Particle Filters on Multi-Core Processors

Chitchian, M.M.; van Amesfoort, A.S.; Simonetto, A.; Keviczky, T.; Sips, H.J.

Publication date
2012

Document Version
Final published version

Citation (APA)
Chitchian, M. M., van Amesfoort, A. S., Simonetto, A., Keviczky, T., & Sips, H. J. (2012). *Particle Filters on Multi-Core Processors*. (PDS-2012-001 ed.) (Parallel and Distributed Systems Group report serie; Vol. 2012, No. 1). Delft University of Technology.

Delft University of Technology
Parallel and Distributed Systems Report Series

Particle Filters on Multi-Core Processors

Mehdi Chitchian[1], Alexander S. van Amesfoort[1],
Andrea Simonetto[2], Tamás Keviczky[2],
and Henk J. Sips[1]

mehdi.chitchian@gmail.com, {a.s.vanamesfoort, a.simonetto, t.keviczky, h.j.sips}@tudelft.nl

[1]Parallel and Distributed Systems Group
Faculty Electrical Engineering, Mathematics, and Computer Science
[2]Delft Center for Systems and Control
Faculty Mechanical, Maritime and Materials Engineering

Report number PDS-2012-001 (February 2012)

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems group
Department of Software and Computer Technology
Faculty Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
`reports@pds.ewi.tudelft.nl`

Information about Parallel and Distributed Systems group:
`http://www.pds.ewi.tudelft.nl/`

Abstract

The particle filter is a Bayesian estimation technique based on Monte Carlo simulation. The non-parametric nature of particle filters makes them ideal for non-linear, non-Gaussian dynamic systems. Particle filtering has many applications: in computer vision, robotics, and econometrics to name just a few. Although superior to Kalman filters, particle filters have higher computational requirements, which limits practical use in real-time applications.

In this paper, we investigate how to design a particle filter framework for complex real-time estimation problems using modern many-core architectures. We develop a robotic arm application that serves as a highly flexible estimation problem to push estimation rates and accuracy to new levels. By varying different filter and model parameters, we derive rules of thumb for good filter configurations. We evaluate our particle filter with a comprehensive performance and correctness analysis.

Our results significantly lower the development effort of particle filters for other real-time estimation problems. For the most demanding robotic arm configuration, we can process one million particles at an update rate of a few hundred state estimations per second. As such, we see our results as a step towards wider adoption of particle filters, and as a prerequisite to investigate larger filter setups for even more complex estimation problems.

1 Introduction

The problem of estimating the state of a dynamical system through noisy measurements has many applications in science and engineering.

Many applications need to localize or track moving objects and "see" using computer vision [15], GPS [20], radar or sonar sensors. Other dynamic systems with noisy measurements can be found in econometrics [8] and rare event simulation. Many of the quantities which constitute the state of a system cannot be observed directly, but need to be *inferred* from noisy measurements. For state estimation in non-linear and non-Gaussian dynamical systems, particle filters provide the most accurate estimate given a sufficiently large particle population. Particle filters represent a class of Bayesian estimation methods based on Monte Carlo simulations. Their non-parametric form renders them suitable for highly non-linear and/or non-Gaussian estimation problems. This greater flexibility and estimation accuracy, however, comes with the price of increased computational complexity and, thereby, limited practical applicability for real-time estimation problems.

Due to the aforementioned limitations, parametric filters such as the extended Kalman filter or the unscented Kalman filter are often preferred. Although limited to Gaussian systems, these filters can produce accurate estimations with a limited degree of non-linearity. The main advantage of such parametric filters lies in their computational efficiency.

This paper presents a comprehensive study on the issues regarding the design and implementation of particle filters for many-core architectures, such as GPUs and multi-core processors, the required algorithmic changes and its implications for the filter accuracy. A generic particle filtering framework for a number of many-core architectures is proposed which can be applied to various estimation problems.

We will use a realistic (industrial) robot application to examine the central research question of whether particle filters can be *efficiently* implemented on modern many-core hardware for complex real-time estimation problems. In this context, efficiency refers to the utilization of the available hardware resources (e.g. computation and memory throughput). Furthermore, we quantify the effects of the filter parameters on the estimation quality and analyze the performance of implementation on a number of hardware platforms.

We identify the following major contributions for the work presented in this paper:

- We present a fully distributed particle filtering algorithm that is suitable for various many-core architectures.
- Following this distributed scheme, we introduce a generic particle filtering framework, which can be used to implement particle filters given arbitrary, user-specified models.

- We conduct a performance analysis of our particle filter implementation on a number of many-core platforms. We calculate the effective utilization of the hardware resources in terms of instruction throughput as well as memory bandwidth and identify the performance bottlenecks of the various steps of the algorithm. Furthermore, we analyze the scalability of the presented implementation in two directions: (i) increasing filter size, and (ii) increasing state dimension.
- With a particle filter implementation to simulate a realistic tracking problem, we present an in-depth analysis of the filter estimation quality under varying filter parameters.

The remainder of this paper is organized as follows. We begin with an introduction of Bayesian estimation and particle filters in Section 2. In Section 3, we briefly introduce GPGPUs and multi-core CPUs as two different many-core architectures. Our particle filter implementation is described in Section 4. In Section 5, we discuss all our experiments using a realistic tracking application. We present an evaluation of the filter accuracy and performance in Section 6. Related work is discussed in Section 7, followed by Section 8 to conclude this paper with a summary of the results and a look into possible future research directions.

2 Background

2.1 Bayesian Estimation

Bayesian estimation computes a *Probability Density Function* for the state of a dynamical system over the range of possible values. Suppose $\mathbf{x}$ is a quantity which we wish to infer from the measurement $\mathbf{z}$. The probability distribution $p(\mathbf{x})$ represents all our knowledge regarding this quantity *prior* to the actual measurement. This distribution is, therefore, called the *prior probability distribution*. The conditional probability $p(\mathbf{x} \mid \mathbf{z})$, called the *posterior probability distribution*, represents our knowledge of $\mathbf{x}$ having incorporated the measurement data. This distribution is usually unknown in advance as a result of the complex dynamics involved in most systems. The inverse probability $p(\mathbf{z} \mid \mathbf{x})$, however, directly relates to the measurement characteristics. The *Bayes rule* allows us to calculate a conditional probability based on its inverse.

In order to discuss how the *Bayes filter* [24] calculates the state estimate, we first need to model the dynamics of the system. Let $\mathbf{x}_k$ denote the state at time k, and $\mathbf{z}_k$ denote the set of all measurements acquired at time k. Assuming the system exhibits Markov properties, the state $\mathbf{x}_k$ depends only on the previous state $\mathbf{x}_{k-1}$. Therefore, the evolution of the state is governed by the probability distribution: $p(\mathbf{x}_k \mid \mathbf{x}_{k-1})$, which is referred to as the *state transition probability*. The measurements of the state follow the probability distribution $p(\mathbf{z}_k \mid \mathbf{x}_k)$ which is called the *measurement probability*.

The Bayes filter calculates the state estimate, from an initial state $p(\mathbf{x}_0)$, recursively in two steps:

Predict In this step, the state estimate from the previous step is used to predict the current state. This estimate is known as the *a priori* estimate, as it does not incorporate any measurements from the current time step.

$$p(\mathbf{x}_k) = \int p(\mathbf{x}_k \mid \mathbf{x}_{k-1})\, p(\mathbf{x}_{k-1} \mid \mathbf{z}_{k-1})\, d\mathbf{x}_{k-1}$$

Update The state estimate from the previous step is updated according to the actual measurements done on the system. Therefore, this estimate is referred to as the *a posteriori* estimate.

$$p(\mathbf{x}_k \mid \mathbf{z}_k) = \eta\, p(\mathbf{z}_k \mid \mathbf{x}_k)\, p(\mathbf{x}_k)$$

The Bayes filter, in its basic form, is inapplicable for any complex estimation problem. The main problem is that the prediction step requires an integration in closed form, which is only possible for some estimation problems.

