laser points that need to be searched, have to be able to cope with significant errors in initial alignment, and have to rarely give false positives, as incorrectly closing a loop will give large errors in the global map. False negatives are preferred as they only delay the creation of a loop closure constraint until the robot continues to explore areas it has previously visited.

Our loop closing algorithm is a three step process:

- Based on the information matrices of the constraints around the graph, we calculate what existing local maps are in the area the current local map could be in. Each local map stores the current global covariance of the constraint between it and its parent, calculated by rotating the information matrix of the constraint by the global rotation of the parent and inverting. For each local map in the tree, j, the covariance between it and the current node, i is given by summing the covariances of the constraints on the path between the two nodes and their common parent in the tree, $E_{i \to j}$. This gives:

$$\Sigma'_{i \to j} = \sum_{(a,b) \epsilon E_{i \to j}} (R_a \Omega_{ba} R_a^T)^{-1} \quad (8)$$

  where R_a is the homogeneous rotation matrix for the current global position of node a. If the absolute value of the difference in global position between the centres of local maps i and j is less than the covariance $\Sigma'_{i \to j}$ (index $(0, 0)$ and $(1, 1)$ for (x, y) coordinates), node j is close enough to node i to warrant further examination.
- The similarity of potentially matching local maps can be determined quickly by the correlation of their histograms (see Section III-D). First, circular convolutions of the orientation and entropy histograms are performed, and peaks are identified to give an indication of possible angle offsets between the two local maps. Example orientation histograms and their correlation are shown in Fig. 3. For each peak, at rotational offset θ_r, a correlation between the maps at an offset of θ_r is performed. This means that if the projection histogram for one map has a projection line angle of θ_o, the projection histogram for the other map in the correlation will have a projection line angle of $(\theta_o + \theta_r)$. The peak value corresponds to a potential translational offset. A correlation is also performed using the projection histograms with a projection line perpendicular to this so that the translational offset can be resolved in both directions. The histograms used are $\theta_o + \frac{\pi}{2}$ for the one map, and $\theta_o + \theta_r + \frac{\pi}{2}$ for the other map. Bosse and Zlot [9] found that setting θ_o to the largest peak in the orientation histogram produced the best results.
  The closeness of a match at peak θ_r can be computed by summing the values of the entropy and orientation histograms at θ_r with the maximum values in the correlation of the two projection histograms. A value of 3.35 was empirically found to deliver good results with few false positives. Finally, if d_{θ_r} and $d_{\theta_r + \frac{\pi}{2}}$ are the offsets that correspond to the maximum values of the two projection histograms, the translational offsets, t_x and t_y can be calculated by solving:

$$\begin{pmatrix} \cos(\theta_r) & -\sin(\theta_r) \\ \sin(\theta_r) & \cos(\theta_r) \end{pmatrix} \begin{pmatrix} t_x \\ t_y \end{pmatrix} = \begin{pmatrix} d_{\theta_r} \\ d_{\theta_r + \frac{\pi}{2}} \end{pmatrix} \quad (9)$$

