Proceedings of Australasian Conference on Robotics and Automation, 2-4 Dec 2013, University of New South Wales, Sydney Australia

GPU Accelerated Parallel Occupancy Voxel Based ICP for Position Tracking

Adrian Ratter and **Claude Sammut**
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia
adrianr@cse.unsw.edu.au and claude@cse.unsw.edu.au

Abstract

Tracking the position of a robot in an unknown environment is an important problem in robotics. Iterative closest point algorithms using range data are commonly used for position tracking, but can be computationally intensive. We describe a highly parallel occupancy grid iterative closest point position tracking algorithm designed for use on a GPU, that uses an Extended Kalman Filter to estimate motion between scans to increase the convergence rate. By exploiting the hardware structure of GPUs to rapidly find corresponding points and by using an occupancy map structure that can be safely modified in parallel with little synchronisation, our algorithm can run substantially faster than CPU based occupancy grid position tracking. We reduce the runtime from an average of 33ms to just 3ms on a commodity GPU, allowing our algorithm to use more dense and more frequent data, resulting in improved accuracy. Additionally, our solution can use a much larger occupancy grid without significantly impacting runtime.

1 Introduction

Many areas of robotics, such as mapping and navigation, require highly accurate position update information. As motor encoder odometry can be highly unreliable in many situations, such as in environments with loose terrain [Kadous *et al.*, 2006] [Kitano and Tadokoro, 2001], scan matching techniques using range data are widely used. The majority of these algorithms use variants of the *Iterative Closest Point* (ICP) algorithm [Chen and Medioni, 1991]. ICP is an iterative algorithm which works by finding corresponding points between laser scans, and minimising the distance between the scan points and their matching correspondent points. This is repeated until scans converge, that is, the error between the scan points and their corresponding points falls below a certain threshold.

Several ICP variants use occupancy grids [Milstein *et al.*, 2011] [Kohlbrecher *et al.*, 2011] to greatly reduce the cost of searching for the corresponding points. Occupancy grids also enable the current scan to be aligned against a history of past observations, reducing the possibility that errors in a single scan will cause the algorithm to fail. While occupancy grids improve the efficiency of ICP to the point where it can run on mobile robots with limited processing power, they still consume a large amount of the processing time available, leaving little CPU time for other tasks, and scale poorly with the density of scan points.

Over the last several years, GPUs have started to be used for robotics applications, with their high data-throughput and data-parallelism being used to speed up algorithms. However, they have a significantly different programming, memory and synchronisation model to CPUs, with performance greatly reducing with branching threads, random memory access and data-access synchronisation [Shams and Barnes, 2007]. These characteristics mean algorithms have to be substantially adapted to run efficiently on a GPU. In particular, the limited synchronisation and highly parallel nature of GPUs mean that data structures have to be carefully designed to avoid data corruption.

The large amount of interaction between different calculations and the need for hundreds of threads to add new scans to the occupancy grid in parallel without causing data corruption make it difficult to map occupancy grid ICP algorithms to GPUs. In this paper we present a highly efficient occupancy grid based ICP algorithm that takes full advantage of the structure of GPU hardware. In particular, we propose two novel approaches that fully exploit the high data parallelism of GPUs without causing data corruption:

- Storing the map as a lookup table into an active cells list. This enables efficient shifting of the map as the robot moves, while retaining the advantages

of occupancy grids for quickly finding corresponding points. This structure enables us to make parallel modifications to the same cells in the occupancy grid without corrupting data.

- A synchronisation free local search of the occupancy grid to find corresponding points by exploiting the Single Instruction, Multiple Data (SIMD) structure of GPUs.

To demonstrate our approach, we used our algorithm to adapt a successful occupancy grid based ICP algorithm, *Occupancy Grid Metric Based Iterative Closest Point* (OG-MBICP) [Milstein *et al.*, 2011], to a GPU. OG-MBICP aligns scans to a history of past observations by using a metric for finding corresponding points in the occupancy grid that obtains both linear and angular displacements in a single pass. In using our algorithm to adapt OG-MBICP to a GPU, we found the runtime reduced from an average of 33ms an iteration to just 3ms. The accuracy of the algorithm increased due to reducing the number of skipped frames. We also show that our algorithm scales well with map size. It is possible to use our algorithm to adapt other approaches, such as by Kohlbrecher et al. [2011], for efficient use on a GPU.

We also present a novel way to extract the 2D gradient of the map around the scan points from our parallel occupancy grid structure, which can be used to calculate a Hessian matrix for a scan match and thereby the observation covariance for an Extended Kalman Filter (EKF) update. We use the filter as an initial guess for the motion between scans, and show that this increases the convergence speed of our algorithm.

2 Related Work

Recently, GPUs have started to be used for robotics applications, with Olson [2009] and Newcombe et al. [2011] presenting the first mappings of ICP algorithms onto a GPU. The Kinect Fusion algorithm [Newcombe *et al.*, 2011] uses ICP to align successive 3D scans to generate a surface map of a small area. While the ICP alignment in this algorithm does mean position tracking information can be extracted, the surface reconstruction aim of the alignment introduces significant computational complexities that are not needed for position tracking and requires substantial overlap of each frame. This work was extended to work over a larger range by Whelan et al. [2012], but no evaluation of the accuracy was presented.

