Adapting Particle Filter Algorithms to the GPU Architecture

Mehdi Chitchian

Adapting Particle Filter Algorithms to the GPU Architecture

Mehdi Chitchian

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Computer Science

Parallel and Distributed Systems Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Thesis Committee:

Chair	prof. dr. ir. Henk J. Sips	Faculty EEMCS, TU Delft
Supervisor	drs. Alexander S. van Amesfoort	Faculty EEMCS, TU Delft
Member	dr. ir. Tamas Kevizcky	Faculty 3mE, TU Delft
Member	dr. Alexandru Iosup	Faculty EEMCS, TU Delft

© 2011 Mehdi Chitchian. All rights reserved.

mehdi.chitchian@gmail.com

Abstract

The particle filter is a Bayesian estimation technique based on Monte Carlo simulations. The non-parametric nature of particle filters makes them ideal for non-linear non-Gaussian systems. This greater filtering accuracy, however, comes at the price of increased computational complexity which limits their practical use for real-time applications.

This thesis presents an attempt to enable real-time particle filtering for complex estimation problems using modern GPU hardware. We propose a GPU-based generic particle filtering framework which can be applied to various estimation problems. We implement a real-time estimation application using this particle filtering framework and measure the estimation error with different filter parameters. Furthermore, we present an in-depth performance analysis of our GPU implementation followed by a number of optimisations in order to increase implementation efficiency.

Contents

List of Figures

List of Tables

1 Introduction

Many dynamical systems operate under various degrees of uncertainty. This uncertainty could stem from an imperfect understanding of the underlying dynamics, noisy observations and/or actuation or numerous other factors. Bayesian estimation, a probabilistic approach, captures this uncertainty in the form of a probability distribution function for the state of the dynamical system.

Tractable closed-form solutions for the Bayes filter exist only for linear Gaussian systems in the form of the Kalman filter. Non-linear Gaussian systems can still use variants of this filter (e.g. extended Kalman filter, unscented Kalman filter) depending on the amount of non-linearity. The particle filter, based on Monte Carlo simulations, is a non-parametric filter which can be used for state estimation in non-linear non-Gaussian systems. Although much more powerful than extended/unscented Kalman filters, particle filters need many particles, and thus computational resources, in order to produce an accurate result for complex estimation problems. These high computational demands limit their practical use for real-time applications.

In recent years, Graphics Processing Units (GPUs) have turned into massive parallel processors with high computational capacity. With the introduction of programmable shaders in the beginning of the last decade, followed by the move towards a unified shading architecture and the arrival of general purpose programming languages targeting GPUs in the latter half of the decade, GPUs have been increasingly used in many different fields beyond computer graphics.

In contrast to many other multi-core platforms, GPUs are specifically designed for certain workloads typically encountered in computer graphics rendering. These workloads are highly parallel, with the overall throughput being favoured over the latency of individual tasks. Memory access latencies are hidden through massive threading and rapid context switching. Therefore, the use of GPUs for a particular application depends on the presence of a significant amount of (fine grained) parallelism which might require algorithm changes or even a complete redesign.

This thesis presents a comprehensive study on the issues regarding the design and implementation of particle filters for modern GPUs, the required algorithmic changes and its implications for the filter accuracy. A GPU-based generic particle filtering framework is proposed which can be applied to various estimation problems. We will use two real-life applications in order to examine both the generic as well as the model-specific parts of the implementation. Furthermore, we determine the effects of the various filter parameters on the estimation quality and analyse the performance of implementation on two GPU platforms.

1.1 Objectives

The central research questions addressed in this thesis can be formulated as:

> Can particle filters be *efficiently* implemented on modern GPUs for complex real-time estimation problems? What, if any, modifications are needed to the particle filter algorithm for parallel execution? How do these modifications influence the filter behaviour?

In this context, efficiency refers to the utilisation of the available hardware resources (e.g. memory throughput). In order to answer these questions, we (i) study the particle filter algorithm to identify parallelism opportunities and limitations, (ii) explore the modern GPU architecture and its programming model, and (iii) use a real-life estimation problem to measure the filter accuracy and implementation efficiency.