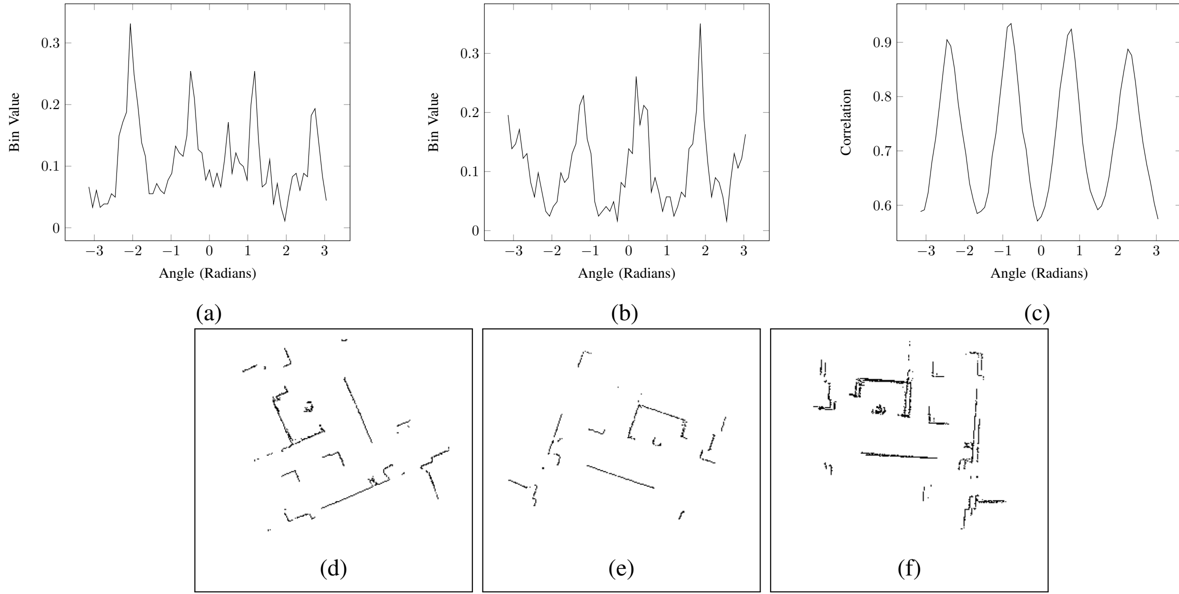

Fig. 3: The orientation histograms of matching local maps (d) and (e) are shown in (a) and (b) respectively. (c) shows the correlation of the orientation histograms, with four possible peaks. The largest peak is the correct alignment. (f) shows the rough alignment of the local maps after the histogram correlation.

- Using the rough alignment provided by the histogram correlation, similar local maps can be accurately aligned by ICP. The same method as described in Section IV-A is used. To further reduce the chance of false positive matches, if the ICP alignment doesn't converge in a set number of steps, or if the number of matching points found in the last iteration of the ICP algorithm is too low, the match is discarded.

D. Graph Optimisation

Once a loop closing constraint has been detected, the graph can be optimised to maximise the likelihood of the observations. Similarly to Grisetti et al. [12], we achieve this by using a modified version of stochastic gradient descent in moving the nodes in the graph to reduce the error introduced by each constraint, δ_{ba}. Only the nodes on the path in the tree between nodes a and b, $E_{a\rightarrow b}$, are moved. The nodes on the path are updated in the direction of the residual, $r_{ba}(\mathbf{x})$. The amount of movement is scaled by the amount of information in the constraint, and a weighting representing the curvature of the error function. This is given by:

$$\Delta x_i = \lambda s |E_{a\rightarrow b}| w_i \Omega'_{ba} r'_{ba} \tag{10}$$

where:

- r'_{ba} is the residual of the constraint δ_{ba} relative to global coordinates. It is given by $r'_{ba} = P_a + R_a\delta_{ba} - P_b = \sum_{i^{[-]}\epsilon E^{[-]}_{a\rightarrow b}} x_{i^{[-]}} - \sum_{i^{[+]}\epsilon E^{[+]}_{a\rightarrow b}} x_{i^{[+]}} + R_a\delta_{ba}$, where $E^{[-]}_{a\rightarrow b}$ is the part of the path ascending up the tree of local maps from a to the root, and $E^{[+]}_{a\rightarrow b}$ is the part of the path heading down the tree to node b.
- Ω'_{ba} is the information matrix relative to the global frame, which can be calculated by: $\Omega'_{ba} = R_a \Omega_{ba} R_a^T$.
- w_i is the weighting of the update, designed to spread the residual among all nodes on the path between a and b, with the spread weighted according to all the constraints along the path. This is given by:

$$w_i = \left[\sum_{k=a+1}^{b} D_k^{-1} \right]^{-1} D_i^{-1} \tag{11}$$

where D_k are the diagonal elements of the kth block of the Hessian $\mathbf{H}$.
- $|E_{a\rightarrow b}|$ is the path length from a to b in the tree.
- s has a value of either $+1$ if $x_i \epsilon E^{[+]}_{a\rightarrow b}$ and -1 if $x_i \epsilon E^{[-]}_{a\rightarrow b}$. This is to account for the different directions of traversal through the tree.
- λ is the learning rate, given by $\lambda = \frac{1}{\gamma t}$, where t is the iteration number of the optimisation (to make sure the optimisation converges), and $\gamma = min_{\langle b,a\rangle}\Omega_{ba}$.

