Scalable Indoor Pedestrian Localisation using Inertial Sensing and Parallel Particle Filters

Agata Brajdic, Robert Harle
Computer Laboratory
University of Cambridge
Cambridge, UK
{agata.brajdic,robert.harle}@cl.cam.ac.uk

***Abstract*—Location-aware computing is a fast emerging area in mobile computing. A plethora of approaches to indoor localisation have been demonstrated, but almost all rely on extensive infrastructure. A popular alternative is to use dead reckoning to track inertial sensors. However, sensor drift must be addressed by incorporating external constraints such as the building layout. This dictates the use of computationally expensive particle filters that hinder scalability, especially during localisation phases where the system does not have any estimate of where the user is within a building.**

In this paper, we address the scalability problem by exploiting the latent parallelism in the algorithm and adapting it for execution on commodity Graphical Processing Units (GPUs). We describe how to parallelise the particle filter and evaluate different filter architectures. We find that our GPU implementation can iterate 8.8 times faster than the fastest CPU variant.

We also show how to handle multiple filters using a novel memory paging scheme and an adaptable particle number. We find that between 17 and 101 users can be localised *in real-time* using only a mid-range GPU installed in a standard desktop machine, compared with at most one using a previous sequential approach.

***Index Terms*—Inertial tracking, Particle filters, Localisation, Parallel processing.**

I. Introduction

The recent trend towards mobile computing has sparked an interest in location-aware applications. These have been very successful outdoors, where GPS provides global position fixes, but there is a conspicuous lack of indoor location systems. Despite many demonstrated indoor location solutions, the characteristic requirement for precisely-deployed infrastructure has so far precluded ubiquitous deployment.

One promising approach is the use of inertial sensors to perform infrastructure-less positioning, or dead-reckoning. The recent successes in miniaturising such sensors has resulted in a wide deployment of potential sensors. However, it has also amplified the typical inertial sensor noise characteristics, resulting in fast accumulation of drift (positioning error caused by integrating the inertial measurements). It is not unusual to observe errors in the hundreds of metres after only a minute of unconstrained tracking.

Two dominant techniques to address drift have emerged. The first is to avoid the triple integration typical of inertial systems (single integration of angular velocity followed by double integration of acceleration) by using pedometry, detecting when a step has been taken and characterising it based on duration and similar quantities—i.e. without integrating noisy data. To account for the errors this itself adds, it has been necessary to incorporate building maps as a series of wall constraints using a higher-level particle filter. The second approach is to provide strong external constraints, usually by constraining the sensor to be foot mounted [1]. In fact, a particle filter has proved necessary for medium-term tracking even for foot-mounted sensors.

The touted platform for these systems is the increasingly ubiquitous smartphone. However, particle filters suffer from high computational demands that can preclude their use on such mobile devices. In this paper, we advocate offloading the particle filter to dedicated computing infrastructure in the building. Such a system would alternate between performing the computation locally on the mobile device and performing it remotely on the server while the sensor data is streamed over the network. As a first step towards enabling such a system, we show how to exploit the latent parallelism in the location particle filter to adapt it for efficient execution on commodity Graphical Processing Units (GPUs), thereby offering scalable localisation services to multiple devices concurrently.

Specifically, we make the following contributions:

- We define the characteristics of the location particle filter and identify its parallelisation opportunities. We focus on making the resampling step efficient and appropriately adapting the number of particles dynamically.
- We show how to map the parallelisable portions of a single filter onto a GPU architecture. We partition the computation in such a way that data transfers between the GPU and CPU are minimised while the available parallelism is maximised.
- We show how to localise multiple users by running multiple filters on the GPU concurrently. We develop a novel paging scheme for managing particles which makes sure that the memory used up by each filter is adjusted as the number of required particles is reduced.
- We provide extensive evaluation of our GPU-based location system using previously published real world data and compare its performance with both a sequential and multithreaded CPU implementation.

The paper is structured as follows: in Section II we study related work; Section III provides an overview of the structure and function of a parallelised particle filter; in Section IV we discuss our evaluation approach; Sections V and V-B look at the use of a GPU to optimise the localisation of a single device; Section VI then extends this to multiple devices needing simultaneous location results. We evaluate our system experimentally throughout and provide concluding remarks in Section VII.

II. Related Work

Pedestrian inertial navigation systems began with Foxlin's work on foot-mounted inertial sensors [1]. Foxlin introduced zero velocity updates (ZUPTs)—whereby the sensor is essentially recalibrated when the foot is in contact with the floor and is therefore stationary—and applied an Extended Kalman Filter to generate *step events* describing each step in terms of a {length, heading} vector. ZUPTs greatly extend the tracking range of the sensors, but drift inevitably builds to the point of domination.

Krach and Robinson [2]; Widyawan, Klepal and Beauregard [3]; and Woodman and Harle [4] independently incorporated building maps via particle filters to help constrain the drift accrual. Particle filters [5] are numerical approximations to Bayesian filters and are composed of a set of *particles*. In this context, each particle represents a distinct hypothesis for the user's location and contains a *weight* value that represents the current confidence in that particular hypothesis. The filter continuously updates the particles based on incoming step events. This makes it particularly easy to incorporate 2D maps: each room is stored as a simple polygon and any particle that crosses a polygon (equivalent to walking through a wall) is assigned a weight, w=0. The process is illustrated in Figure 2, where each particle is represented by a filled circle (w=1) or a hollow circle (w=0) and the true position is shown by a blue cross. In Figure 2(a), N particles are uniformly distributed under the assumption we do not know where the user is. In Figure 2(b), a step event shifts all particles to the left; some cross the wall boundary and are assigned w=0. In Figure 2(c) we *resample*—i.e. we create N new particles by cloning the current set in proportion to their weights. This implicitly removes those that crossed the wall, whilst ensuring N is not reduced for the next step. The three steps of the cycle are described in more detail in Figure 1.

As more steps are integrated, the particles should either converge on the true position as shown in Figure 3 or a positioning failure occurs, where there are no particles with non-zero weights left (Figure 4). In such cases we must completely restart the filter.

Woodman et al. reported that $N \approx 10^6$ particles were needed to reliably localise a pedestrian within a building of 8725 m^2 and that a dedicated desktop machine could only update 300,000 particles per second [4]. However, they also noted that far fewer particles were needed once the particle cloud had converged. They considered there to be two regimes: *localisation* before the cloud had converged; and *tracking* once it had. The transition between the regimes typically occurred after tens of steps.