1.2 Contributions

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

- Based on [Simonetto and Keviczky, 2009], we present a fully distributed particle filtering scheme. Without making any particular assumptions on the underlying hardware architecture, this scheme divides the "classical" particle filter into a network of smaller particle filters running concurrently. These particle filters exchange only a few representative particles from their respective particle population.
- Following this distributed scheme, we introduce a GPU-based 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 two GPU platforms.
    - In order to better understand and model certain GPU performance characteristics, we extend an existing microbenchmark suite. This allows us to reason better about the performance of our implementation.
    - We calculate the effective utilisation of the hardware resources in terms of instruction throughput as well as memory bandwidth.
    - We identify the performance bottlenecks of the various steps of the algorithm.
    - We analyse the scalability of the presented implementation in two directions: (i) increasing filter size, and (ii) increasing state dimension.
- With a particle filter implementation for a real-life estimation problem, we present an in-depth analysis of the filter estimation quality under varying filter parameters.

1.3 Outline

The remainder of this thesis is organised as follows. We begin in Chapter 2 with an introduction of the Bayesian estimation framework, with an emphasis on particle filters. Chapter 3 presents our approach to distributing the particle filter over many computation units based on an earlier work. In Chapter 4, we take a look at the modern GPU architecture and review its programming model. We discuss two real-life estimation problems in Chapter 5 which can benefit from the accuracy of particle filters. The actual GPU implementation of the generic particle filtering framework as well as the model-specific code are discussed in Chapter 6. Chapter 7 presents an in-depth performance analysis of the filter implementation on two GPU platforms followed by a number of optimisation steps. We evaluate the filter accuracy under varying parameters in Chapter 8 using one of the applications discussed earlier. Finally, Chapter 9 concludes this thesis with a summary of the presented work and a look into possible future research directions.

Part I

Background

2 Recursive Bayesian Estimation

The problem of estimating the state of a dynamical system through noisy measurements has many applications in science. Many of the quantities which constitute the state of a system cannot be observed directly, but need to be *inferred* from noisy sensor data. *Bayesian* estimation is based on the assumption that this uncertainty should be represented by probabilities [West and Harrison, 1997]. A *probabilistic* state estimation computes a *Probability Density Function* (PDF) for the state over the range of possible values.

First, in Section 2.1, we will describe the generic Bayesian filtering framework. Next we will discuss the Kalman filter in Section 2.2, and the Particle filter in Section 2.3. The material for this chapter is based on [Thrun et al., 2005] and [Arulampalam et al., 2002].

2.1 Bayesian Filtering

Suppose x is a quantity which we wish to infer from the measurement z. The probability distribution $p(x)$ represents all the knowledge we have regarding this quantity *prior* to the actual measurement. This distribution is therefore called the *prior probability distribution*. The conditional probability $p(x \mid z)$, called the *posterior probability distribution*, represents our knowledge of x having incorporated the measurement data. This distribution, however, is usually unknown in advance as a result of the complex dynamics involved in most systems. *Bayes rule* allows us to calculate a conditional probability based on its inverse:

$$p(a \mid b) = \frac{p(b \mid a)\ p(a)}{p(b)}$$

The inverse probability $p(z \mid x)$ directly relates to the measurement characteristics.

In order to discuss how the Bayes filter calculates the state estimate, we first need to model the dynamics of the system. Let x_t denote the state at time t, and z_t denote the set of all measurements acquired at time t. The evolution of the state is governed by the following conditional probabilistic distribution:

$$p(x_t \mid x_{t-1}, \ldots, x_0, z_{t-1}, \ldots, z_1)$$

Assuming the system exhibits Markov properties[1], the state x_t depends only on the previous state x_{t-1}. Therefore, we can rewrite the above distribution simply as:

$$p(x_t \mid x_{t-1})$$

The above probability is also referred to as the *state transition probability*. The measurements of the state follow the probability distribution $p(z_t \mid x_t)$ which is called the *measurement probability*.

The Bayes filter calculates the estimate of the state recursively in two steps:

[1] We maintain this assumption throughout this chapter.

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 timestep.

$$p(x_t) = \int p(x_t \mid x_{t-1})\, p(x_{t-1} \mid z_{t-1})\, dx_{t-1}$$

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.

$$p(x_t \mid z_t) = \eta\, p(z_t \mid x_t)\, p(x_t)$$

In order to implement this filter, three probability distributions need to be known in advance:

i. Initial state $p(x_0)$,

ii. Measurement probability $p(z_t \mid x_t)$,

iii. State transition probability $p(x_t \mid x_{t-1})$.

The Bayes filter, in its basic form, is inapplicable for complex estimation problems. The main problem is that the prediction step requires us to be able to integrate over the state transition probability distribution in closed form which is only possible in a restricted number of problems. In the following sections, we will examine two derived filters which are more powerful and can be applied to a wider range of estimation problems.