One of the most popular methods of registering point clouds for position tracking is ICP [Chen and Medioni, 1991][Lu and Milios, 1994]. In their survey of popular variants of ICP, Rusinkiewicz and Levoy [2001] found that a common source of error occurred because the search for point correspondences in the ICP variants does not consider that rotations of the sensor can cause points distant from the sensor to be far away from their correspondent point. The MBICP algorithm, introduced by Minguez et al. [2005] overcomes this issue by using a metric for finding correspondences that takes into account rotational and translational movement of the sensor in a single pass. This was found to increase the accuracy and robustness of the base ICP algorithm. A 3D extension to the MBICP metric was proposed by Armesto et al. [2010], but requires wheel odometry to build 3D scans when using a laser range finder.

A drawback of many ICP algorithms is the computationally intensive search for point correspondences. Montesano et al. [2005] used Mahalanobis distance to calculate a subset of previous points that are statistically compatible with each laser point to narrow the correspondence search. Olson [2009] employed an exhaustive sampling and correlation approach, whereas Diosi and Kleeman [2007] used a polar coordinate system to avoid an extensive correspondence search. The search can also be avoided by storing previous scans in an occupancy grid [Milstein *et al.*, 2011] [Kohlbrecher *et al.*, 2011]. ICP algorithms, such as those in the Point Cloud Library [Rusu and Cousins, 2011], commonly align the current scan to only the previous scan, thereby allowing errors in a single scan to cause to algorithm to fail. Occupancy grid based approaches can overcome this by storing a history of past scans, and can take into account the number of observations of a cell and time since it was last observed. Storing a history of past scans in an occupancy grid also enables a single 2D scan from a tilting laser to be aligned to a 3D grid. The OG-MBICP algorithm is notable for storing laser points in the occupancy grid, allowing laser scans to be aligned to a resolution much greater than the size of occupancy grid cells.

Efforts have also been made to decrease the convergence time required for each iteration of the ICP algorithm. Censi [2008] proposed a special point to line metric that was found to significantly improve the convergence speed, but isn't robust to initial alignment errors. An alternate way achieve this is to guess a likely displacement at the start of the ICP alignment. Milstein et al. [2011] used the previous update from the ICP algorithm as the initial guess, whereas Kohlbrecher et al. [2011] used the pose prediction from an EKF as the initial guess, and used the map gradient around each matching laser point as the key part in the calculation of the observation covariance. The map gradient was calculated by examining the neighbouring four pixels in an occupancy grid to the aligned laser point. Previous laser points added to the occupancy grid with small errors in their alignment can result in the adjacent grid cells are all occupied, thereby obscuring the gradient.

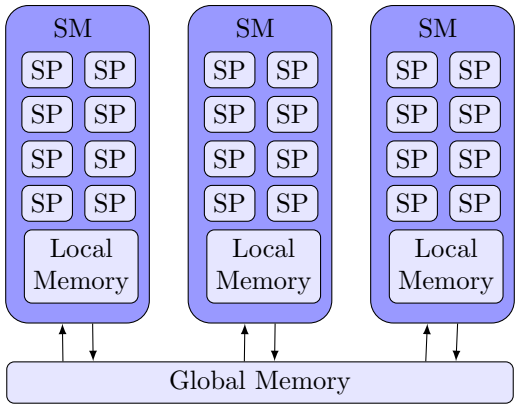

Figure 1: The hardware structure of a GPU.

3 Background

3.1 GPU Programming with OpenCL

GPUs are based on 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. Each SP has its own set of registers to store per thread data, but 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 [Nickolls *et al.*, 2008]. 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, distinctly unlike multi-threaded CPU programs.

GPUs have two major levels of shared memory: local and global. As local memory of a SM resides on-chip, it operates at full clock speed, and in some ways approximates the function of CPU caches. Global memory is significantly slower to access (around 200-300 clock cycles per access [Ryoo *et al.*, 2008]), 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. This means that minimising uncoalesced (random) memory accesses is one of the most important criteria for efficient execution of GPU algorithms [NVIDIA, 2009]. The memory and processor model is shown in Fig. 1.

Open Computing Language (OpenCL) [A. Munshi, 2008] is a C based language designed to be used for parallel computing on devices such as GPUs. An OpenCL program consists of one or more computation *kernels* (functions that operate on the GPU) and CPU based functions for queuing of kernels and copying of data to and from GPU global memory. Threads, or *work items*, execute the kernel code, normally using different data, and are grouped into a series of *work groups* [Munshi *et al.*, 2011]. Each work group in a kernel resides on a single SM, and work groups can run in any order. Therefore, threads have no mechanism to synchronise with threads in other work groups, but standard barrier synchronisation operations are available for threads inside a work group.

As each SP in a SM executes the same instructions at the same time, a greater level of synchronisation is available to threads in a work group that are active at the same time. These sub-groups of threads, known as a *warp*, are guaranteed by the hardware structure to be at the same position in the execution of the kernel as each other. On NVIDIA GPUs, each warp is 32 threads [NVIDIA, 2009]. GPUs also provide atomic operations to local and global memory to prevent data corruption.