1) **Propagation**. The new state of each particle is predicted from its previous state and a motion model. Each particle is displaced by a vector that matches the step event and perturbed by noise to represent the event measurement error.

2) **Correction**. The particle weights are recomputed based on information from the map. All particles that pass through walls are assigned a weight of 0.0 (shown as hollow circles in Figure 2(b)).

3) **Resampling**. We draw a new set of particles from the distribution approximated by the remaining particles. Particles are sampled in direct proportion to their weight, making it more likely that the trusted hypotheses will continue to the next iteration (in our simplified example, all particles within the room have equal chance of being sampled, whilst those outside the room will not be sampled at all.) Assuming the particle set has a constant size, some particles will be duplicated in the next iteration as shown in Figure 2(c).

Fig. 1: The particle filter steps.

To allow more tractable values of N, researchers have either provided an initial hint about the user's location [2], [3] or incorporated WiFi fingerprinting to bound the area in which the user is to be found [6]. The latter is a scalable solution that can dramatically reduce N. However, N may still be too large for a mobile device such as a smartphone to run in realtime since integrated device sensors are invariably of lower quality than the test devices used to date *and* they cannot be constrained to lie on the foot. The associated increase in error demands an increase in the N to avoid positioning failure. Timeliness issues aside, the processing would also adversely affect both its battery lifetime and its core performance. Constant WiFi scanning in particular is a costly task.

An alternative approach is to offload as much processing as possible from the mobile device. In principle, it could be used as a pure wireless sensor, constantly reporting the raw sensor data to a network processing service. However, this would entail sending a constant stream of sensor data to a server, which would itself severely impact lifetime and performance. Instead, we advocate processing the sensor data into step events on the mobile device, and then offloading these to a network processing node that can apply a particle filter. With a typical generation frequency of 1 Hz and a size of a few bytes, the step events put minimal load on the mobile device's wireless controller.

In such an architecture, it is essential that the processing infrastructure be scalable—it is not realistic to require a dedicated desktop-grade machine for every building user. In

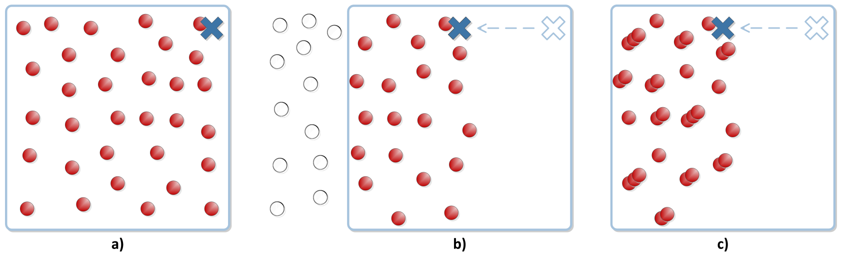


Fig. 2: First iteration of a particle filter.

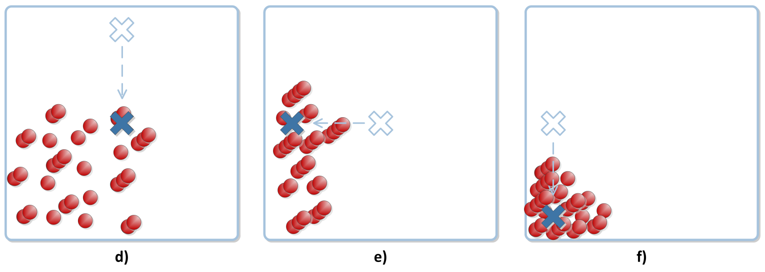


Fig. 3: Second, third and fourth iterations of a particle filter showing convergence.

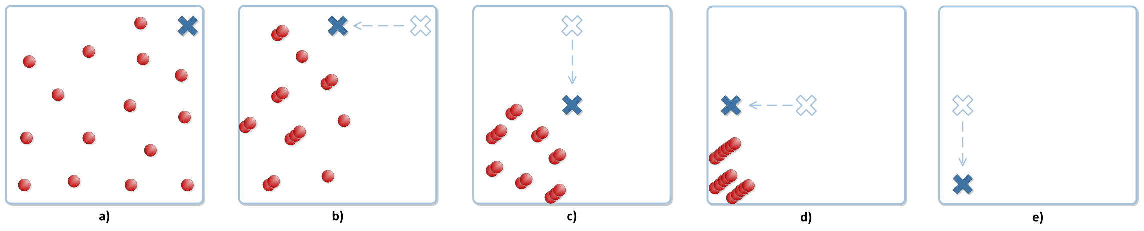


Fig. 4: Example of the particle filter with reduced number of particles.

this work we seek to parallelise the location particle filter and use commodity GPUs to increase the number of concurrent filters. There is an added benefit that such a system could apply directly to future mobile devices that feature their own GPU.

Parallelised particle filters running on GPUs have already been demonstrated in the image processing domain [7]–[11]. However, there are important differences between this and the pedestrian tracking domain:

- **Number of particles.** The total number of particles in imaging applications rarely exceeds 10^3 [12]. Quality is improved by expanding the dimensionality of the particle state rather than increasing particle numbers. Filters for location have small state dimensionality and particle sets in excess of 10^6 may be necessary.
- **Particle weighting procedure.** Weight assignment is the usual motivation for using GPUs in imaging, since generating comparison images or comparing many state dimensions is a slow process. For location filters, the motivation for parallel processing is the sheer number of particles.
- **Particle number adaptation.** For efficiency it is wise to dynamically adapt the number of particles in a filter. This has been used for visual tracking, where reported reductions of 20% are possible. In contrast, reductions of around 90% can be seen for location systems [4].

There have also been attempts to boost particle filter performance using GPUs in other domains [13]–[17]. These approaches were also designed for smaller sets of particles: [13]–[16] reported evaluation for less than 10^5 particles with super real-time processing times, while [17] reported duration of about $100s$ per step for 10^6 particles. In contrast, our approach, using comparable hardware, enables processing of even larger particle sets within less than a second per step.

Parallelisation of particle filters on a network of computers has also been investigated in prior work [18]–[23]. These works targeted generic, canonical examples of particle filters with the main concern being how to perform the filter's resampling step in a distributed setting.