The global map is optimised by firstly having a thread on the GPU for each local map calculate the constraints that have a path through it, and from this the local map's contribution towards the Hessian $\mathbf{H}$. Due to the structure of the Jacobians J_{ba}, this involves adding the diagonal elements of the information matrix of each constraint, Ω'_{ba}.

Unlike the method presented by Grisetti et al. [12], each constraint is not updated iteratively, rather the calculation of Δx_i for each node in each constraint is split across threads in the GPU, with each thread calculating Δx_i for one node of one constraint. As these updates happen simultaneously in any order, they are stored in a temporary update variable in each local map. Once this has finished, the global position of each map is changed by the contents of this variable.

E. Combining Local Maps

Once a loop closing node has been identified and the graph optimised, the robot must be in an area of the environment that is has visited before. If the centre of the next local map created is within a threshold distance of either the old local map that was involved in the loop combining, its parent, or any of its children in the tree, the local map is considered for being combined. If there is a nearby older map, an ICP match with the older map and the current map is performed, using the same algorithm described in Section IV-A, with a starting transformation given by the current global displacement between to two local maps. If enough matching points are found in the last iteration of the ICP algorithm, the maps are combined. The laser points in the current map are transformed to be relative to the old match using the offset from ICP, and added to the old map. The histograms in the current map are shifted according to the change in position and averaged with the old map.

A similar process is repeated for the next local map, with its centre being compared with the centre of the old local map that was just combined, its parent and all its children. After a local map is combined with an older map, the parent of the next local map will be set to the index of the older map, and the current local map will be deleted. This continues until the robot starts traversing a new area. Combining the local maps reduces the amount of memory our SLAM system uses, and increases the efficiency of the parts of the SLAM algorithm. Additionally, in performing an ICP match during the combining process, the accuracy of the SLAM algorithm is improved as the maps are aligned to each other. The parent constraints between the old local maps can be refined with the new information from the current traversal of the area to further improve accuracy. If old local map a^1 is combined with a new map, and next next new local map is combined with old map b^1, the original constraint between a and b, $\delta_{ba}^{(1)}$, can be combined with the new constraint from the movement between the two local maps, $\delta_{ba}^{(2)}$ according to:

$$\delta_{ba} = (\Omega_{ba}^{(1)} + \Omega_{ba}^{(2)})^{-1}(\Omega_{ba}^{(1)}.\delta_{ba}^{(1)} + \Omega_{ba}^{(2)}.\delta_{ba}^{(2)}) \qquad (12)$$

The information matrix of the constraints can be added.

V. EMPIRICAL EVALUATION

To evaluate our approach, we captured several datasets from robots using Hokuyo range finders. While the algorithm can also be run with other depth sensors, such as a Microsoft Kinect, we consider range finder lasers to be more appropriate for position tracking and mapping due to their wide field of view (270°). We tested the algorithms by running the datasets at capture frame rate on a test rig consisting of a 3.4GHz Intel Core i7 CPU and a NVIDIA 570gtx GPU. This GPU has 15 streaming multiprocessors with 32 streaming processors per SM.

The ground truth datasets were captured by driving the robot around an office environment in a loop of around 80m to 120m, arriving precisely back at the starting location. The position error at the end of the loop shows the accuracy of each algorithm. Deviations were taken in the loop around the office to increase the potential for errors to accumulate. The environment consists of some walls with few features, and other walls with many occluded features. An example dataset and the route taken is shown in Fig. 5, as generated by our parallel position tracking algorithm.

TABLE I: Performance comparison of CPU and GPU

Algorithm	Error	Angular Error	Av. Iteration Time
OG-MBICP	32m	160.0°	148ms
OG-MBICP (1/4 res)	1.2m	8.0°	36ms
GPU OG-MBICP	0.69m	5.1°	3.3ms
CPU Graph Slam	0.18m	4.6°	38ms
GPU Graph Slam	0.10m	3.2°	3.4ms
GPU Map Combining	0.04m	2.7°	3.4ms

TABLE II: Performance comparison of SLAM methods

Algorithm	Error	Angular Error	Av. Iteration Time
Parallel Graph SLAM	0.11m	2.4°	3.4ms
Hector SLAM	19m	27°	2.5ms
Fast SLAM	0.2m	3.0°	165ms