3.2 MBICP

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 [Lu and Milios, 1994]. As the pitch and roll of the robot can be determined if the robot is equipped with an attitude sensor, such as an Inertial Measurement Unit, we define an alignment in three dimensions to be $q = (x, y, z, \theta)$. 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.

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

MBICP uses a metric to allow the translation and orientation of successive scans to be determined in one pass [Minguez *et al.*, 2005]. The metric defines a positive constant, L, that controls the relative importance of angular displacement against linear displacement. While Minguez et al. [2005] presented their MBICP method in two dimensions, aligning to $q = (x, y, \theta)$, it can be easily extended to work in three dimensions, with the 3D metric given by:

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

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

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

where:

$$q(p_1) = \begin{pmatrix} x + p_{1x}\cos\theta - p_{1y}\sin\theta \\ y + p_{1x}\sin\theta + p_{1y}\cos\theta \\ z + p_{1z} \end{pmatrix} \quad (4)$$

$$q(p_1) \approx \begin{pmatrix} x + p_{1x} - p_{1y}\theta \\ y + p_{1x}\theta + p_{1y} \\ z + p_{1z} \end{pmatrix} \quad (5)$$

As each iteration only produces a small q, (eq. 4) can be approximated by the linearisation shown in (eq. 5). By setting this result to equal p_2 and substituting into (eq. 2), a quadratic with a unique minimum is obtained. Applying this minimum for $\|q\|$ into (eq. 3) gives:

$$d(p_1, p_2) \approx \sqrt{\frac{\delta_x^2 + \delta_y^2 + \delta_z^2 - (\delta_y p_{1z} - \delta_z p_{1y})^2 + (\delta_z p_{1x} - \delta_x p_{1z})^2 + (\delta_x p_{1y} - \delta_y p_{1x})^2}{p_{1x}^2 + p_{1y}^2 + p_{1z}^2 + L^2}} \quad (6)$$

where $\delta = (p_2 - p_1)$. Using this result in the original ICP minimisation, (eq. 1), with $p_1 = p_i$ and $p_2 = q(p_i')$, and using the approximation for q in (eq. 5), we get:

$$E_{dist}(q) = \sum_{i=1}^{n} \left(\frac{\delta_{ix}^2 + \delta_{iy}^2 + \delta_{iz}^2 - (\delta_{iy} p_{iz} - \delta_{iz} p_{iy})^2 + (\delta_{iz} p_{ix} - \delta_{ix} p_{iz})^2 + (\delta_{ix} p_{iy} - \delta_{iy} p_{ix})^2}{p_{ix}^2 + p_{iy}^2 + p_{iz}^2 + L^2} \right) \quad (7)$$

$$\begin{aligned} \delta_{ix} &= p'_{ix} - \theta p'_{iy} + x - p_{ix} \\ \delta_{iy} &= \theta p'_{ix} + p'_{iy} + y - p_{iy} \\ \delta_{iz} &= p'_{iz} + z - p_{iz} \end{aligned} \quad (8)$$

$E_{dist}(q)$ can be rewritten as a quadratic in terms of a symmetric matrix A, a vector b and a constant c. The value of q that minimises $E_{dist}(q)$ can therefore be calculated by finding the minimum of the quadratic.

$$E_{dist}(q) = q^T A q + 2b^T q + c \quad (9)$$

$$q_{min} = -A^{-1} b \quad (10)$$

With these equations, the corresponding point for each new point p_i' can be found by searching the existing points for the one that minimises (eq. 6). Once all the corresponding points are found, q_{min} can be found by solving (eq. 10). The set of new points is then transformed by q and the process repeated until q converges.

3.3 OG-MBICP

The OG-MBICP algorithm [Milstein *et al.*, 2011] stores previously observed points inside an occupancy grid, so that the process of finding corresponding points is simplified by 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. This enables a history of past scans to be used for alignment, making the algorithm more robust to temporary occlusion and mobile objects. Additional information, such as the number of times a cell has been observed and the time since it was last observed can be used to determine if a cell should be made inactive.

This extra information can also be used to influence the effect each match has on the overall alignment, through performing:

- Match weighting: An existing reference point in the grid is considered more reliable if it has been observed many times. The ICP minimisation formula can be changed to:

$$E_{dist}(q) = \sum_{i=1}^{n} f(p_i) d(p_i, q(p_i'))^2 \quad (11)$$

  where $f(p_i)$ is the number of times a cell has been observed divided by the maximum number of observations permitted. The same idea can be used when searching for matching points, with frequently observed points being considered closer. This can be achieved by modifying the distance formula from (eq. 6) to be $\frac{d(p_1,p_2)}{f(p_1)}$.
- Distance weighting: Objects further away will inherently be observed less than close objects, yet are no less informative. To boost the influence of distant objects in the match, the distance calculation for each point p_i in (eq. 1) is multiplied by $\|p_i\|/max\_dist$, where max_dist is the maximum sensor reading possible.
- Variable L: L determines the relative influence of translation and angular components in the match. L is redefined as a function in p_i because objects further away tend have more accurate angular readings, while close objects tend to be more appropriate for determining translation. This gives distant points greater influence on rotation:

$$L(p_i) = 1 - \frac{\|p_i\|}{max\_dist + \varepsilon} \quad (12)$$

  with $\varepsilon > 0$ to prevent $L = 0$ for all points.