III. Parallelising the Particle Filter

The structure of the particle filter makes it a promising target for parallelisation. The first two steps, propagation and correction, are naturally parallelisable as they involve processing only the local state of each particle. The final step, resampling (see Figure 1), can also benefit from parallelisation, although it is predominantly sequential. However, its parallel formulation is not as straightforward since it depends on joint processing of all particles. Indeed, resampling is regarded as the major challenge in parallelising particle filters [19]–[23].

The difficulty is that the resampling step requires a *global* view of the particle weights in order to ensure it resamples in proportion to the weights. There are various resampling procedures that can be used, but all require this global view that hinders parallelisation. In our implementation we have used the most prevalent algorithm: Sequential Importance Sampling with Re-sampling (SISR).

Our implementation of SISR involves numbering each particle and then forming a mapping between the particle number and the cumulative weight (the cumulative weight sum, or CWS). The CWS can be computed in parallel by a parallel prefix-sum (scan) algorithm [24]. To resample N particles, we draw N random numbers uniformly between zero and the sum of all weights (i.e. the maximum CWS value). With each random number we look up the particle number associated with that value, clone it, and add it to the resampled set.

It is useful to dynamically vary N both because it minimises the necessary resampling effort, and because it allows optimisation of memory to support multiple filters simultaneously. As in [4], we apply the KLD (Kullback-Leibler Distance) adaptation first described in [25]. If we consider the set of particles at each iteration as an approximation of an underlying distribution, KLD is a statistical procedure that provides a bound on the error introduced when reducing the set size. A growing error means that more resampling is needed; a shrinking error allows us to use fewer particles.

In our KLD implementation, we divide the building floor into a regular grid of *bins*. We add a small step to the end of the propagation-correction-resample cycle that computes the bin to which each particle belongs. As we iterate in the filter, we continually evaluate the KLD number, terminating the resampling process whenever it falls below the number of particles we have already resampled.

We give details of our GPU implementations of the resampling step in Sections V-A and V-C.

A. Scaling to Multiple Users

Although not well defined, we expect the transition between the localisation regime and the tracking regime to occur within tens of steps taken by the user. At any given moment we expect only a minority of building users to be in the costly localisation regime. In this work we exploit this to efficiently share GPU resources amongst multiple filters. We address two sub-problems in turn:

1) implementing a single location filter on a GPU; and
2) efficiently running multiple GPU-based location filters concurrently.

IV. The Approach to Evaluation

Before dissecting our GPU-based filters, we briefly discuss GPU limitations and our experimental setup.

A. Dealing with GPU Limitations

Whilst GPUs have become very powerful in recent years, they still have limitations that must be addressed for any general purpose computation such as a particle filter. The two primary limitations are host-GPU communication bottlenecks and small amounts of on-board GPU memory.

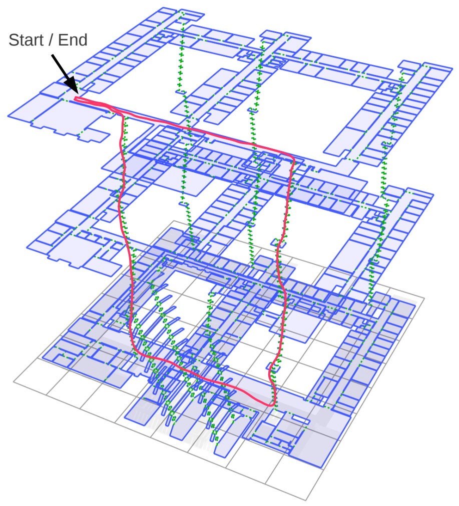


Fig. 5: Trace in a three-storey building [4] used as an input in experiments.

Communication bottlenecks occur when transferring large volumes of data between the host (CPU) and GPU memories, a process that is typically very slow relative to the data processing speeds of the GPU as a whole. Therefore, in our implementation we try to minimise the data transferred between host and GPU.

The on-board memories available on a GPU include fast, dedicated local memory available to each thread. Unfortunately such memory is typically too small to accommodate the high particle numbers, particle state size and complex map data of our particle filter. The consequence of this is that filters must use the slower forms of memory on the graphics card, most notably the large but slow global memory. We judiciously use the local memory to alleviate the penalty of using global memory whenever possible.

B. Implementation and Evaluation Setup

We have implemented and evaluated a single and multiple users version of the location filter on a GPU (see Sections V and VI). The CPU version of the filter (the baseline) was implemented in C using POSIX threads, while the GPU-based filters were implemented using the CUDA platform for GPU computing from NVIDIA[1].

To benchmark the various filters we used the data from [4]. These data were collected in a 8725 m^2 building with three floors and contain step events and a detailed building model—full details were provided in [4]. For this paper, we singled out a representative trace (see Figure 5) and used it as an input in all reported experiments. The performance was similar for other collected traces.

All tests were executed using a standard desktop host and a mid-range consumer NVIDIA GPU (see Table I).

[1]http://developer.nvidia.com/cuda

CPU	
Model:	Intel® Core^{TM}2 Quad
CPU clock:	2 GHz
OS:	Ubuntu 9.04 (64 bit)
GPU	
Model:	GeForce GTX 460
No. of multiprocessors:	7
No. of cores:	48
GPU clock:	1.42 GHz
Computing capability:	2.1
Driver ver:	3.20

TABLE I: Hardware setup.

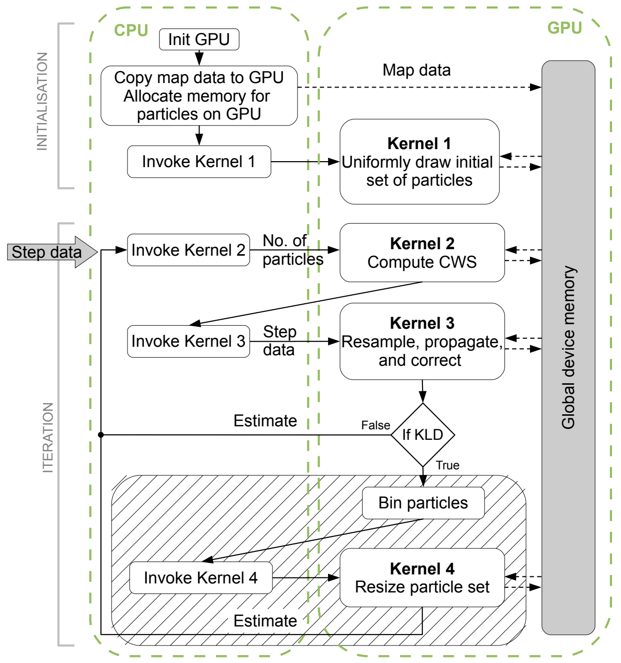


Fig. 6: Structure of the GPU particle filter.