In our first set of experiments, we compared the performance of CPU and GPU versions of our algorithm. The CPU tests were repeated using $1/4$ of the laser resolution as this was found to give improved results over full resolution as the faster processing time allowed more laser frames to be processed. On both the CPU and the GPU, our graph SLAM algorithm had negligible effect on the average time for an iteration, as the only processing performed every iteration is adding points to the local map. Only when a potential loop closure has been identified does it take a non-trivial time to run. In this experiment, the CPU graph SLAM algorithm took an average of 51ms to perform loop closure and optimisation, while our GPU version took an average of 6ms.

As can be seen in Table I, the accuracy of the base OG-MBICP algorithm is lower than our parallel version. This difference can be attributed to the OG-MBICP algorithm using less laser points in the match and skipping frames due to the longer processing time. While the CPU version of graph SLAM largely corrects for this, it is not as accurate as the GPU version due to larger errors being present inside each local map affecting the loop closing. Our graph SLAM algorithm with map combining further improved the accuracy to give error of just 0.04m. This represents an error of under 0.1%, and is smaller than the accuracy at which the robot can be positioned. The generated map is shown in Fig. 4.

In our second set of experiments, we compared our algorithm to other SLAM algorithms that have been successfully used in Urban Search and Rescue scenarios, such as in the RoboCup Rescue competition. In this dataset, the bottom left of the map as seen in Fig. 6 has very few points available to be used for position tracking, potentially causing the robot to become lost. Our algorithm was still able to recover and correct the loop. The slightly larger error (see Table II) and lower quality of some parts of the map, such as the left wall

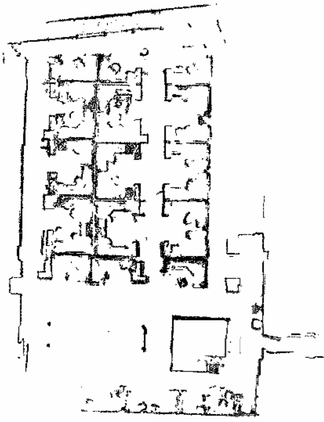

Fig. 4: Parallel Graph SLAM with a 10m laser.

Fig. 5: Parallel OG-MBICP without SLAM loop closure.

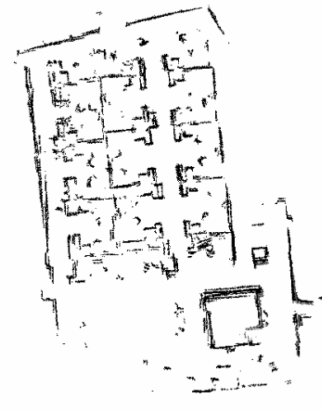

Fig. 6: Parallel Graph SLAM with a 4m laser.

and the obstacle on the bottom right are from the position tracking errors in bottom left being partially spread out over all the local maps during the map optimisation.

The Hector SLAM algorithm [5] is available open source as part of the ROS environment. During the initial part of the run, it was as accurate as our algorithm, with a 1cm error around the small loop on the bottom right. Over the larger loop, the robot's heading became incorrect when turning the corner around the top of the map, causing the large final error. Also, as the algorithm has no explicit loop closing, it sometimes was not able to recover after the robot traversed the bottom left of the map, and became lost. On average, the Hector SLAM algorithm is slightly faster than our SLAM algorithm. As Hector SLAM is also an occupancy grid ICP algorithm, it would be possible to adapt our parallel ICP method to implement it on a GPU in place of OG-MBICP.

We also compared our algorithm against a standard implementation of FastSLAM [6]. Our parallel OG-MBICP was used to provide position updates. We found 1000 particles gave the best compromise with accuracy and processing time. While this gave an small error of 0.2m, the processing time increased to an average of 165ms an iteration, making it not suitable for use on a low powered mobile robot.

VI. CONCLUSIONS

In this paper we have presented a highly efficient and accurate SLAM algorithm designed for use on mobile robots that doesn't require motion encoders. By using the output of a parallel occupancy grid ICP algorithm to build a series of local maps into a graph, our solution enables us to detect loops, optimise the graph in parallel, and refine the map when revisiting areas in real time on a mid range GPU. Our results demonstrate that our algorithm is substantially more accurate than competing solutions designed for Urban Search and Rescue situations, and is highly efficient.