2.2 Kalman Filter

The Kalman filter is a popular Bayesian filtering technique, first presented in the late '50s and early '60s, independently, by Swerling [Swerling, 1958] and Kalman [Kalman, 1960]. This filter assumes the following properties hold for the system:

- The state variable x_t takes values in a *continuous* space;

- The state transition probability, denoted by $p(x_t \mid x_{t-1})$, as well as the measurement probability, denoted by $p(z_t \mid x_t)$, are *linear* functions with *Gaussian* noise;

- The initial state is *normally* distributed.

The aforementioned properties ensure that the posterior probability remains a Gaussian. From the above we can write the state transition as:

$$x_t = A_t x_{t-1} + \varepsilon_t$$

where A_t is a square matrix of the same dimension as the state vector x_t and ε_t denotes a Gaussian vector with mean 0 and variance R_t representing the uncertainty in the state transition. The state transition probability can be written as:

$$p(x_t|x_{t-1}) = det(2\pi R_t)^{-1/2} e^{-1/2(x_t - A_t x_t)^T R_t^{-1}(x_t - A_t x_t)}$$

The measurement probability can also be written as:

$$z_t = B_t x_t + \delta_t$$

where B_t is a square matrix of the same dimension as the measurement vector z_t and δ_t denotes a Gaussian vector with mean 0 and variance Q_t representing the uncertainty in the measurements. The measurement probability can, therefore, be written as:

$$p(z_t \mid x_t) = det(2\pi Q_t)^{-1/2} e^{-1/2(z_t - B_t x_t)^T Q_t^{-1}(z_t - B_t x_t)}$$

Finally, the initial state with mean μ_0 and variance Σ_0 can be written as:

$$p(x_0) = det(2\pi\Sigma_0)^{-1/2} e^{-1/2(x_0 - \mu_0)^T \Sigma_0^{-1}(x_0 - \mu_0)}$$

One of the main reasons the Kalman filter has gained much popularity lies in its computational efficiency. Each iteration of the Kalman filter is lower bound by $O(k^{2.4} + n^2)$, where k is the dimension of the measurement vector z_t and n is the dimension of the state space. This computational efficiency results from the assumption that the state and measurement functions are linear and operate on Gaussian variables which can be computed in closed form.

2.2.1 Extended and Unscented Kalman Filters

As discussed in the previous section, the Kalman filter imposes two main restrictions on the underlying system: linear state transition and measurement functions and a Gaussian posterior probability distribution. This limits the applicability of the Kalman filter, as most state transition and measurement functions are non-linear. If we relax the first restriction, the state transition and measurement functions become:

$$x_t = g(x_{t-1}) + \varepsilon_t$$

$$z_t = h(x_t) + \delta_t$$

Assuming non-linear functions g and h, the above filter does not have a closed-form solution. One solution is to approximate these functions through *linearisation*.

The extended Kalman filter (EKF) uses *Taylor expansions* to linearise the non-linear state transition and measurement functions. The approximation is a linear function tangent to the target function at the mean of the Gaussian. The accuracy of this estimation depends on the degree of non-linearity.

For highly non-linear systems, the extended Kalman filter does not produce an accurate estimation as it does not preserve the mean and covariance of the Gaussian distribution. The unscented Kalman filter (UKF) uses a deterministic set of sampling points which are propagated through the non-linear functions in order to capture the true mean and covariance of the Gaussian distribution.

Each update step, for both EKF and UKF, requires $O(k^{2.4} + n^2)$ time.

2.3 Particle Filter

Particle filtering [Gordon et al., 1993] 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 mentioned above, particle filters represent the posterior by a set of particles. Each particle $x_t^{[m]}$ can be considered as an instantiation of the state at time t. In the prediction step of the particle filter, each particle $x_t^{[m]}$ is generated from the previous state $x_{t-1}^{[m]}$ by sampling from the state transition probability $p(x_t \mid x_{t-1})$. In the update step, when measurement z_t is available, each particle is assigned a weight $w_t^{[m]}$ according to:

$$w_t^{[m]} = p(z_t \mid x_t^{[m]})$$

```
$X_t \leftarrow \emptyset$
for $i$ in $1 : M$ do
    sample $x_t^{[i]} \sim p(x_t \mid x_{t-1}^{[i]})$
    $w_t^{[i]} \leftarrow p(z_t \mid x_t^{[i]})\, w_{t-1}^{[i]}$
    $X_t \leftarrow X_t + \{x_t^{[i]}, w_t^{[i]}\}$
end for
```

Algorithm 1: Basic Particle Filter (Sequential Importance Sampling, SIS)

A high level description of the basic particle filter algorithm is given by Algorithm 1.