2.2 Particle Filter

Particle filtering [11, 1] is a recursive Bayesian filtering technique using Monte Carlo simulations. Particle filters represent the posterior by a finite set of random samples drawn from the posterior with associated weights. Because of their non-parametric nature, particle filters are not bound to a particular distribution form (e.g. Gaussian) and are compatible with arbitrary (i.e. non-linear) state transition functions.

As previously mentioned, particle filters represent the posterior by a set of particles. Each particle $\mathbf{x}_k^{[m]}$ can be considered as an instantiation of the state at time t. In the prediction step of the particle filter, each particle $\mathbf{x}_k^{[m]}$ is generated from the previous state $\mathbf{x}_{k-1}^{[m]}$ by sampling from the state transition probability $p(\mathbf{x}_k \mid \mathbf{x}_{k-1})$. In the update step, when measurement $\mathbf{z}_k$ is available, each particle is assigned a weight $\omega_t^{[m]}$ according to:

$$\omega_t^{[m]} = p(\mathbf{z}_k \mid \mathbf{x}_k^{[m]})$$

Given a large enough particle population, the weighted set of particles $\{\mathbf{x}_k^{[i]}, \omega_t^{[i]}, i = 0, \dots, N\}$ becomes a discrete weighted approximation of the true posterior $p(\mathbf{x}_k \mid \mathbf{z}_k)$.

2.2.1 The Degeneracy Problem and Resampling

A common problem with the basic particle filter algorithm mentioned previously is the *degeneracy* problem. It has been shown that the variance of the weights can only increase over time [6]. This results in a situation where, after only a few iterations, a single particle holds the majority of the weight with the rest having negligible weight. This results in wasted computational effort on particles which eventually contribute very little to the filter estimation.

```
Input: $X_{k-1}$, $\mathbf{z}_k$
Output: $X_k$
1  $X'_k \leftarrow \emptyset$;
2  foreach $\mathbf{x}_k^{[i]} \in X_{k-1}$ do
3      sample $\mathbf{x}_k^{[i]} \sim p(\mathbf{x}_k \mid \mathbf{x}_{k-1}^{[i]})$;
4      $\omega_k^{[i]} \leftarrow p(\mathbf{z}_k \mid \mathbf{x}_k^{[i]})$;
5      $X'_k \leftarrow X'_k \cup \{(\mathbf{x}_k^{[i]}, \omega_k^{[i]})\}$;
6  end
7  $X_k \leftarrow \emptyset$;
8  for $i \leftarrow 1$ to $|X'_k|$ do
9      draw $r \sim U[0 : \sum \omega_k]$;
10     $j, sum \leftarrow 0$;
11     while $sum$ ¡ $r$ do
12         $s \leftarrow sum + \omega_k^{[j]}$;
13         $j \leftarrow j + 1$;
14     end
15     $X_k \leftarrow X_k \cup \{\mathbf{x}_k^{[j]}\}$;
16 end
```

Algorithm 1: Particle Filter with Resampling

Resampling is a statistical technique which can be used to combat the degeneracy problem. Resampling involves eliminating particles with small weights in favor of those with larger weights. This is achieved by creating a new set of particles by sampling with replacement from the original particle set according to particle weights. Particles with a higher weight will, therefore, have a higher chance of surviving the selection process. One of the implications of the resampling step is the loss of diversity amongst particles as the new particle set most likely contains many duplicates.

Algorithm 1 gives an overview of the particle filter algorithm with resampling. The first for loop (lines 2 through 6) generates, for each particle i, state $\mathbf{x}_k^i$ based on $\mathbf{x}_{k-1}^i$ (line 3) and assigns a weight according to the measurement $\mathbf{z}_k$ (line 4). The second for loop (lines 8 through 16) transforms the particle set X_k' into a new set X_k by resampling according to the weights. On line 9, a uniformly distributed random number r is drawn from the interval $[0, \sum \omega_k]$. By calculating the prefix sum of the weights in the inner while-loop (lines 11 through 14), the randomly drawn weight is mapped to an actual particle, which is subsequently added to the new set. This ensures that the selection likelihood of each particle in each round is proportional to its weight. So the resampling step resets the weights for the whole particle population.

To evaluate the model on enough particles, deliver a global estimate and maintain a healthy particle variance, we need to select hardware platforms that provide enough processing capacity and adapt the filter design to it. Particle filtering has many independent operations, but also needs some global coordination. In the next section, we briefly introduce the many-core architectures and what software for them needs to run efficiently.

3 Many-Core Hardware Platforms

We briefly describe many-core hardware platforms, consisting of GPGPUs and multi-core CPUs.

3.1 General-Purpose GPUs (GPGPUs)

NVIDIA released CUDA [18] in February 2007 to program its GPGPUs. A consortium under the Khronos Group standardized OpenCL [14] in November 2008 to program any (multi-core) processor. The CUDA and OpenCL hardware/software platforms present the same virtual platform with a host and device side. Their APIs offer equivalent functionality with some minor differences in (not yet standardized) vendor-specific extensions. The host runs on the CPU to manage device memory, initiate data transfers and launch kernels. The Bulk Synchronous Parallel model of alternate execution and communication+barrier phases is the basic model of execution. It can be loosened by allowing data transfers to overlap kernel execution. The device consists of an array of streaming multi-processors (SM) to execute kernels and is connected to off-chip, device-wide shared ("global") memory. For GPGPUs, it is important to access global memory in a "coalesced" way, because then the hardware can combine multiple requests to extract high bandwidth. Each SM contains a data parallelism oriented set of processing units that runs up to a few work groups of threads concurrently. It also features a large register file and local memory (cache and/or scratch pad), both partitioned over work groups at kernel launch time. Data in local memory is stored interleaved over multiple banks, so efficient accesses avoid serializing "bank conflicts". An SM executes vector instructions in SIMT fashion (single instruction, multiple threads). SIMT supports scatter/gather and a thread runs in each vector lane that can independently diverge on branches (though diverging often costs performance).

In contrast to CPUs, GPUs are designed to reach high throughput for massively parallel workloads. They feature more computing resources and global memory bandwidth than CPUs, but require much more (esp. spatial) locality of reference and less control flow to maintain pace.

3.2 Multi-Core CPUs

Multi-core CPUs have 2–8 cores. Some execute up to four hardware threads per core. An extensive cache hierarchy must keep all cores fed with typically around 2x 32 kB L1, 256 kB L2 and many MBs L3 cache. Higher cache levels are shared by increasingly more cores. Multiple channels of main memory are connected directly to each CPU (NUMA). If cores are idle, shared resources are fully used by active cores, and given enough thermal headroom, clock frequencies are increased dynamically.

Coalesced accesses, bank conflicts and branch divergence are less important for CPUs. The maximum power drain of a high-end CPU is around 100 W, so one GPU tends to use somewhat more power than a *dual* CPU configuration.