REFERENCES

[1] M. Kadous, R. Sheh, and C. Sammut, "Effective user interface design for rescue robotics," in *Conference on Human-robot interaction*. ACM, 2006, pp. 250–257.

[2] Y. Chen and G. Medioni, "Object modeling by registration of multiple range images," in *ICRA*. IEEE, 1991, pp. 2724–2729.

[3] R. Shams and N. Barnes, "Speeding up mutual information computation using nvidia cuda hardware," in *Digital Image Computing Techniques and Applications*. IEEE, 2007, pp. 555–560.

[4] E. Olson, J. Leonard, and S. Teller, "Fast iterative alignment of pose graphs with poor initial estimates," in *IRCA*. IEEE, 2006.

[5] S. Kohlbrecher, O. von Stryk, J. Meyer, and U. Klingauf, "A flexible and scalable slam system with full 3d motion estimation," in *Safety, Security, and Rescue Robotics*. IEEE, 2011, pp. 155–160.

[6] A. Milstein, M. McGill, T. Wiley, R. Salleh, and C. Sammut, "A method for fast encoder-free mapping in unstructured environments," *Journal of Field Robotics*, vol. 28, no. 6, pp. 817–831, 2011.

[7] J.-S. Gutmann and K. Konolige, "Incremental mapping of large cyclic environments," in *Computational Intelligence in Robotics and Automation*. IEEE, 1999, pp. 318–325.

[8] C. Estrada, J. Neira, and J. D. Tardós, "Hierarchical slam: Real-time accurate mapping of large environments," *Transactions on Robotics*, vol. 21, no. 4, pp. 588–596, 2005.

[9] M. Bosse and R. Zlot, "Map matching and data association for large-scale two-dimensional laser scan-based slam," *The International Journal of Robotics Research*, vol. 27, no. 6, pp. 667–691, 2008.

[10] K. Granstrom, J. Callmer, F. Ramos, and J. Nieto, "Learning to detect loop closure from range data," in *ICRA*. IEEE, 2009, pp. 15–22.

[11] M. Bosse, P. Newman, J. Leonard, M. Soika, W. Feiten, and S. Teller, "An atlas framework for scalable mapping," in *ICRA*, vol. 2. IEEE, 2003, pp. 1899–1906.

[12] G. Grisetti, C. Stachniss, S. Grzonka, and W. Burgard, "A tree parameterization for efficiently computing maximum likelihood maps using gradient descent," in *RSS*, 2007.

[13] P. Agarwal, G. Tipaldi, L. Spinello, C. Stachniss, and W. Burgard, "Robust map optimization using dynamic covariance scaling," in *ICRA*, 2013.

[14] N. Sunderhauf and P. Protzel, "Towards a robust back-end for pose graph slam," in *ICRA*. IEEE, 2012, pp. 1254–1261.

[15] "3d is here: Point cloud library (pcl)," in *International Conference on Robotics and Automation*, Shanghai, China, May 9-13 2011.

[16] A. Milstein, M. McGill, T. Wiley, R. Salleh, and C. Sammut, "Occupancy voxel metric based iterative closest point for position tracking in 3d environments," in *ICRA*. IEEE, 2011, pp. 4048–4053.

[17] J. Minguez, F. Lamiraux, and L. Montesano, "Metric-based scan matching algorithms for mobile robot displacement estimation," in *ICRA*. IEEE, 2005, pp. 3557–3563.

[18] E. Olson, "Real-time correlative scan matching," in *ICRA*. IEEE, 2009, pp. 4387–4393.

[19] R. Newcombe, A. Davison, S. Izadi, P. Kohli, *et al.*, "Kinectfusion: Real-time dense surface mapping and tracking," in *Symposium on Mixed and Augmented Reality*. IEEE, 2011, pp. 127–136.

[20] B. Clipp, J. Lim, J.-M. Frahm, and M. Pollefeys, "Parallel, real-time visual slam," in *IROS*. IEEE, 2010, pp. 3961–3968.