V. A Single Filter on a GPU

We now describe how we mapped the parallelisable portions of a single filter onto a GPU architecture. We do so first for the standard filter with SISR resampling (Section V-A) and then for the variant with KLD adaptation (Section V-C).

A. Structuring the Filter for a GPU

The particle filter algorithm is comprised of an *initialisation* stage that allocates space for the particles and draws an initial particle set, followed by an *iteration* stage that carries out the propagation, correction and resampling steps (Figure 1) in a continuous loop. We mirror these stages in the GPU implementation of the algorithm.

Figure 6 illustrates the structure of our GPU filter. In the initialisation stage, we copy map data to the GPU, allocate GPU memory for particles and then use a GPU kernel to draw the initial (uniform) distribution of particles. Because the polygons used to describe the environment are quite complex (we allow both convex and concave shapes), this is a computationally expensive task. The GPU kernel is run in such a way so that one thread is assigned per polygon. Inevitably this approach causes uneven workload distribution among working threads which is not desirable on the GPU. However, we store the resultant particle distribution on the GPU and re-use it to initialise every particle filter. In this way we only have to generate the particle set once in the lifetime of the program, so generating it directly on the GPU avoids the start-up overhead of a large data transfer. We consider the map data to be immutable (a reasonable proposition for most buildings) so we store it in the read-only cached texture memory rather than in the global device memory of the GPU, to allow faster memory access. The use of texture memory boosted the performance of the GPU filter by up to 40%.

The iteration stage is an adaptation of the traditional particle filter to perform a resample-propagate-correct sequence of steps rather than the more traditional propagate-correct-resample. The cyclic nature of the filter means this makes no difference to the outcome, but it enables us to use a single GPU kernel to embody the particle filter iteration. We can compute the CWS either sequentially with a single GPU thread or in parallel using the parallel prefix-sum algorithm. Parallel CWS computation has additional costs for data transfer between local and global memories and for thread synchronisation, so such an implementation will be faster than the single-threaded one only for sufficiently large particle sets.

We emphasise that in the GPU filter there is no significant data movement between the CPU and the GPU. Only the number of particles and the step data from CPU to GPU and the location estimate from GPU to CPU are transferred, at negligible cost. No particle data ever leaves the global device memory, making the GPU filter fully optimised with respect to the amount of CPU-GPU data transfer involved.

We discuss the KLD portion of the GPU filter structure in Section V-C.

B. Filter Performance

The unusual capabilities of the GPU make it hard to predict whether it would be better to compute the CWS sequentially or in parallel. Therefore, we experimentally evaluate performance of the two versions for varying number of particles. Figure 7 shows how the two GPU versions performed against the fastest CPU version with CWS computed both using a single thread and parallel prefix-sum. We observe that the single-threaded CWS computation offers no advantage whatsoever for large particle numbers used in localisation.

The fastest CPU version was obtained by running the multithreaded CPU filter, which works by partitioning the particle set into equal portions and handling each portion with a separate thread, with 4 threads. Since our CPU has 4 cores, there is no performance benefit in using more than 4 threads on a CPU for running the filter. This effect is evidenced in Figure 8, which shows how the average time taken for an iteration of GPU and CPU filters with parallel CWS varied with the number of particles in use. The massively parallel GPU filter outperformed the CPU implementation for

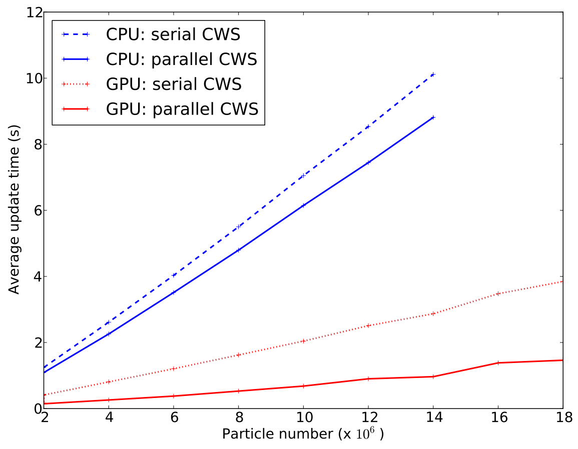


Fig. 7: Average iteration times of GPU filter with single threaded and parallel prefix-sum CWS computation compared with CPU baselines.

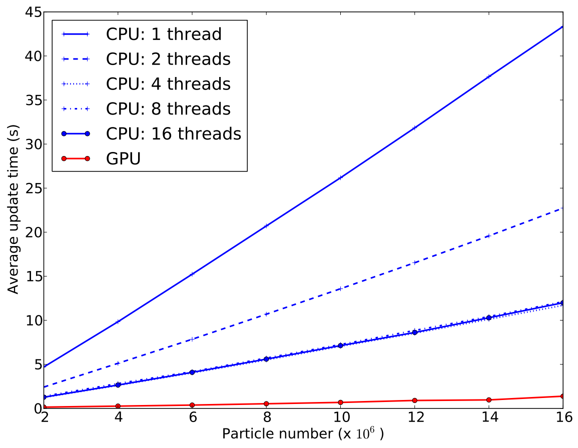


Fig. 8: Comparison of the GPU filter iteration time with the multithreaded CPU implementation.

all numbers of CPU threads, and enabled processing of step events in real time even for the largest particle set containing 16 million particles.

The theoretical computational complexity of the sequential particle filter algorithm is $\mathcal{O}(n \log(n))$, where n is the number of particles in use. A truly parallel version could achieve $\mathcal{O}(\log(n))$ complexity, but sadly neither the CPU nor GPU cannot offer this because the number of particles (and hence the number of threads) exceeds the number of available processing units by far. The result is that the practical complexity of the parallel version of the GPU and all CPU variants remains the same as the theoretical sequential algorithm, albeit with a different multiplicative constant. From Figure 8 we find that the GPU filter gave speed-up of approximately 8.8 against the 4-threaded CPU variant.