4 Approach

In this section we describe the design of our occupancy grid based ICP algorithm and the features that make it highly efficient when deployed on a GPU. Using the base OG-MBICP algorithm, we devised a design that allows a high level of parallelism, maximises coalesced memory operations, exploits inherent warp level synchronisation, and with a flexible map structure that can be expanded with little impact on run time.

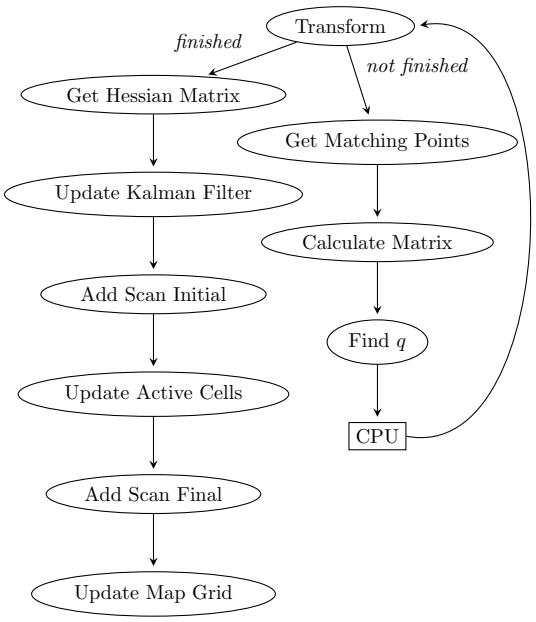

Figure 2: The structure of our parallel occupancy grid ICP algorithm.

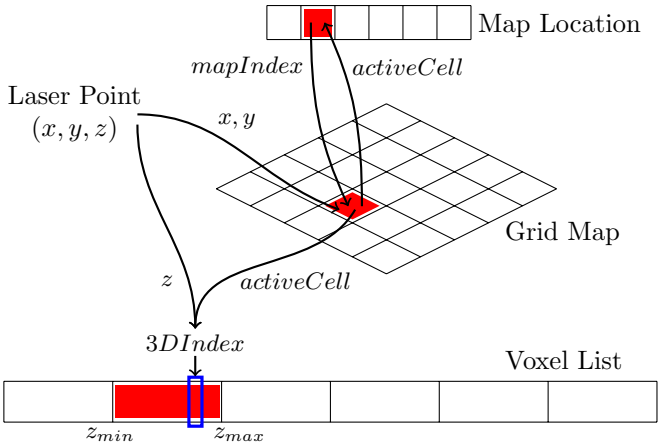

Figure 3: The data structures used. *Map Location* and *Voxel List* arrays store the information about each active cell, while *Grid Map* stores stores the index into these arrays for each active cell.

Fig. 2 shows the overall design of our algorithm, where each ellipse represents a separate OpenCL kernel. As there is no method of global synchronisation available on a GPU, multiple kernels, run sequentially, can be used to achieve global synchronisation points. The algorithm can be split into two major parts—finding an alignment, q (the right column of Fig. 2), and updating the map (the left column). When a new set of points from the laser is received, they are transferred to the GPU, and the kernels involved in finding q are queued. Once an alignment has been found, the values of q are copied to the CPU, and the CPU determines if q has converged. If q hasn't converged, the kernels to transform the laser points by q and find a new q are again queued. Once q converges, the kernels estimate the observation covariance, and run q through an EKF. The laser points are then transformed according to the EKF update and added to the map, along with the gradient of the scan around each laser point.

In either case, the first kernel queued is the `transform` kernel, which transforms each laser point by the change in q from the previous iteration. In the case of the first iteration of the algorithm for a given set of laser points, the pose estimate from the EKF is used as the initial transformation. This initial estimate of the displacement between scans can significantly reduce the number of iterations needed for q to converge. In this kernel, each thread is allocated a single laser point to transform, with adjacent threads allocated adjacent laser points to ensure coalesced memory access.

4.1 Map Structure and Updates

One of the major challenges in the design of the occupancy grid structure is ensuring the data remains correct with thousands of threads reading from it and writing to it at the same time, while retaining fast access even when a large map is used. Additionally, as the robot is always located at the center of the map, existing cells have to be shifted in the map efficiently as the robot moves.

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 (voxels) for the z dimension, and each sub-cell stores points that were previously observed in it. This structure is shown in Fig. 3. 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. Information about active cells, such as the observation count, are stored as arrays containing that 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 grid map is shifted by the alignment so the robot remains centered in the grid, the new points are added to the map, and active cells that haven't been observed recently are deleted. The kernels used to do this are in the left column of Fig. 2 and their pseudocode is shown in Algorithm 1. Each kernel will execute once for each laser point or active cell.