4 Parallel PF Implementation

4.1 Particle Filter Design for Many-Cores

The particle filter algorithm, as described in Algorithm 1, consists of three phases: (i) Sampling (prediction), (ii) Importance weight calculation (update), and (iii) Resampling. The first two phases apply independent functions to each particle. Vector processing may or may not be straightforward; that depends on the complexity of the model and the vector instruction set. Delivering a single estimate and resampling require coordination between different (groups of) particles.

```
Input: $X_{k-1}$, $\mathbf{z}_k$, $num\_particles$, $num\_filters$
Output: $X_k$
for $n \leftarrow 1$ to $num\_filters$ do
    $X'_k \leftarrow \emptyset$;
    for $i \leftarrow 1$ to $num\_particles$ do
        sample $\mathbf{x}_k^{[n][i]} \sim p(\mathbf{x}_k \mid \mathbf{x}_{k-1}^{[n][i]})$;
        $\omega_k^{[n][i]} \leftarrow p(\mathbf{z}_k \mid \mathbf{x}_k^{[n][i]})$;
        $X'_k \leftarrow X'_k \cup \{(\mathbf{x}_k^{[n][i]}, \omega_k^{[n][i]})\}$;
    end
    sort $X'_k$ according to weight;
    calculate local/global estimate;
    foreach neighboring filter $q$ do
        for $t \leftarrow 1$ to $num\_transfer$ do
            send $(\mathbf{x}_k^{[n][t]}, \omega_k^{[n][t]})$ to neighbor $q$;
        end
    end
    $X_t \leftarrow \emptyset$;
    for $i \leftarrow 1$ to $num\_particles$ do
        draw $r \sim U[0 : \sum \omega_k^{[n]}]$;
        $j, sum \leftarrow 0$;
        while $sum$ ¡ $r$ do
            $s \leftarrow sum + \omega_k^{[n][j]}$;
            $j \leftarrow j + 1$;
        end
        $X_k \leftarrow X_k \cup \{\mathbf{x}_k^{[n][j]}\}$;
    end
end
```

Algorithm 2: Distributed Particle Filter

Our design is based on a fully distributed scheme [23]. The idea is to construct a network of smaller particle filters ("sub-filters"). Instead of communicating estimates or other aggregate data, these sub-filters exchange only a few representative particles each round. Such an approach is inherently more scalable, as instead of scaling the *size* of each sub-filter, we can scale the *number* of sub-filters in the network. Each sub-filter runs the algorithm as described in Algorithm 2.

The particle sub-filters can then be distributed over available CPU cores or GPGPU SMs, while keeping enough independent work within sub-filters to utilize any SIMD unit. Depending on the underlying memory or communication architecture, a suitable network topology can be chosen for efficient (particle) exchange between computational units. Figure 1 depicts a number of possible network topologies.

Another design choice for a distributed particle filtering network is how to determine the global estimate out of all particles that represent the probability density function of the modeled state. We know that the worst local winning estimate in such a network may still be close to the best local winning estimate [23]. So depending on the communication costs, it may or may not pay off to take all local winners into account.

As discussed in Section 2, resampling reduces the variation amongst the particle population. Resampling too often can, however, lead to a significant loss of variation and, therefore, estimation accuracy. Particle filter literature [1] suggests calculating a certain metric regarding the particle variation and performing resampling only when it falls bellow a predetermined threshold value. While this works, it is better for *real-time* estimation to perform some resampling every round rather than full resampling when really needed. We propose a simple resampling scheme with parameter $r \in [0, 1]$ indicating the fraction of sub-filters that perform the resampling step each round. This can be implemented by having each sub-filter draw independently $u \sim \mathcal{U}(0, 1)$ and only perform its resampling step if $u < r$.

Parameter	Symbol
Number of particles per sub-filter	m
Number of sub-filters	N
Exchange network topology	T
Number of particles per exchange	t
Resampling ratio	r

Table 1: Distributed Particle Filter Parameters

While traditional centralized particle filters are characterized only by the total number of particles, the

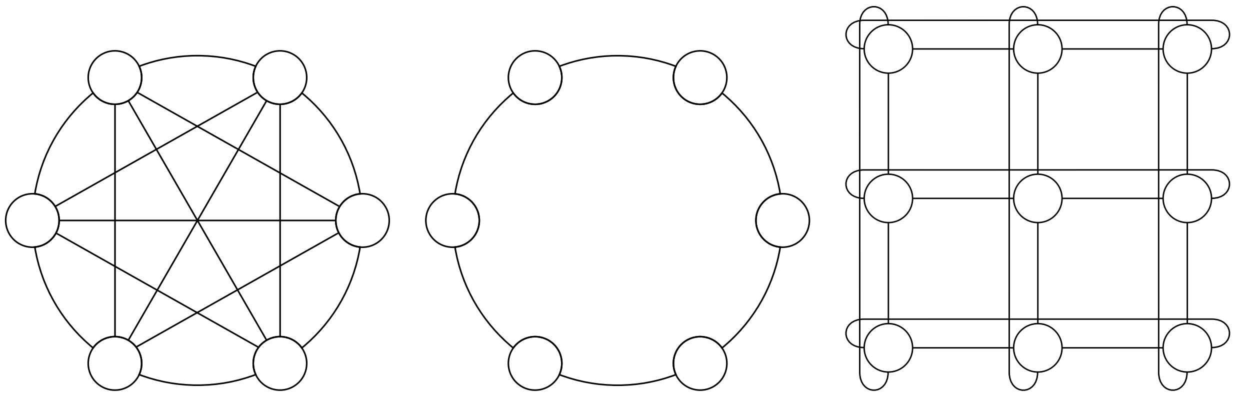

(a) All-to-All (b) Ring (c) 2D Torus

Figure 1: Considered exchange topologies

behavior of our distributed particle filter depends on more parameters, listed in Table 1.

4.2 CUDA and OpenCL Implementations

We have implemented a distributed particle filter algorithm using CUDA (for NVIDIA GPUs) as well as OpenCL (supported by NVIDIA and AMD GPUs and multi-core CPUs). Because of the similarity of both programming models the implementations are quite similar.

Device to host bandwidth is often a bottleneck for GPGPU programs. Unlike some previous studies, *all* our distributed particle filter operations described in Algorithm 2 are executed on the CUDA/OpenCL device. This means that our particle data stays on the device and only measurement data and estimates are transferred.

We have opted to map our particle filter operations to CUDA/OpenCL, such that each particle is represented by a single work item, and each sub-filter by a single work group. In this way, the steps that require a lot of communication are limited to a single work-group and we can explicitly utilize fast, local memory and synchronization to efficiently work cooperatively in parallel.

To maximize attained memory bandwidth, we need to ensure that as many global memory transfers as possible are coalesced. If particle data in global memory is more than 16 bytes, transferring in *Structure of Arrays* (SoA) format will not result in coalesced transfers, so we store it in the *Array of Structures* (AoS) format.

In a few kernels where we cannot have very efficient reads, it can be beneficial to pack individual elements of the particle data into larger, aligned structures. This reduces the number of inefficient memory operations on some platforms. For more information on optimizing CUDA/OpenCL programs, see the vendor's optimization manuals [17, 13].

Based on the details of Algorithm 2 we implemented the following computational kernels.

1. Pseudo-Random Number Generation
2. Sampling and Importance weight calculation
3. Local Sorting
4. Global Estimate
5. Particle Exchange
6. Resampling

4.2.1 Pseudo-Random Number Generation

Particle filters rely heavily on (pseudo-random) number generators (PRNGs). Mersenne Twister [16] is a widely used PRNG, characterized by a large period, good test results and an inspiring name. But a PRNG running on many-cores must be able to generate many uncorrelated sequences. To provide this, MTGP [22] was developed as an MT variant optimized for CUDA. We ported MTGP to OpenCL, added a Box-Muller transformation to get a normal distribution, and used it for all our experiments as a separate kernel. MT/MTGP does need substantial computations and state. For GPUs this matters; ideally, MTGP would be directly usable from the sampling and resampling kernels, but then their static resource size increases, allowing fewer concurrent threads per SM.

4.2.2 Sampling and Importance weight calculation

The sampling step involves generating new particles $\mathbf{x}_k^{[m]}$ from the previous particles $\mathbf{x}_{k-1}^{[m]}$. Numbers just generated are used to generate samples from the state transition distribution. The importance weight calculation assigns weights to each particle using measurement data. We can combine sampling and importance weight calculation in one kernel, as measurement data is available at the start of each round.

4.2.3 Local Sorting

Each sub-filter needs to sort its particles according to their weights for the next steps. We use a bitonic sort, which executes a fixed sequence of parallel comparisons and has a complexity of $O(log^2(n))$. The particle data is usually too large to fit in local memory. Therefore, we sort weights and keep track of the permutation using an index array that we both store locally. To sort the data in global memory, we apply the index array and prefer non-contiguous reads over the more expensive non-contiguous writes.