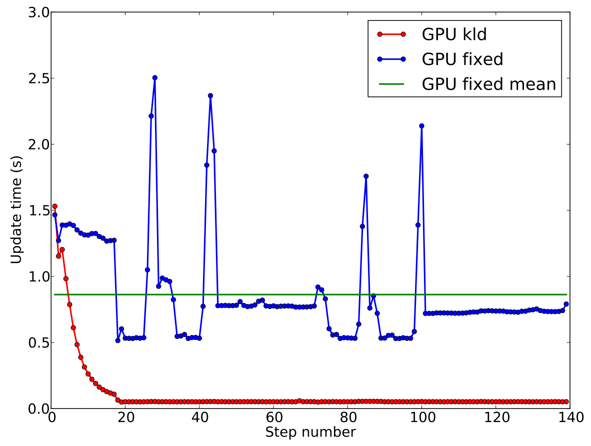


Fig. 9: Update times of GPU filter with fixed and KLD number of particles over a sequence of steps.

C. KLD Resampling

Given that the parallel particle filter algorithm offers the same computational complexity, it is advantageous to use KLD resampling to keep the number of particles approximately minimal at each stage. The portion of the GPU filter in charge of KLD resampling is illustrated in the shaded region of Figure 6. We add a new task to the end of the third kernel: computing the bin to which the updated particle belongs. Because the particle updates occur in parallel, we must update every particle from the previous iteration, including assignment of the bin. This differs from the sequential approach, where we were able to terminate resampling before it had completed on every particle by continually evaluating the KLD parameter after each resample. Here, we first determine bins for all particles and then in a subsequent kernel calculate the number of non-empty bins (again using parallel prefix-sum) and the new number of particles.

To demonstrate effectiveness of KLD resampling, we ran the GPU filter with the maximal fixed and variable number of particles on our trace. We measured the duration of each filter update for each step from the trace. The obtained processing times are shown in Figure 9. We observe that for the KLD filter update times plummeted, reflecting the reduction of the particle set. On the other hand, update times of the filter with fixed number of particles not only did not improve much but instead varied significantly. This discrepancy in update times was caused by geometries of varying complexity occurring at different parts of the trace (cf. Figure 5).

Next we compared the performance of KLD resampling on CPU and GPU. Figure 10 shows the mean time taken for iteration of CPU and GPU filters with KLD resampling for different bin sizes on trace from Figure 5. We used KLD with ϵ value of 1.5% and varied bin sizes from 0.6 m to 4.0 m. Each bin size required initial uniform distribution of a different number of particles, decreasing as the bin size grew larger—e.g., 11.4 million particles were required for bin size

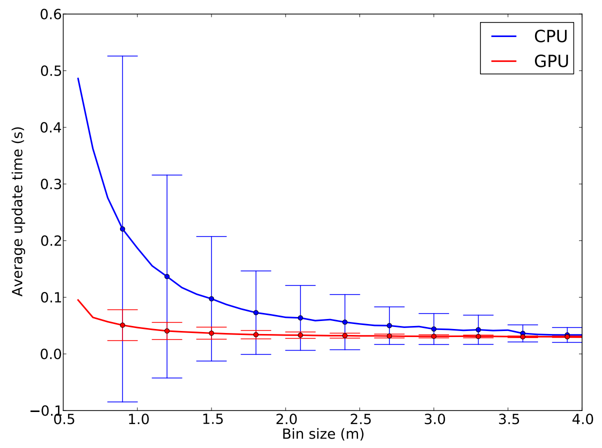


Fig. 10: Average update times of CPU and GPU filters with KLD resampling for different bin sizes.

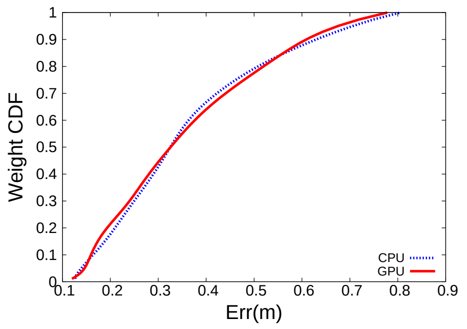


Fig. 11: Comparison of accuracy achieved with sequential and parallel implementations.

0.6 m, 4.1 million for bin size 1.0 m and 1 million for bin size 2.0 m. While we observe a steady decrease in mean processing times as the bin size increases for both filters, the CPU filter manifests much higher standard deviation (indicated by vertical segments in Figure 10; the standard deviation for bin size 0.6 m, which is not shown, was 1.46). This effect was also caused by differences in geometry along the trace. However, GPU filter benefited from the available parallelism in the GPU and was able to process step events in complex geometries much faster than the CPU. Since smaller bin sizes are favoured whenever possible (the probability of unsuccessful localisation is higher for larger than for smaller bin sizes), GPU filter provides much greater benefits than the CPU variant.

a) Run-time profile: To provide a better insight into the distribution of workload during filter computation we measured proportions of time spent in different particle filter stages. Figure 12 shows a profiler-style breakdown for both CPU and GPU versions of the filter with SISR and KLD resampling.

b) Accuracy: Figure 11 presents a comparison of the accuracy of the estimated pedestrian position obtained from the GPU and CPU versions of the filter. We used the same ground truth data as used in [4], although we did not use the clustering algorithm described therein. We observed that the KLD-enabled GPU filter achieves comparable results to the slower CPU version and take this as evidence that our parallel implementation was functioning as expected.

VI. Multiple Concurrent Particle Filters

Having optimised the particle filter itself, we now turn to the problem of handling multiple users, each associated with a distinct particle filter.

Figure 14 depicts our multi-user system. It can work in one of two modes, *fixed* or *paged*, depending on whether the particle number assigned to each user will be kept fixed or dynamically adjusted.

When working in fixed mode, the system allocates GPU memory to hold a fixed number of particles per user and does *not* use KLD to dynamically adapt the number in use. If there is insufficient memory available, users are queued.