The kernels `update active cells` and `update map grid` are responsible for shifting the grid map. This is split over two kernels to avoid concurrency issues. In

Algorithm 1 Pseudocode of map update process

```
1: function ADD SCAN INITIAL(laserPoint)
2:     mapIndx ← getMapIndex(laserPoint)
3:     if gridMap[mapIndx] = EMPTY then
4:         res ← atomicAdd(gridMap[mapIndx], BIG_NUM)
5:         if res = EMPTY then
6:             emptyCell ← atomicInc(nxtEmptyCell)
7:             activeCell ← emptyCells[emptyCell]
8:             mapLocation[activeCell] ← mapIndx
9:         end if
10:        status[laserPoint] = NUM_CELLS + mapIndx
11:    else if gridMap[mapIndx] > NUM_CELLS then
12:        status[laserPoint] = NUM_CELLS + mapIndx
13:    else
14:        status[laserPoint] = gridMap[mapIndx]
15:    end if
16: end function
17: function UPDATE ACTIVE CELLS(activeCell)
18:    if isActive(activeCell) then
19:        indx ← mapLocation[activeCell]
20:        mapVal ← gridMap[indx]
21:        shift(mapLocation[activeCell])
22:        gridMap[indx] ← EMPTY
23:        if mapVal > NUM_CELLS then
24:            gridMap[indx] ← NUM_CELLS + activeCell
25:        else if isOld(activeCell) then
26:            free ← atomicDec(nxtEmptyCell) − 1
27:            emptyCells[free] ← activeCell
28:        end if
29:    end if
30: end function
31: function ADD SCAN FINAL(laserPoint)
32:    activeCell ← status[laserPoint]
33:    if activeCell > NUM_CELLS then
34:        val ← activeCell − NUM_CELLS
35:        activeCell ← gridMap[val] − NUM_CELLS
36:    end if
37:    3DIndex ← get3DIndex(laserPoint, activeCell)
38:    addPoint(laserPoint, 3DIndex)
39: end function
40: function UPDATE MAP GRID(activeCell)
41:    if isActive(activeCell) then
42:        index ← mapLocation[activeCell]
43:        gridMap[index] ← activeCell
44:    end if
45: end function
46: function TRANSFORM(laserPoint)
47:    if isInitialTransform() then
48:        if status[laserPoint] > NUM_CELLS then
49:            index ← status[laserPoint] − NUM_CELLS
50:            if gridMap[index] > NUM_CELLS then
51:                gridMap[index] ← EMPTY
52:            end if
53:        end if
54:    end if
55:    //Transform the laserPoint
56: end function
```

the `update active cells` kernel, lines 19—22 reset the references to the active cells in the grid map and shift each cell, with the *mapLocation* array storing the index into the grid map for each active cell. The `update map grid` kernel inserts references to each active cell in the new location in the grid map.

If a new laser point being added to the map is inside a currently active cell, the active cell index is stored in the temporary storage array *status* on line 14 by reading the value from the grid map, and lines 37—38 store the laser point information and update the active cell.

Much of the difficulty in updating the map occurs when the new laser point is not within a currently active cell, as it is possible for several threads running at the same time to want to create a new active cell in the same location. Empty active cell indexes are stored in a stack called *emptyCells*. The process of creating a new active cell is as follows:

1. `Add scan initial`: If the grid map at the laser point is empty (line 3), a value larger than the number of active cells is added atomically (line 4). The thread that performs the first add (atomic adds return the value before the add) can then create a new active cell (lines 5—9). If line 12 is reached, another thread must have already created a new cell. In either case, the *status* array stores the index of the laser point in the grid map, offset by the maximum number of active cells.
2. `Update active cells`: If the entry in the grid map is a new active cell (line 23), the grid map entry, instead of being reset, is set to the active cell, offset by the maximum number of active cells (line 24).
3. `Add scan final`: If the *status* buffer indicates the laser point is being added to a new active cell (line 33), the index of the active cell is extracted from the grid map (lines 34—36) so that laser point information can be added to the active cell.
4. `Transform`: In the initial call to transform for a new iteration, some entries in the grid map may not be properly reset if they were used for the creation of a new active cell in the previous iteration. Lines 48—53 ensure that the grid map has been reset correctly.

4.2 Finding an Alignment

The right hand column of Fig. 2 shows the kernels involved in finding an alignment q. The major step of this process is finding the point stored in the map that minimises (eq. 6) for each new point. This is by far the most computationally expensive part of OG-MBICP. 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. 6) for the center 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 because all threads in the warp are automatically synchronised, so data written by one thread can be safely read by another thread in the next instruction. Finally, the threads in the warp calculate d for each of the points stored in the closest sub-cells, and the best match can then be chosen. As the coarse to fine strategy to find each matching point is split over many threads and requires no synchronisation, finding all the matching points is extremely efficient, taking an average of only $41\mu s$ for laser scans containing over 300 points.

Once all matching points have been found in the `get matching points` kernel, the `calculate matrix` kernel calculates each of the elements of A and b needed to find q_{min}. Each thread calculates the contribution of a matching point to each of the elements of A and b. Once this is finished, each warp then finds the sum of the contributions from its threads, and then each work group calculates the sum of each warp for each element. Finally, these partial sums are added to the global value of each element. As each sum is a floating point value, and GPUs, like most hardware, don't allow atomic floating point adds, a series of atomic swaps have to be performed to ensure no data corruption in the add, using the following algorithm:

```
safeFloatAdd(data, address):
   while data is not 0:
      data = atomicSwap(address, data+
                  atomicSwap(address, 0))
```

Finally, the `calculate matrix` kernel calculates (eq. 10).