4.2.4 Global Estimate

To output a global estimate, we perform a parallel reduction on all particles. The reduction operator can compute the particle with the highest weight, a weighted average, or any other associative operator suitable for the application. We select the particle with the highest weight and since we just sorted locally, we only perform the last reduction round(s).

4.2.5 Particle Exchange

The exchange topology (T) determines which sub-filters exchange particles. It is important to realize that on our platforms, all exchanges go through *globally shared memory* (possibly cached). We have implemented the following topologies: (i) All-to-All, (ii) Ring, and (iii) 2D Torus. For All-to-All, the parameter t indicates the number of particles that each sub-filter supplies. Then, all sub-filters read back the same t "best" particles from the supplied set. The Ring and 2D Torus are more distributed in that exchanged particles are unique to (virtually) neighboring sub-filters. There, each sub-filter exchanges t particles with its neighbor.

4.2.6 Resampling

The resampling step generates surviving particle sets locally by drawing randomly from the weighted, local particle set. Implementing this in CUDA/OpenCL requires a parallel prefix sum to compute an array of cumulative sums. We use a bank-conflict avoiding implementation [12]. Then, each thread draws a uniformly distributed number, multiplies it with the sum of the local weights, and performs a binary search to find the highest index with a weight not larger than the drawn number. These indices refer to the surviving particle. Like in the Local Sorting kernel, data is now reordered as efficiently as possible.

This high-performance, scalable resampling algorithm (including distributed exchanges) is novel to the best of our knowledge and takes the estimation and runtime performance of particle filters a clear step forward.

5 Experiments

In this section, we explain our robotic arm application and our test setup, followed by experiments to measure filtering frequency and the effects of varying filtering parameters.

5.1 Robotic Arm Application

To test and benchmark our particle filter, we use the realistic industrial application of a robotic arm. The main reason for this choice is that its measurement equations are highly non-linear and challenging for standard estimation techniques. Another reason is its parametric form. By adjusting the number of joints, we can increase the state dimensions.

In our experiments, the robotic arm has a number of joints N which can be controlled independently. It has one degree of freedom per joint plus the rotation of the base. Each joint has a sensor to measure the angle. The camera at the end is used for tracking an object that is moving on a fixed $x - y$ 2D plane. Figure 2 gives an illustration of this robotic arm.

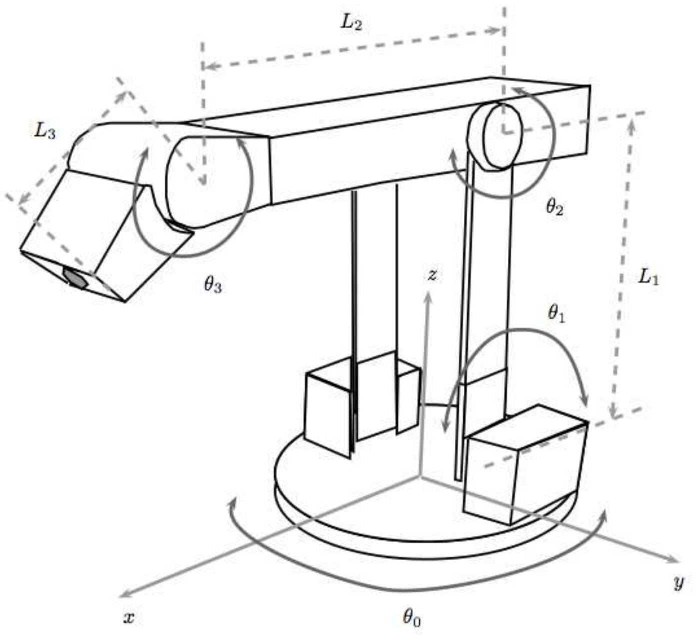


Figure 2: A 3-joint robotic arm with a camera at the end.

Let $\theta_{i,k}$ be the angles of the joint i at the discrete time k ($i = 0$ represents the rotational degree of freedom of the base). Let $(x_k, y_k) \in \mathbb{R}^2$ be the position of the object to be tracked at the discrete time step k in the fixed reference system of the robotic arm, as indicated in Figure 2, while let $(v_{x,k}, v_{y,k}) \in \mathbb{R}^2$ be its velocity. Denote by $\mathbf{x}_k = (\theta_0, \ldots, \theta_N, x_k, y_k, v_{x,k}, v_{y,k})^\top$ the state of the robotic arm and object dynamics. We model the angle dynamics as single integrators and the object dynamics as double integrators as:

$$\theta_{i,k} = \theta_{i,k-1} + h_s u_{i,k-1} + w_{\theta_i,k-1} \quad \forall i \in \{0, \ldots, N\}$$

$$x_k = x_{k-1} + v_{x,k-1} h_s + w_{x,k-1}$$

$$y_k = y_{k-1} + v_{y,k-1} h_s + w_{y,k-1}$$

$$v_{x,k} = v_{x,k-1} + w_{v_x,k-1}$$

$$v_{y,k} = v_{y,k-1} + w_{v_y,k-1}$$

where the terms w model the process noise, u_i is the control action applied to the joints, while h_s is the sampling time. This system of dynamical equations represents our a priori $p(\mathbf{x}_k|\mathbf{x}_{k-1})$.

The camera detects the object in its own frame of reference. Let $(x_{C,k}, y_{C,k})$ be the position of the object in the camera moving frame, which can be related back to the state $\mathbf{x}_k$ via rotations and translations. Let $\hat{\theta}_{i,k}$ be the measured value for the angle of each joint and the base. Let $\mathbf{z}_k = (x_{C,k}, y_{C,k}, \hat{\theta}_{0,k}, \ldots, \hat{\theta}_{N,k})^\top$ be the measurement vector. The measurement equations read as:

$$\begin{bmatrix} x_{C,k} \\ y_{C,k} \end{bmatrix} = \mathbf{h}(\mathbf{x}_k) + w_{C,k} \tag{1}$$

and

$$\hat{\theta}_{i,k} = \theta_{i,k} + w_{\hat{\theta}_i,k}$$

Number of particles/sub-filter (GPGPU)	256
Number of particles/sub-filter (CPU)	128
Number of sub-filters	2048
Exchange network topology	Ring
Number of particles per exchange	1
Resampling ratio	1.0
Number of joints	5
State dimension (#joints + 4)	9
Arm length (meter)	0.5
$w_{\theta_i,k}$	$\mathcal{N}(0, 0.1)$ rad/s
$w_{\hat{\theta}_i,k}$	$\mathcal{N}(0, 0.1)$ rad
$w_{C,k}, w_{x,k}, w_{y,k}$	$\mathcal{N}(0, 0.1)$ m
$w_{v_x,k}, w_{v_y,k}$	$\mathcal{N}(0, 0.1)$ m/s

Table 2: Default filter and model parameters with noise terms

Name	Intel Core **i7-920**	Intel Xeon **X5650**	NVIDIA GeForce **GTX 280**	NVIDIA GeForce **GTX 480**	NVIDIA GeForce **GTX 580**	AMD Radeon **HD 6970**
Platform Type	CPU	dual CPU	GPGPU	GPGPU	GPGPU	GPGPU
#Cores or #SMs	4	2x 6	30	15	16	24
Clock (GHz)	2.67	2.67	1.3	1.4	1.5	0.88
Main Mem. (GB)	8	24	1	1.5	1.5	2
On-chip Mem. (kB)	4x 256, 8192	6x 256, 12288	30x 16, 10x 24	15x 64, 768	16x 64, 768	24x 8, 512
Comp. SP (GFlop/s)	85.4	2x 128.1	933	1345	1581	2703
Memory Bw. (GB/s)	25.6	2x 32.0	141.7	177.4	192.4	176
TDP (Watt)	130	2x 95	236	250	244	250
Runtime Software	Intel OpenCL SDK 1.5		NVIDIA CUDA 4.0			AMD APP 2.6

Table 3: Hardware platforms

for all $i = 0, \ldots, N$. Here $\mathbf{h}(\mathbf{x}_k)$ represents the highly non-linear rotation-translation function, and the terms w are the measurement noise terms. From these measurement equations we derive our a posteriori $p(\mathbf{z}_k|\mathbf{x}_k)$.