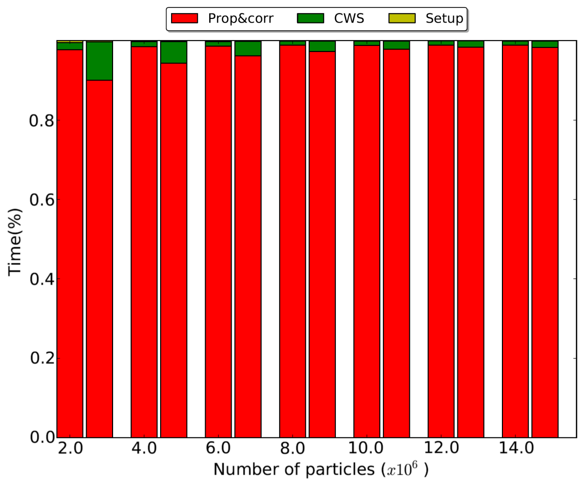
In contrast, paged mode initially assigns the same particle number per user as the fixed mode, but reduces it when the particle set starts to converge. Over the lifetime of a filter, we expect the majority of time to be spent in the converged state and hence this scheme allows more users to be simultaneously tracked. However, it also requires additional computation to handle user and particle related data, namely, Kernels 1 and 4 in Figure 14.

It is important to note that in both modes no particle data (i.e. neither state, weights nor bins) are exchanged between the CPU and the GPU. Both modes of operation will be explained in more detail in the next sections.

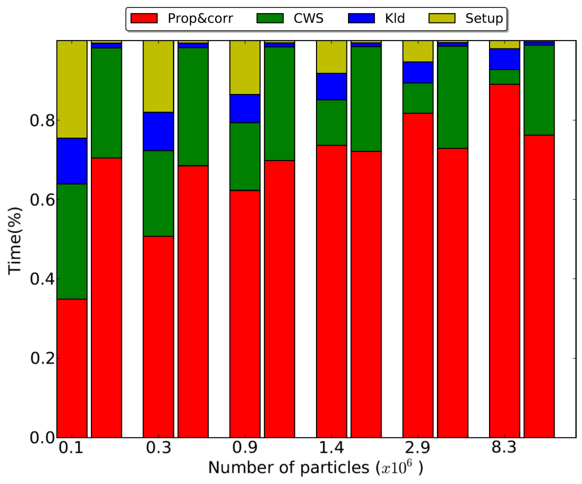
A. Performance in Fixed Mode

The fixed mode allows us to guarantee that the system can process a predefined number of users, as dictated by the available memory. Using our GPU with 2 GB of memory we were able to support up to 17 concurrent particle filters with 1,200,000 particles. Each filter was updated in a round-robin fashion as new step events were produced. Figure 13 shows how long it took to update all of the filters (labelled ‘GPU fixed’ in figure) for different numbers of system users. To obtain the results for n users we simply duplicated the input sensor trace n times. As expected, we observe a linear increase in the processing time as the number of users increased. This is simply a reflection that the GPU carried out a very similar operation on each user in each iteration. The lack of noise in the trend highlights a further advantage of the GPU: it is not affected by other processes running on the machine, as the CPU might be.

Our interest is in how many filters can be concurrently processed in real time. By this we mean that the processing of each step for all users must complete before the next step begins. From the data we have, a typical step lasts 1 s. From Figure 13 we estimate that our setup could track up to 20 users in real time (with little extra device memory).



(a) SISR resampling with a fixed number of particles.



(b) Number of particles varied by KLD resampling. The number on x-axis is the initial particle number.

Fig. 12: Proportions of time spent in different particle filter stages. In all 2-column groups, the left column corresponds to the 4-threaded CPU filter and the right column to the GPU filter.

B. *Paged Mode*

Although the fixed mode already gave a significant boost, it may not be an optimal solution. As we have already demonstrated, the required number of particles significantly decreases as we move from localisation to tracking, and the fixed mode does not exploit this. Dynamically reducing the number of particles per user not only reduces the step processing time, but it also frees up the GPU memory that can then be used to support the tracking of more users. Clearly we cannot support any more users when all of the current ones are in the localisation phase (since they each may need the full number of particles), but this should be a rare occurrence—at any given moment we would expect the vast majority of users to be in the *tracking* phase since the localisation phase lasts only tens of steps.

This discrepancy is mitigated with the paged mode. In this mode, our implementation allocates the entirety of the GPU device memory in the initialisation phase, and manages it using custom data structures implemented within that allocated memory (i.e., management was entirely on the GPU). By creating a *paging* system, named for its analogy to virtual memory paging in operating systems, the allocated GPU memory is partitioned into equal-sized chunks ('pages'), each of which can hold the state for a predetermined number of particles. Two array structures embedded within the allocated memory keep track of the user-page assignments and the unused pages.

In the initialisation phase, each filter is assigned just enough pages to exceed the required number of particles. At the end of each step, the KLD parameter is evaluated for each filter and the number of assigned pages adapted to suit the new particle requirement. Consequently, the number of particles assigned to a filter is always an integer multiple of the number supported by each page. Figure 15(a) (Figure 15(b)) shows the variation in particle number (resp. processing time) with respect to the step number obtained for a specific sensor trace. Results are shown for a variety of page sizes.

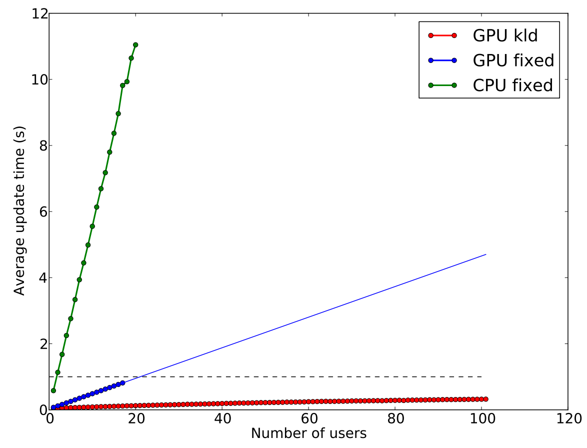


Fig. 13: Performance comparison of the multi-user system.

C. *Selecting a Page Size*

Selecting the optimal page size is more of an art than a science. Too small and the GPU spends a long time looking up pages in the large data structures; too large and KLD will have little effect. We chose a page size to match the typical number of particles needed in the tracking phase (this was 128,000 particles for our data).