4.3 Extended Kalman Filter

The EKF stage of the algorithm is performed over two kernels, `get hessian matrix`, which calculates the Hessian matrix for the scan match, which is used for the calculation of the observation covariance, and `update kalman filter`, which performs the EKF update. After the update has been performed, the laser points are transformed by the filtered q at the start of `add scan initial` and then added to the map.

The Hessian matrix of the scan match [Kohlbrecher *et al.*, 2011] is given by:

$$H = \frac{1}{N}\sum_{i=1}^{n}\left(\nabla M\left(q(p_i)\right)\frac{\partial q(p_i)}{\partial q}\right)^T\left(\nabla M\left(q(p_i)\right)\frac{\partial q(p_i)}{\partial q}\right) \tag{13}$$

Each thread in the `get hessian matrix` kernel calculates the contribution of a single laser point to the Hessian matrix. In this equation, $\nabla M\left(q(p_i)\right)$ is the gradient of the map near the transformed laser point. To calculate this, we use the gradient of the corresponding point to the laser point that was found in the last iteration of the alignment algorithm. The gradient of the corresponding point is the gradient of the laser scan that contained the point around the location of the point. Finally, using (eq. 4), $\nabla M\left(q(p_i)\right)$ is given as:

$$\frac{\partial q(p_i)}{\partial q} = \begin{pmatrix} 1 & 0 & -\sin(\theta)p_{i,x} - \cos(\theta)p_{i,y} \\ 0 & 1 & \cos(\theta)p_{i,x} - \sin(\theta)p_{i,y} \end{pmatrix} \tag{14}$$

The `update kalman filter` kernel uses the Hessian matrix to estimate the observation covariance for the EKF. While much of this update cannot be performed in parallel, the small number of calculations required for the update means that it is not worth the data copy overheads to perform this on the CPU. The Hessian Matrix can be transformed into the observation covariance estimate needed by the Kalman Filter according to:

$$Q = \gamma^2 H^{-1} \tag{15}$$

where γ is a scaling factor dependent on the scanning device. We use a constant velocity model to estimate the motion update, with the process covariance set to:

$$R = \begin{pmatrix} k_x\overline{x} & 0 & 0 \\ 0 & k_y\overline{y} & 0 \\ 0 & 0 & k_\theta\overline{\theta} \end{pmatrix} \tag{16}$$

In this matrix, $\overline{x}$, $\overline{y}$ and $\overline{\theta}$ are the motion update estimates, and k_x, k_y and k_θ are the error parameters modelling the Gaussian distribution of the robot's likely motion. As the robot's vertical motion is much less than its horizontal motion, we exclude modelling z in the EKF. In practice, we set minimum levels for each of $\overline{x}$, $\overline{y}$ and $\overline{\theta}$ to model the motion as the robot starts moving.

5 Empirical Evaluation

To evaluate our approach, we used our algorithm to adapt OG-MBICP for use on a GPU, creating *Parallel Occupancy Grid Metric Based Iterative Closest Point* (POG-MBICP). We captured several datasets from an auto levelled Hokuyo UTM-30LX laser, running at 40Hz and generating around 1100 laser points per scan, mounted on a four wheel drive robot. This robot is intended for use in disaster environments. It is designed to traverse rough terrain with debris, in which motor encoders are completely unreliable. While the algorithm can also be run using other depth sensors, such as a Microsoft Kinect, we consider the range finder laser to be more suitable for position tracking because of its very

Table 1: Performance comparison of move prediction strategies.

Strategy	Av. Runs	Av. Time	Error	Angular Error
No prediction	16.27	5.61ms	0.78m	6.9°
Last move	9.42	3.33ms	0.69m	5.1°
Adjacent cell KF	9.07	3.15ms	0.49m	4.8°
Gradient KF	8.83	3.07ms	0.25m	3.3°

Table 2: Performance comparison using a 10m by 10m local map, where the OG-MBICP algorithm was tested using different fractions of the full resolution scan.

Algorithm	Fraction	Average Time	Error	Angular Error
OG-MBICP	1	148ms	32m	160.0°
OG-MBICP	1/4	36ms	1.2m	8.0°
OG-MBICP	1/8	31ms	1.8m	8.6°
POG-MBICP	1	3.3ms	0.69m	5.1°

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 approximately 105m, arriving precisely back at the starting location. The position error at the end of the loop shows the accuracy of each algorithm. Many 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. Additionally, the datasets include people walking around the office. In previous work, Milstein et al. [2011] found that the MBICP algorithm produced highly inaccurate results in this type of environment because small alignment errors accumulate.

In our first set of experiments, we compared strategies to obtain a starting estimate for the optimisation process to see the effect they have on the convergence speed of our algorithm. We used a small $10m$ by $10m$ grid, with a cell size of $5cm^3$, as this was found by Milstein et al. [2011] to work the best for OG-MBICP. This grid is large enough to contain local features, but small enough to keep the processing time as small as possible. In each test we measured the average number of iterations of the POG-MBICP algorithm needed for the measurement from a single scan to converge, the average *total* processing time for a single scan, and the final position and angular error.