The default filter and model parameters with noise terms (the w's) are listed in Table 2. We have adapted the number of particles per sub-filter to the platform, trying to keep them close together without incurring a significant performance penalty. We use these parameters for all tests, unless indicated otherwise.

5.2 Test Setup

Table 3 lists the platforms we used. All floating point computations are in single precision; double precision does not improve our estimation accuracy. We use a *dual* Xeon CPU configuration, so we can compare a GPGPU against a CPU platform with similar power usage (TDP).

5.3 Filtering Runtime Performance

Figure 3 presents the achieved filtering frequency on all platforms. On GPGPUs, we can reach an update frequency of a few hundred Hz with 1 million particles. The dual CPU platform is an order of magnitude

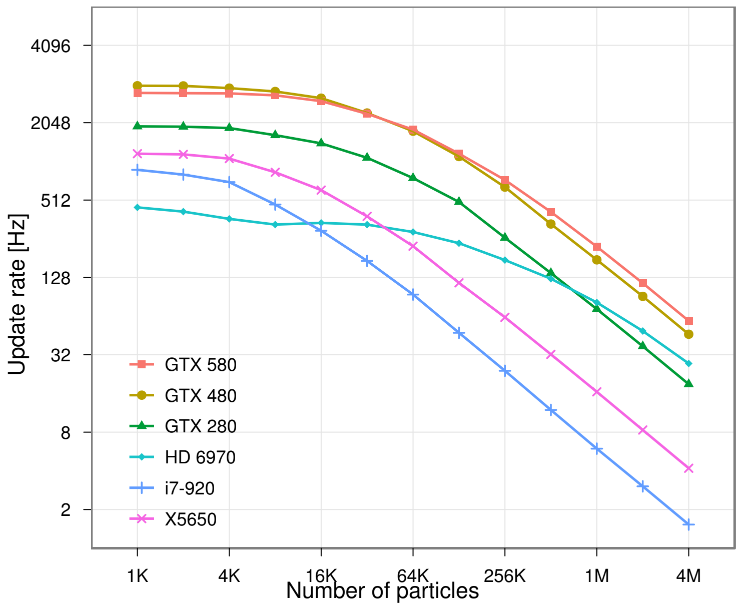
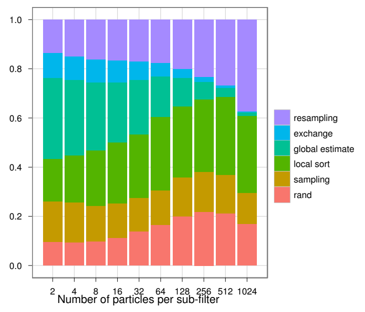
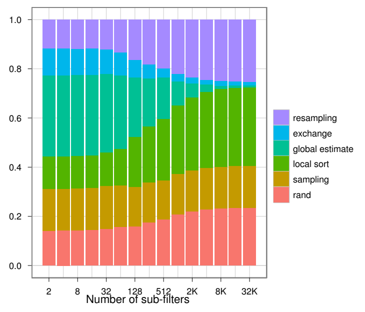


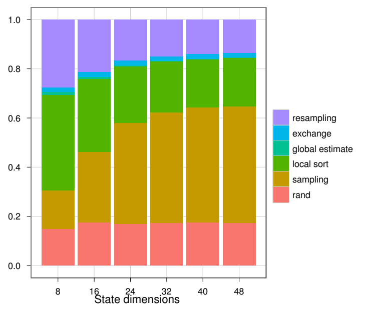
Figure 3: Achieved particle filter frequency



(a) Number of particles/sub-filter



(b) Number of sub-filters



(c) State dimensions

Figure 4: Performance breakdown when scaling various parameters

slower, but still reaches 100 Hz around 64K particles. The HD 6970 GPGPU stays behind even more for small filters, but after 1M particles, it overtakes the GTX 280 (which is 1.5 years older). We also have timed our OpenCL codes on the GTX 580 and that is at most 5% slower than CUDA.

We examine the performance of our filter by scaling the number of particles/sub-filter from 2 to 1024. We now explore the relative performance impact of our filter kernels when scaling in turn the number of particles/sub-filter, the number of sub-filters, and the number of state dimensions. For each experiment, all other parameters stay the same. (The total number of particles does increase for the first two experiments). The plotted breakdowns in Figure 4 have been run on a GTX 580 running CUDA. The differences with other GPGPUs are small. The biggest difference between our dual CPU and GPGPU performance is that the CPU spends much more time on random numbers (40% at 64 particles/sub-filter). This can be attributed to the

PRNG kernel used; (our OpenCL port of) MTGP is highly optimized for (NVIDIA) GPGPUs and apparently performs poorly on CPUs. Checking this further, we found that our OpenCL port of MTGP runs 50% slower on the dual X5650 than "SFMT", the optimized CPU implementation of MT.

5.3.1 Scaling the Number of Particles per Sub-Filter

From Figure 4a, we see that when the number of particles increases, the compute-heavy sorting and resampling stages dominate the runtime at the cost of non-local stages. This is good news, because it means that the filter can be configured to take advantage of higher/lower computation to communication ratios by changing this parameter. Our experiments on the CPU platform confirm this with non-local stages being much cheaper.

5.3.2 Scaling the Number of Sub-Filters

From Figure 4b, we see that when the number of sub-filters increases towards 32K (16M particles), the local operations dominate, but unlike the last experiment, changes appear to be settling down when reaching 32K. Local sort takes most of the time, so we would do well to take a second look at that in future optimizations. When the kernels that have enough computation to overlap long stalls dominate, execution time rises linearly with more sub-filters. The CPU trades in relative time in non-local operations for more time to sort, but the other kernels stay the same with resampling at about 17%.

5.3.3 Scaling the State Dimensions

In this experiment, we scale a model-specific aspect of filtering by testing state dimensions from 8 (4 joints) to 48 (44 joints) variables (4 bytes each). From Figure 4c, we see that when the state dimensions increase to 48, the sampling (with weight calculation) fraction increases to 45% of the runtime at the cost of local sorting and resampling. As the problem becomes more complex, the "filtering" aspect loses relevance and the model implementation more and more determines the runtime. Depending on the model, sampling may be easier to implement efficiently, and certainly to parallelize.

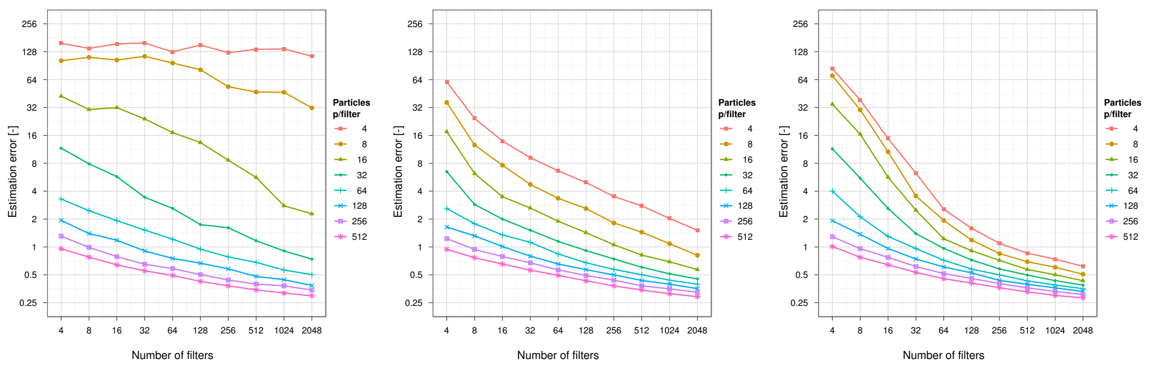
5.4 Estimation Accuracy

After performance, we inspect the impact to accuracy from exchange network topology, number of exchanged particles, and resampling ratio. We scale the number of sub-filters, since that is the principal size scaling direction for distributed particle networks. The presented estimation errors in Figure 5 are the average errors from 100 runs over 100 time steps for each configuration.