Note that the number of steps it takes for the filter to reach the tracking mode is greater when the page size is smaller. Intuitively, this is because smaller page sizes permit more

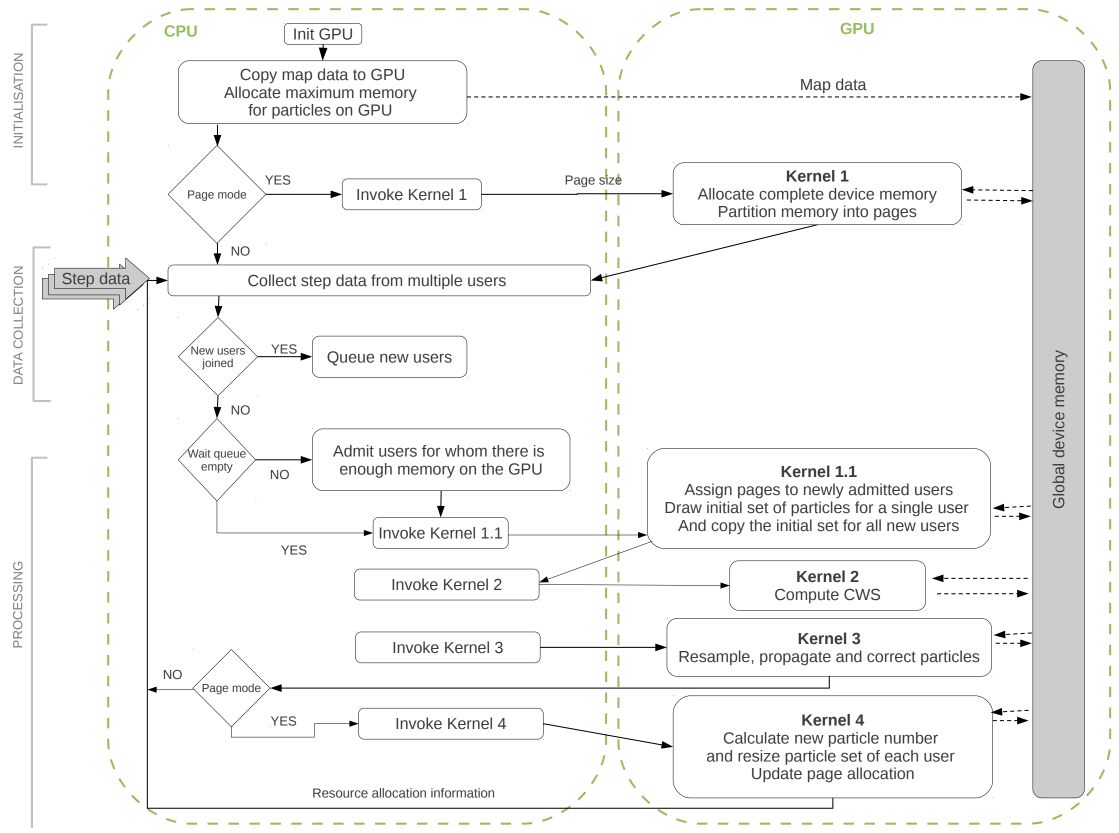

CPU
GPU
INITIALISATION
DATA COLLECTION
PROCESSING
Init GPU
Copy map data to GPU
Allocate maximum memory
for particles on GPU
Map data
Page mode
YES
NO
Invoke Kernel 1
Page size
Kernel 1
Allocate complete device memory
Partition memory into pages
Step data
Collect step data from multiple users
New users joined
Queue new users
Wait queue empty
Admit users for whom there is enough memory on the GPU
Invoke Kernel 1.1
Kernel 1.1
Assign pages to newly admitted users
Draw initial set of particles for a single user
And copy the initial set for all new users
Invoke Kernel 2
Kernel 2
Compute CWS
Invoke Kernel 3
Kernel 3
Resample, propagate and correct particles
Invoke Kernel 4
Kernel 4
Calculate new particle number
and resize particle set of each user
Update page allocation
Resource allocation information
Global device memory

Fig. 14: The multi-user system.

regular decreases in the particle number, as per Figure 15(a). In the future, we hope to consider dynamic adaptation of the page size, starting with a large value at the beginning of localisation, and then reducing the page size once the current particle number reaches the current page size.

D. KLD Overhead

Inevitably, the KLD adaptation and the paging scheme add an overhead to the processing. The initial overhead is caused by the binning and evaluation of the required number of particles at the end of each update. Paging does give one small advantage here: since we allocate particles in chunks of the page size, we need only evaluate the KLD factor for every page rather than for every particle. In the data we have presented, we used a single GPU thread to carry out the KLD factor evaluation at every page boundary.

E. Performance in Paged Mode

Because KLD is fundamentally adaptive, it is difficult to assign meaningful performance metrics. In Figure 13 we seek to bound the performance by looking at how many users can be added to the multi-user system when they all fall into the tracking phase. With the above settings there was sufficient GPU memory to *track* 101 users, and enough processing power in the GPU for all of them to be processed in real time.

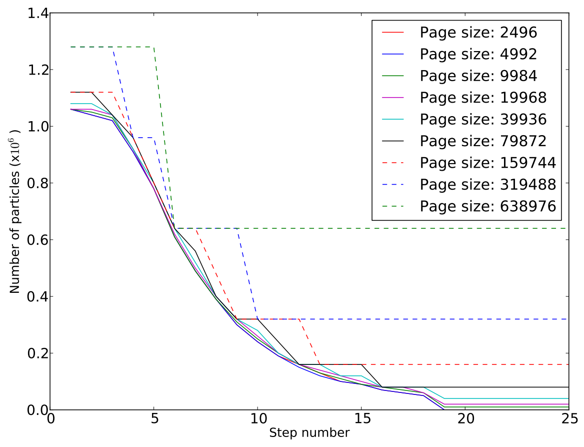
VII. Conclusions

Infrastructure-less positioning systems are promising alternatives to fixed infrastructure systems, but they can require a significant amount of processing. Today's systems, which employ a particle filter together with map data to provide accurate estimates, fail to work in real time.

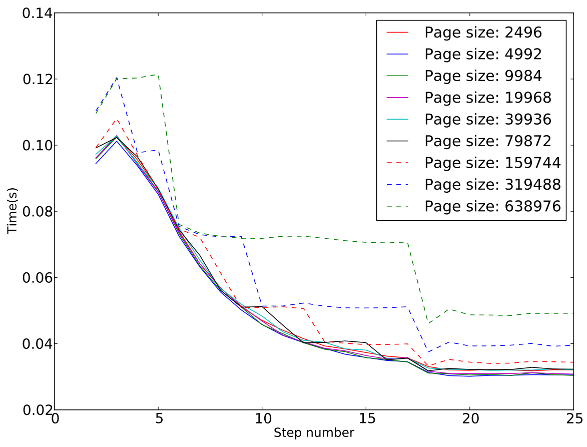
In this work we have looked at how to parallelise the particle filter to support high-accuracy localisation of multiple users using only one standard system. Our main contributions are in the parallelisation of the location particle filter, its implementation using a commodity GPU, and its extension to cope with multiple concurrent users using a novel paging scheme.