The results from these experiments are shown in Table 1, where the different strategies are:

- *No prediction:* No initial motion guess is made.
- *Last move:* The last update of the POG-MBICP algorithm is used as the estimate.
- *Adjacent cell KF:* The pose estimate from an EKF is used as the estimate. The Kalman Filter uses the observation covariance approach by Kohlbrecher et al. [2011], which extracts the map gradient from the occupancy status of cells in the map adjacent to each laser scan point.
- *Gradient KF:* Our EKF estimation approach.

Table 3: Performance comparison with different map sizes.

Algorithm	Map	Average Time	Error	Angular Error
OG-MBICP	10m	36ms	1.2m	8.0°
OG-MBICP	60m	49ms	22m	80.2°
POG-MBICP	10m	3.3ms	0.69m	5.1°
POG-MBICP	60m	2.9ms	0.03m	0.6°

As can be expected, making no initial estimate resulted in a significantly higher number of iterations before the algorithm converged, and therefore a longer processing time. The EKF approach presented in this paper had a faster convergence speed by a small, but noticeable amount than the other prediction strategies. This difference can be attributed to our method's use of the gradient around each corresponding point giving a more accurate measurement of the map gradient than the occupancy status of the nearby grid cells, thereby resulting in a more informative observation covariance matrix. Interestingly, our method produced a smaller position error than the other strategies, presumably due to the more accurate starting estimates resulting in the ICP algorithm getting stuck in fewer local minimums.

For our second set of experiments, we compared to the performance of the OG-MBICP to our POG-MBICP algorithm, again using a 10m by 10m grid. As OG-MBICP used the previous update as the starting estimate for the next iteration, experiments were made using this strategy for both algorithms. We found that the performance of the OG-MBICP algorithm depended greatly on the number of laser points processed, as using the full laser resolution increased the run time to cause many frames to be skipped, thereby increasing the error.

Table 2 shows a comparison of POG-MBICP at full resolution, to OG-MBICP at full resolution, 1/4 resolution and 1/8 resolution. With an average runtime of 148ms when using the full laser resolution, enough frames were skipped to cause the robot to become completely lost. Fig. 4 shows the map generated. Processing the laser scans at full resolution however proved no problem for POG-MBICP, requiring an average of only 3ms to produce an alignment, 45 times faster than OG-

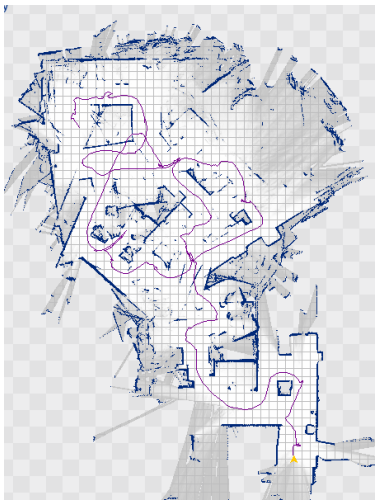

Figure 4: OG-MBICP using all laser points.

Figure 5: OG-MBICP with 1/4 laser points.

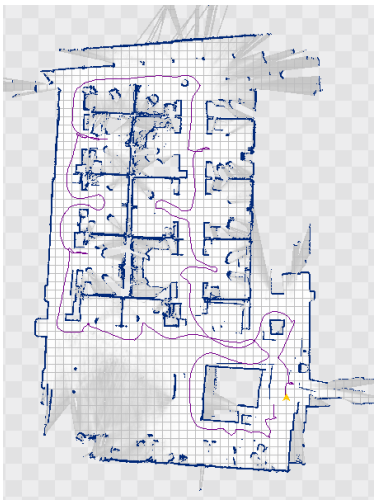

Figure 6: POG-MBICP with all laser points.

MBICP, whilst accumulating a distance error of under 1%. The map generated in the run is shown in Fig. 6. The best performance of OG-MBICP came when 1/4 resolution was used, and the map generated shown in Fig. 5. In this case, the accuracy came close to POG-MBICP, with the slightly lower accuracy due to the lower resolution being used and the average processing time still being slightly over the full frame rate; 11 times slower than POG-MBICP.

Our third set of experiments consisted of using a larger map size of $60m$ by $60m$. This map size was chosen as it is large enough to store the entire office, so features seen towards the end of the loop can be matched from when they were seen at the beginning. A larger map size also enables the 30m maximum range of the laser to be used, so objects further than 5m away from the robot can be used in the alignment process. Table 3 shows the results. We ran the OG-MBICP algorithm at 1/4 laser scan resolution as this gave the best results. The increased processing time from larger map size caused the OG-MBICP algorithm to become lost. However, the larger map size didn't significantly affect the our POG-MBICP algorithm, with the slightly lower average time a result of less iterations being needed on average for each alignment. Less iterations were needed as the inclusion of more distant points in the alignment process increased the convergence speed. As features recorded at the start of the run could be used for alignment at the end, POG-MBICP was able to correct for any slight positioning errors, giving a final error of 3cm, which is less than the accuracy at which the robot can be manually positioned.