5.4.1 Filter Network Size and Topology

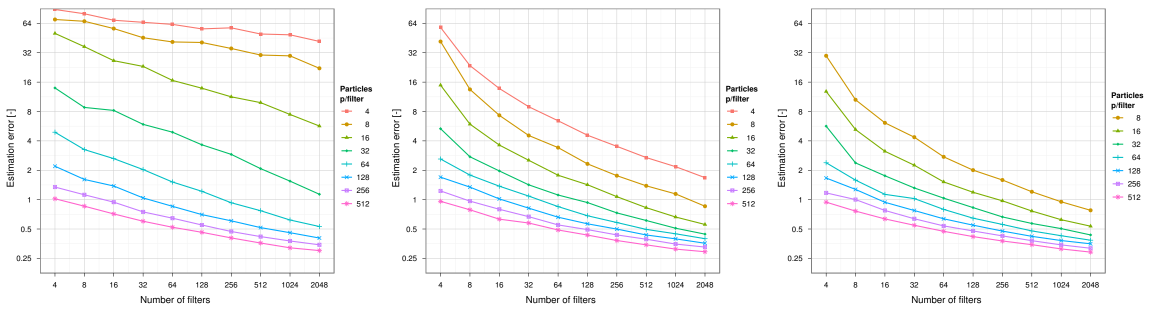
From Figure 5, we can clearly see that the All-to-All topology delivers very poor estimates. This happens as a consequence of a loss of diversity amongst the whole particle population as the same particles are fed into all sub-filters.

The most interesting observation from both the ring and 2D Torus is that in all cases, a low number of particles can be compensated by adding more sub-filters. This confirms our strategy of dividing a large particle filter into a network of smaller sub-filters. We also observe that with a low number of sub-filters, the Ring outperforms the 2D Torus, while with a large number of sub-filters, the 2D Torus outperforms the Ring. The extra exchanges from the 2D Torus result in a loss of diversity in small networks, while it enables faster propagation of more likely particles in large networks.



(a) All-to-All (b) Ring (c) 2D Torus

Figure 5: Estimation error with varying particle filter network size and topology



(a) $t = 0$ (b) $t = 1$ (c) $t = 2$

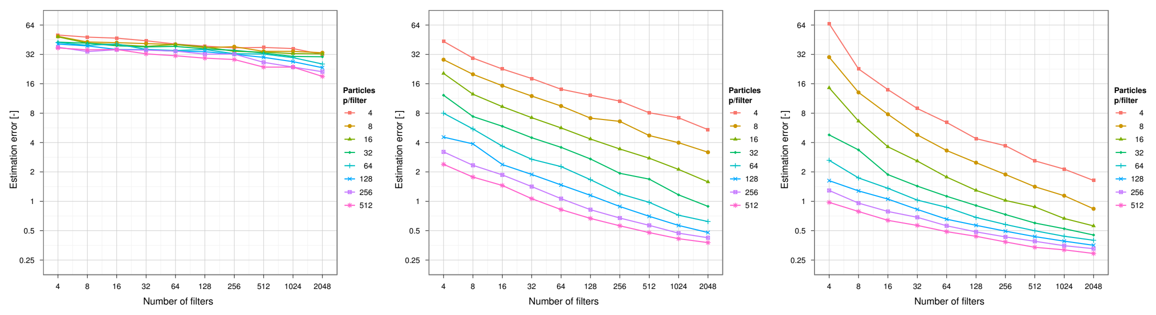
Figure 6: Estimation error with varying number of exchanged particles

5.4.2 Particle Exchange

From Figure 6, we see that the benefit of particle exchange is evident. Exchanging more than one particle offers a minor improvement, but exchanging a single particle seems sufficient for the likely particles to spread. We ran up to $t = 16$ to verify the trend.

5.4.3 Resampling Ratio

In Figure 7, we show the results of resampling never, little, and always. Resampling is indeed an essential step in particle filtering; as expected, adding more sub-filters instead does not work. Even infrequent resampling greatly improves accuracy. The optimum resampling ratio, depends on the network size and application. Lower resampling ratios favor low particle and low filter count settings, while other combinations favor more frequent resampling.



(a) r=0.0 (b) r=0.2 (c) r=1.0

Figure 7: Estimation error with varying resampling ratio

5.5 Discussion

Using our robotic arm application, we attained high filtering rates with large particle sets using GPGPUs. A comparable CPU platform performs slower, but we believe the gap can only be significantly reduced by using CPU specialized kernels.

We analyzed the scaling behavior in three directions and found that for larger particle populations, local operations increasingly take more runtime. However, a large particle population is only needed for complex models with large state dimensions, where ultimately, model-specific sampling and weight calculations become the dominant factor in the total runtime.

We are now convinced that given a proper exchange topology, a network of sub-filters can match a centralized particle filter. Even minimal communication amongst the individual filters is sufficient to spread likely particles throughout the network.

There is not a clearly optimal filtering configuration. The general trend we observed is that in low particle settings infrequent resampling, limited communication and a low connectivity network gives the best results. High particle settings tend to perform better with a more connected network, frequent resampling and increased communication.

6 Evaluation

In this section, we explain how we validated that our particle filter produced expected estimates. We also take one platform as an example to show that our final implementation executes efficiently, and where we could potentially still improve.

6.1 Correctness Validation

We used two techniques to validate our particle filter implementations: (i) Using a ground truth, and (ii) using reference implementations. Initially without any implementation for our model, a particle filter can be verified by checking if it converges to a known series of states under various noise levels and filter configurations. In Figure 8, the red line shows our ground truth as a lemniscate path that starts by heading up from the right side, as would be observed from the camera on our robotic arm. Two particle filters "blue" and "green" start in the center on the right *off* the ground truth. Blue runs with 512x512 particles and converges, but green only uses 16x16 particles causing it to diverge.

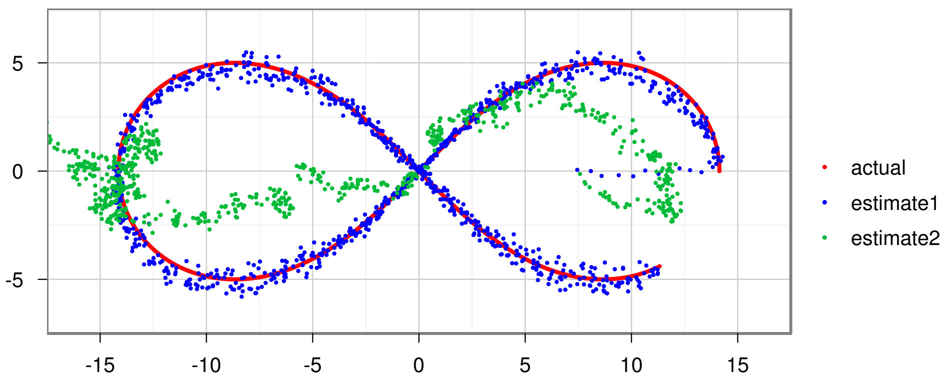


Figure 8: Lemniscate ground truth with two filter traces

We also developed a centralized and a distributed particle filter in Python as sequential reference implementations. These are much easier to implement as intended than optimized CUDA/OpenCL versions. Once tested against ground truths, output from optimized versions can be compared to reference output in more detail.

6.2 Performance Analysis

We explain the used performance model and plot the bounds of the three most intensive kernels on the GTX 480 GPGPU.