We have evaluated our implementation in detail using real-world data collected in a large building. Our results show a significant improvement over the current state-of-the-art. Whereas previous work using the same data was unable to process more than one user in real-time (and did not always achieve even this), our system is able to process between 17 and 101 users in real-time, depending on how constrained their positions are. In the general case, we would expect the vast majority of users to be well localised and thus only a few such systems would be needed to track the occupants of a large building.

We believe that the use of GPUs or similar parallel processing architectures will significantly enhance the scalability of future tracking systems and we hope to build on this work



(a) Variation in the number of particles assigned to a single filter for different page sizes.



(b) Variation in processing times for different page sizes.

Fig. 15: Analysis of the paging system.

to demonstrate a building-wide tracking system capable of tracking all occupants.

References

[1] E. Foxlin, "Pedestrian tracking with shoe-mounted inertial sensors," *IEEE Computer Graphics and Applications*, vol. 25, no. 6, pp. 38–46, 2005.

[2] B. Krach and P. Robertson, "Integration of foot-mounted inertial sensors into a Bayesian location estimation framework," in *5th Workshop on Positioning Navigation and Communication*. IEEE, 2008, pp. 55–61.

[3] M. Klepal and S. Beauregard, "A Backtracking Particle Filter for fusing building plans with PDR displacement estimates," in *5th Workshop on Positioning Navigation and Communication*. IEEE, 2008, pp. 207–212.

[4] O. Woodman and R. Harle, "Pedestrian localisation for indoor environments," in *UbiComp 2008*, vol. 344. ACM, 2008, pp. 114–123.

[5] J. Hightower and G. Borriello, "Particle filters for location estimation in ubiquitous computing: A case study," in *UbiComp 2004*, ser. LNCS, vol. 3205. Springer, 2004, pp. 88–106.

[6] O. Woodman and R. Harle, "RF-based initialisation for inertial pedestrian tracking," in *Pervasive*, 2009, pp. 238–255.

[7] A. S. Montemayor, J. J. Pantrigo, A. Sánchez, and F. Fernández, "Particle filter on GPUs for real-time tracking," in *ACM SIGGRAPH 2004 Posters*, ser. SIGGRAPH '04. ACM, 2004, p. 94.

[8] O. Lozano and K. Otsuka, "Simultaneous and fast 3D tracking of multiple faces in video by GPU-based stream processing," in *IEEE International Conference on Acoustics, Speech and Signal Processing, 2008 (ICASSP 2008)*, Apr 2008, pp. 713–716.

[9] C. Lenz, G. Panin, and A. Knoll, "A GPU-accelerated particle filter with pixel-level likelihood," in *Vision, Modeling, and Visualization Conference 2008*. Aka GmbH, 2008, pp. 235–241.

[10] R. Cabido, D. Concha, J. J. Pantrigo, and A. S. Montemayor, "High speed articulated object tracking using GPUs: A particle filter approach," in *10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN 2009)*. IEEE Computer Society, 2009, pp. 757–762.

[11] P. Li, "An efficient particle filter-based tracking method using graphics processing unit (GPU)," *Signal Processing Systems*, vol. 68, no. 3, pp. 317–332, 2012.

[12] C. Chang and R. Ansari, "Kernel particle filter for visual tracking," *Signal Processing Letters, IEEE*, vol. 12, no. 3, pp. 242 – 245, march 2005.

[13] M.-A. Chao, C.-Y. Chu, C.-H. Chao, and A.-Y. Wu, "Efficient parallelized particle filter design on CUDA," in *2010 IEEE Workshop on Signal Processing Systems (SIPS)*, Oct 2010, pp. 299–304.

[14] C.-Y. Chu, C.-H. Chao, M.-A. Chao, and A.-Y. Wu, "Multi-prediction particle filter for efficient parallelized implementation," *EURASIP J. Adv. Sig. Proc.*, vol. 2011, p. 53, 2011.

[15] J. Wu and V. Jilkov, "Parallel multitarget tracking particle filters using graphics processing unit," in *44th Southeastern Symposium on System Theory (SSST)*, Mar 2012, pp. 151–155.

[16] A. Lee, C. Yau, M. B. Giles, A. Doucet, and C. C. Holmes, "On the utility of graphics cards to perform massively parallel simulation of advanced Monte Carlo methods," *Journal of Computational and Graphical Statistics*, vol. 19, no. 4, pp. 769–789, 2010.

[17] G. Hendeby, R. Karlsson, and F. Gustafsson, "Particle filtering: The need for speed," *EURASIP J. Adv. Sig. Proc.*, vol. 2010, 2010.

[18] S. Sutharsan, A. Sinha, T. Kirubarajan, and M. Farooq, "An optimization based parallel particle filter for multitarget tracking," in *Proceedings of the SPIE*, vol. 5913, no. 12, 2005, pp. 87–98.

[19] S. Hong and P. M. Djuric, "High-throughput scalable parallel resampling mechanism for effective redistribution of particles," *IEEE Transactions on Signal Processing*, vol. 54, no. 3, pp. 1144–1155, 2006.

[20] B. Balasingam, M. Bolic, P. M. Djuric, and J. Míguez, "Efficient distributed resampling for particle filters," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2011)*. IEEE, 2011, pp. 3772–3775.

[21] J. Míguez, "Analysis of parallelizable resampling algorithms for particle filtering," *Signal Processing*, vol. 87, no. 12, pp. 3155–3174, 2007.

[22] M. Bolic, P. M. Djuric, and S. Hong, "Resampling algorithms and architectures for distributed particle filters," *IEEE Transactions on Signal Processing*, vol. 53, no. 7, pp. 2442–2450, 2005.

[23] H. Medeiros, X. Gao, J. Park, R. Kleihorst, and A. Kak, "A parallel implementation of the color-based particle filter for object tracking," in *ImageSense'08: Workshop on Applications, Systems, and Algorithms for Image Sensing*, 2008.

[24] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Computers*, vol. 38, no. 11, pp. 1526–1538, 1989.

[25] D. Fox, "Adapting the sample size in particle filters through KLD-sampling," *I. J. Robotic Res.*, vol. 22, no. 12, pp. 985–1004, 2003.