An alternate dataset was recorded during the 2013 RoboCup Rescue competition using the auto levelled Hokuyo laser attached to a motor that constantly altered its pitch to allow capturing of 3D data. A tilting laser enables the alignment to be corrected using objects at different heights, but means parts of scans that provide no information, such as floor or ceiling hits have to be discarded. Due to the environment, we were unable to return to the exact starting point, but the accuracy can be evaluated by observing any divergence in walls viewed from opposite sides at different stages in the recording. The map produced by our algorithm can be seen in Fig. 7. It can be seen that the position tracking remained accurate despite the environment consisting of rough terrain including gravel traps and mismatched pitch and roll ramps that cause the robot to suddenly drop.

Figure 7: Map from 3D dataset, projected onto a plane.

6 Conclusions

This paper has presented a parallel position tracking algorithm suitable for use on a GPU, with a flexible occupancy grid data structure that can be read and modified safely in parallel, a highly parallel method of finding the corresponding points for ICP, and an EKF based initial

estimation strategy. As a result our algorithm allows for highly accurate and efficient position tracking.

Our results demonstrate when run on a mid range GPU, our algorithm is able to perform at least 10 times faster than OG-MBICP, even when using a greater density of scan data and a larger map. The use of larger maps allows longer range input to be used and for more data to be stored in the occupancy grid to assist with alignment. In reducing the run time of position tracking to 3ms, our future work is to combine POG-MBICP with an on-board GPU-based SLAM algorithm suitable when internal odometry is unavailable or inaccurate.

References

[A. Munshi, 2008] Ed A. Munshi. The OpenCL specification. *Khronos OpenCL Working Group*, 2008.

[Armesto *et al.*, 2010] L. Armesto, J. Minguez, and L. Montesano. A generalization of the metric-based iterative closest point technique for 3D scan matching. In *Int. Conference on Robotics and Automation*, pages 1367–1372. IEEE, 2010.

[Censi, 2008] A. Censi. An ICP variant using a point-to-line metric. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 19–25. IEEE, 2008.

[Chen and Medioni, 1991] Y. Chen and G. Medioni. Object modeling by registration of multiple range images. In *International Conference on Robotics and Automation*, pages 2724–2729. IEEE, 1991.

[Diosi and Kleeman, 2007] A. Diosi and L. Kleeman. Fast laser scan matching using polar coordinates. *The International Journal of Robotics Research*, 26(10):1125–1153, 2007.

[Kadous *et al.*, 2006] M. Kadous, R. Sheh, and C. Sammut. Effective user interface design for rescue robotics. In *Conference on Human-robot interaction*, pages 250–257. ACM, 2006.

[Kitano and Tadokoro, 2001] H. Kitano and S. Tadokoro. Robocup rescue: A grand challenge for multi-agent and intelligent systems. *AI Magazine*, 22(1):39, 2001.

[Kohlbrecher *et al.*, 2011] S. Kohlbrecher, O. von Stryk, J. Meyer, and U. Klingauf. A flexible and scalable slam system with full 3D motion estimation. In *International Symposium on Safety, Security, and Rescue Robotics*, pages 155–160. IEEE, 2011.

[Lu and Milios, 1994] F. Lu and EE Milios. Robot pose estimation in unknown environments by matching 2d range scans. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 935–938. IEEE, 1994.

[Milstein *et al.*, 2011] 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*, pages 4048–4053. IEEE, 2011.

[Minguez *et al.*, 2005] J. Minguez, F. Lamiraux, and L. Montesano. Metric-based scan matching algorithms for mobile robot displacement estimation. In *Int. Conference on Robotics and Automation*, pages 3557–3563. IEEE, 2005.

[Montesano *et al.*, 2005] L. Montesano, J. Minguez, and L. Montano. Probabilistic scan matching for motion estimation in unstructured environments. In *International Conference on Intelligent Robots and Systems*, pages 3499–3504. IEEE, 2005.

[Munshi *et al.*, 2011] A. Munshi, B. Gaster, and T.G. Mattson. *OpenCL programming guide*. Addison-Wesley Professional, 2011.

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

[Nickolls *et al.*, 2008] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[NVIDIA, 2009] NVIDIA. NVIDIA OpenCL best practices guide. 2009.

[Olson, 2009] E.B. Olson. Real-time correlative scan matching. In *ICRA*, pages 4387–4393. IEEE, 2009.

[Rusinkiewicz and Levoy, 2001] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. In *3-D Digital Imaging and Modeling*, pages 145–152. IEEE, 2001.

[Rusu and Cousins, 2011] R. Rusu and S. Cousins. 3D is here: Point cloud library (PCL). In *ICRA*, Shanghai, China, May 9-13 2011.

[Ryoo *et al.*, 2008] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, and D. Kirk. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[Shams and Barnes, 2007] R. Shams and N. Barnes. Speeding up mutual information computation using NVIDIA CUDA hardware. In *Digital Image Computing Techniques and Applications*, pages 555–560. IEEE, 2007.

[Whelan *et al.*, 2012] Thomas Whelan, Michael Kaess, Maurice Fallon, Hordur Johannsson, John Leonard, and John McDonald. Kintinuous: Spatially extended kinectfusion. 2012.