6.2.1 Performance Model

In order to estimate how far our performance lies from the optimum, we use the Roofline model [25]. The Roofline model says that at any time, the performance of a program is limited on either the computation or communication side by some upper bound ("roofline"). Which one, depends on the "operational intensity" of the program (in our case, a certain kernel). The operational intensity signifies the computation : communication ratio. It is defined like arithmetic intensity, except that the notion of what is local is moved outwards just beyond the last level of on-chip memory. Thus, to determine the intensity, computational operations and cached accesses are considered "computation", while DRAM accesses are communication. Either side can be bounded by application/platform properties, such as diverging branches or (uncached) uncoalesced memory accesses.

A roofline diagram is kernel and platform specific and is shown with operational intensity on the x-axis and attainable operations/s (e.g. GFlop/s) on the y-axis. The intersection point of the diagonal line (memory bound) and the horizontal line (compute bound) represents the roofline variant of the machine balance. To find the upper performance bound of a kernel, you must draw the rooflines for your kernel/platform, and know the operational intensity of your kernel. (For many CUDA/OpenCL kernels, the intensity can be counted up to a reasonable level; for practical use, hardware counters can help here.) From the intensity, we draw a vertical line upwards that intersects the lowest roofline to find our current upper bound. The intersection point also indicates if your current kernel version is compute or memory bound on this platform.

To calculate upper bounds for our kernels, we use a set of benchmarking kernels to derive attainable memory bandwidth. We calculate the attainable instruction throughput based on available hardware specifications.

6.2.2 Finding Performance Bounds

We select the GTX 480 GPGPU as our platform, because of available hardware information and an instruction throughput benchmark we can adapt and run. Given the performance breakdowns presented earlier, we focus on kernels that matter for performance with large particle populations, thus Sampling, Local Sorting, and Resampling. (We skip the PRNG, as MTGP is already heavily optimized for the GTX 480.)

To determine our memory bandwidth bounds, we need to run a memory benchmark to check the bandwidth for the access patterns we use. We find an attainable bandwidth of 159 GB/s (90% of specified), though we have to proceed without numbers for our irregular reads to reorder particle data.

We determine instruction throughput bounds using micro-benchmarking efforts [26]. This tool was developed for the GT 200 GPGPU series, but it is available online[1], so we can add the missing comparison instructions we need for the Local Sorting kernel and run it on the GTX 480. This effort is also imperfect, because it does not take various (un)known performance bounds into account, but we can proceed.

Having as many bounds as we can quickly uncover, it is time to compute the operational intensities of our kernels. On the memory side, we calculate the total number of bytes transferred and together with the runtime results in the bandwidth our kernels attain. The amount of caching can be estimated for kernels with little temporal locality, or extracted using NVIDIA's profiler. On the computation side, we consider the floating-point addition a single Flop and scale the throughput of other instructions accordingly using the results from the micro-benchmarks. One can count instructions roughly using the program source, or in more detail by looking at the disassembly. To estimate the amount frequently executed sections of non-predicated, data dependent operations, one needs to do a little homework on representative data (or leave it to hardware counters).

We use the filtering configuration from Table 2.

Kernel	Memory Bandw.	Memory Util.	Instr. Throughput	Instr. Util.
Sampling	136.9	84.7%	210.2	15.6%
Local Sorting	40.6	25.1%	124.1	18.5%
Resampling	53.4	33.1%	326.8	48.6%

Table 4: Throughput and utilization on the GTX 480

The calculated throughput and utilization for the three kernels are presented in Table 4. The operational intensity can now be computed (instruction throughput / memory bandwidth) to create the roofline diagram shown in Figure 9.

The Sampling kernel is clearly memory bound and attains close to peak bandwidth, because we store particle data in the structure-of-arrays format. The Local Sorting kernel is also memory bound on the GTX 480. The uncoalesced (vector) read transfers hurt. The Resampling kernel appears compute bound, but without a lower memory bandwidth roof to account for irregular reading, it is hard to say for sure. To verify whether the irregular accesses are responsible (and to effectively set up the required benchmark), we modify both kernels by removing computational sorting and resampling operations (without letting the compiler optimize away memory accesses). On the GTX 480 this reduces the runtime of Local Sorting by 26% and of Resampling by 32%.

We conclude from these results that there are other factors contributing to the gap between the calculated optimum and the achieved throughput for these two kernels. Considering that both kernels extensively use local memory for sorting and for the parallel prefix sum, it is the most likely place to find more inefficiencies.

[1]`http://www.stuffedcow.net/research/cudabmk`

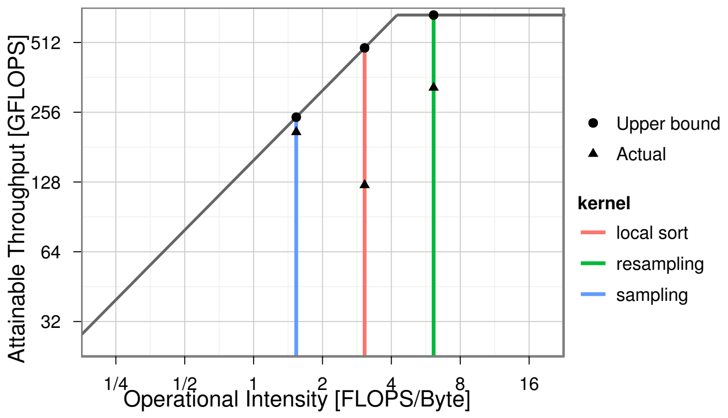


Figure 9: Roofline for three kernels on the GTX 480

7 Related Work

With high computational requirements, researchers have looked into algorithmic, application-specific and platform optimizations such as parallelization to drastically reduce runtime. We are especially interested in parallel particle filtering algorithms, especially if studied for more than one application, or suitable for real-time (instrument) control (i.e. low runtime variance per estimation round).

The basic parallel particle filtering algorithm is to partition the particle population over all processors and to perform sampling and weighting independently for each sub-filter [5]. Local estimate candidates are computed first to obtain a global estimate. The authors find that local resampling can be as accurate as global resampling.

In [2], three methods are proposed to implement distributed particle filters: (i) Global Distributed Particle Filter (GDPF), (ii) Local Distributed Particle Filter (LDPF), and (iii) Compressed Distributed Particle Filter (CDPF). With GDPF, only the sampling and weight calculation steps run in parallel, while resampling is performed centrally. LPDF is comparable to the basic parallel algorithm [5] where resampling is performed locally without communication. CDPF, similar to GDFP, does resampling centrally, but uses only a small representative set of particles for global resampling. The results are sent back to each individual node. The paper concludes from a number of simulations that LDFP provides both better estimation and performance.

Two distributed resampling algorithms are proposed in [4]: (i) Resampling with Proportional Allocation (RPA), and (ii) Resampling with Non-proportional Allocation (RNA). RPA involves a two-stage resampling step (local+global), while RNA involves local resampling followed by a particle exchange step (like we do). These algorithms still involve a certain degree of centralized planning and information exchange. RPA provides a better estimation, while RNA has a simpler design. In later work [3], they compare a standard particle filter with a Gaussian particle filter[2] on an FPGA. The presented results indicate that the Gaussian particle filter, while being faster than a standard particle filter, is equally accurate for (near-)Gaussian problems.

Some of these algorithms (GDPF, RNA, RPA, Gaussian particle filter) are compared using a parallel implementation on a multi-core CPU [21]. The comparison goes only until 10K particles. Nevertheless, the Gaussian particle filter outperforms all other algorithms in Gaussian estimation problems. RNA achieves near linear speedup with respect to the number of cores, which is much better than the other non-Gaussian filters.

With the introduction of GPGPU, new speedup levels have come within reach. Various tracking applications in computer vision [15, 19, 9, 7, 10] have benefited from using CUDA.

[2]Gaussian particle filters are a special kind of particle filter which approximate the posterior with a normal distribution. They do not require a resampling step.

The most recent study we could find investigates a particle filter for localization and map matching for vehicle applications [20] on a CPU using OpenMP and on a GPU using CUDA. The state dimension is only four values and does not benefit from more than 32K particles, but the application is nevertheless an interesting and well explained case for a particle filter. Experimentation shows that with 128K particles, a CPU is 4.7x faster on six cores and that a GPU is another 16x faster. While they do partition all particles over all cores, the GPU code runs resampling on the CPU, but only if particle variance is above a threshold. This means that they need to compute the *global* variance and perform one large resampling step, but do not exchange particles between (groups of) threads. It also means that they can only keep the amount of host-device transfers minimal for rounds where no resampling takes place. When resampling is needed, however, particle weights are transferred to the host and surviving particles are transferred back. This scheme may be fast on average, but it increases the maximum computation time in some rounds, which is undesirable for real-time processing. Their execution breakdown shows that resampling takes ¡1% on 128K particles, but that breakdown is for the sequential CPU version.

None of the studies found scale the number of particles as far as we do, because their systems are not complex enough to benefit from more particles. We also investigate the effects of other parameter than number of particles and performance behavior much more thoroughly. Moreover, we report results on a wider range of platforms than ever before.

8 Conclusions and Future Work

The particle filter is a powerful, but computationally demanding Monte Carlo based Bayesian estimation method. With the introduction of massively parallel many-core processors, particle filtering has become viable, but the design intricacies of the algorithm remained elusive.

For this paper, we investigated various particle filter algorithms and implemented two applications on various hardware platforms using CUDA and OpenCL. We identified the parameters that strongly affect the behavior of our filter and experimentally quantified their scaling effects on estimation quality and runtime. For the random number generation and particle exchange functions, we implemented several alternatives. In addition, we extensively analyzed the performance of our NVIDIA GPU implementation, both analytically and empirically. The analytical model used is based on comparing obtained performance with calculated upper bounds for our application.

Our experiments indicate that real-time particle filtering for complex estimation problems is feasible on current generation GPGPUs and, to some extend, on multi-core CPUs. For small estimation problems with up to four state variables, we can reach kHz estimation rates. For our robotic arm application with 48 state variables (many joints) utilizing 1 million particles, we pushed our GPGPUs to attain estimation rates of 100–200 Hz. Given many-core processor trends, it is important to use a design that effectively combines more (and not larger) sub-filters. The All-to-All particle exchange topology may seem an intuitive choice for a shared memory system, but it decreases particle diversity too much. Above all, its runtime fraction is very low. We discovered that topologies with a lower connectivity are preferable *for accuracy reasons*. The ring performs better with a small number of sub-filters and the added connectivity of the 2D Torus compares favorably with a large number of sub-filters. Other discovered rules of thumb are that accuracy can improve a lot with only infrequent resampling and exchanges of only one particle. Conversely, performance is wasted when exchanging a lot of particles or by resampling often.

As for multi-core CPUs with similar power usage (dual), OpenCL particle filter performance is about a factor 20–25 slower than a CUDA GPGPU. We believe that this gap can be narrowed a lot, but that requires writing platform optimized kernels, such as with the Mersenne Twisters variants that we ran. Note that the filtering performance for particular estimation problems also depends on problem-specific characteristics.

We see two interesting directions for future work. The first direction is a matter of scale: down to real-time applications on embedded systems (possibly with GPU cores), or up to take advantage of CPU or GPU clusters.

Each platform scale direction will present new challenges to performance portability.

Another direction is to focus on applications with different types of estimation problems. We expect to gain an even better understanding of the particle filter design with data available from different estimation problems.

References

[1] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian tracking. *IEEE Trans. on Signal Processing*, 50(2):174–188, Feb 2002. 3, 6

[2] A. S. Bashi, V. P. Jilkov, X. R. Li, and H. Chen. Distributed Implementations of Particle Filters. In *Proc. of the 6th Int'l Conf. of Information Fusion*, pages 1164–1171, July 2003. 17

[3] M. Bolić, A. Athalye, S. Hong, and P. Djurić. Study of Algorithmic and Architectural Characteristics of Gaussian Particle Filters. *Journal of Signal Processing Systems*, 61(2):205–218, Nov 2010. 17

[4] M. Bolić, P. M. Djurić, and S. Hong. Resampling Algorithms and Architectures for Distributed Particle Filters. *IEEE Trans. on Signal Processing*, 53(7):2442–2450, Jul 2005. 17

[5] O. Brun, V. Teuliere, and J. M. Garcia. Parallel Particle Filtering. *Journal of Parallel and Distributed Computing*, 62(7):1186–1202, July 2002. 17

[6] A. Doucet, S. Godsill, and C. Andrieu. On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, 10:197–208, July 2000. 3

[7] J. F. Ferreira, J. Lobo, , and J. Dias. Bayesian real-time perception algorithms on GPU. *Journal of Real-Time Image Processing (Special Issue)*, 6(3):171–186, 2010. 17

[8] T. Flury and N. Shephard. Bayesian inference based only on simulated likelihood: particle filter analysis of dynamic economic models. *Econometric Theory*, 27(05):933–956, May 2011. 1

[9] R. M. Friborg, S. Hauberg, and K. Erleben. GPU accelerated likelihoods for stereo-based articulated tracking. In *the ECCV Workshop on Computer Vision on GPUs*, Sep 2010. 17

[10] M. A. Goodrum, M. J. Trotter, A. Aksel, S. T. Acton, and K. Skadron. Parallelization of particle filter algorithms. In *Proc. of 3rd Workshop on Emerging Applications and Many-core Architecture*, June 2010. 17

[11] N. J. Gordon, D. J. Salmond, and A. F. M. Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *Radar and Signal Processing*, 140(2):107–113, Apr 1993. 3

[12] M. Harris, S. Sengupta, and J. D. Owens. Parallel Prefix Sum (Scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*, chapter 39. Addison-Wesley Professional, Aug 2007. 8

[13] Intel. *Writing Optimal OpenCL Code with Intel OpenCL SDK.* Intel Corp., Oct 2011. 7

[14] Khronos OpenCL Working Group. *The OpenCL Specification*, Nov 2011. 4

[15] O. M. Lozano and K. Otsuka. Real-time visual tracker by stream processing. *Journal of Signal Processing Systems*, 57(2):285–295, 2009. 1, 17

[16] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, Jan 1998. 7

[17] NVIDIA. *CUDA C Best Practices Guide.* NVIDIA Corp., Santa Clara, CA, USA, May 2011. 7

[18] NVIDIA. *NVIDIA CUDA Programming Guide.* NVIDIA Corp., Santa Clara, CA, USA, May 2011. 4

[19] K. Otsuka and J. Yamato. Fast and robust face tracking for analyzing multiparty face-to-face meetings. In *Proc. of the 5th Int'l Workshop on Machine Learning for Multimodal Interaction*, volume LNCS 5237, pages 14–25. Springer, Sep 2008. 17

[20] K. Par and O. Tosun. Parallelization of particle filter based localization and map matching algorithms on multicore/manycore architectures. In *Proc. of the IEEE 2011 Intelligent Vehicles Symposium*, pages 820–826. IEEE, June 2011. 1, 18

[21] O. Rosén, A. Medvedev, and M. Ekman. Speedup and Tracking Accuracy Evaluation of Parallel Particle Filter Algorithms Implemented on a Multicore Architecture. In *2010 IEEE Int'l Conf. on Control Applications*, pages 440–445, Sep 2010. 17

[22] M. Saito. A Variant of Mersenne Twister Suitable for Graphic Processors. `http://arxiv.org/abs/1005.4973`, June 2010. 7

[23] A. Simonetto and T. Keviczky. Recent Developments in Distributed Particle Filtering: Towards Fast and Accurate Algorithms. In *1st IFAC Workshop on Estimation and Control of Networked Systems*, Sep 2009. 5, 6

[24] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics.* Intelligent robotics and autonomous agents. The MIT Press, Sep 2005. 2

[25] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Comm. of the ACM*, 52(4):65–76, Apr 2009. 15

[26] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU Microarchitecture through Microbenchmarking. In *2010 IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pages 235–246, Mar 2010. 16