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

2.3.1 The Degeneracy Problem and Resampling

A common problem with the basic particle filter algorithm mentioned in the previous section is the *degeneracy* problem. It has been proven that the that the variance of the weights can only increase over time [Doucet et al., 2000]. This results in a situation where only 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.

Resampling is a statistical technique which can be used to combat the degeneracy problem. Resampling involves eliminating particles with small weights in favour 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.

2.3.2 Algorithm Overview

Algorithm 2 gives an overview of the particle filter algorithm with resampling. The first for loop (lines 2 through 6) generates, for each particle i, state x_t^i based on x_{t-1}^i (line 3) and assigns a weight according to the measurement z_t (line 4). The second for loop (lines 8 through 16) transforms the particle set X_t' into a new set X_t by resampling according to the weights. On line 9, a uniformly distributed random number r is drawn from the interval $[0, \sum w_t]$. 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 likelihood of the selection of each particle, in each round, is proportional to its weight.

Note that in contrast to the previous version, presented in Algorithm 1, no history is maintained for the particle weights as the resampling step resets the weights for the whole particle population.

```
$X'_t \leftarrow \emptyset$
for $i$ in $1 : M$ do
    sample $x_t^{[i]} \sim p(x_t \mid x_{t-1}^{[i]})$
    $w_t^{[i]} \leftarrow p(z_t \mid x_t^{[i]})$
    $X'_t \leftarrow X'_t + x_t^{[i]}$
end for
$X_t \leftarrow \emptyset$
for $i$ in $1 : M$ do
    draw $r \sim U[0 : \sum w_t]$
    $j, s \leftarrow 0$
    while s < r do
        $s \leftarrow s + w_t^{[j]}$
        $j \leftarrow j + 1$
    end while
    $X_t \leftarrow X_t + x_t^{[j]}$
end for
```

Algorithm 2: Particle Filter with Resampling (Sequential Importance Resampling, SIR)

3 Distributed Particle Filtering

We examined a number of Bayesian filtering techniques in Chapter 2 including particle filters. Because of their non-parametric form, particle filters can be applied to non-linear and non-Gaussian systems where (extended/unscented) Kalman filters fall short. However, the required computational effort, especially when a large number of particles is needed for an accurate estimation, limits their practical use in time-constrained applications.

A natural approach would be to distribute the particle filter calculations amongst multiple computation units. This would allow the use of particle filters for complex estimation problems in time critical applications. In this chapter we will discuss the issues regarding the implementation of a distributed particle filter.

We will start with a classification of distributed particle filtering schemes in Section 3.1. In Section 3.2, we will explore the related work, while a discussion on the design of a distributed particle filter follows in Section 3.3. Next, we will review the parameters which affect the filter behaviour in Section 3.4 and, finally, we conclude this chapter in Section 3.5.

3.1 Classification

There are different approaches for implementing distributed particle filters. We will use the classification from [Simonetto and Keviczky, 2009] in order to categorise these approaches:

Distributed Sensing Particle Filters This class of particle filters assumes that there are multiple sensors performing measurements. Each sensor has a particle filter operating only on local measurements. These particle filters exchange limited information in order to achieve a global estimate.

Distributed Computation Particle Filters In contrast to the previous class, the assumption here is that all measurement data is available to all particle filters. Each particle filter is assigned a subset of the total particle population.

We will only focus on the second class of distributed particle filters in this thesis.

3.2 Related Work

Because of the numerous advantages of particle filtering over its parametric counterparts, much research effort has been dedicated to designing a distributed particle filter in order to overcome its inherent computational complexity. In this section we will examine a number of related studies and discuss their main contributions.

The first work we consider is from [Brun et al., 2002] where a data parallel approach is proposed. The particle population is partitioned into several subsets. Each subset is assigned to a single processor. The sampling as well the weight calculations are performed independently for each subset. In order to calculate a global estimate, local estimates are calculated for each subset. These estimates are subsequently aggregated into a global estimate. The authors claim that performing local resampling on each subset does not compromise on the filter accuracy.

In [Bashi et al., 2003], the authors propose three methods for implementing 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 are parallelised while resampling is performed centrally. LPDF is comparable to the proposed solution of [Brun et al., 2002] in which resampling is performed locally without any communication. CDPF, similar to GDFP, has a centralised resampling implementation. However, only a small representative set of particles is used for global resampling. The results are sent back to each individual node. They conclude from a number of simulations that LDFP provides a better estimation while being faster.

Two distributed resampling algorithms are proposed in [Bolić et al., 2005]: (i) Resampling with Proportional Allocation (RPA) and (ii) Resampling with Non-proportional Allocation (RNA). RPA involves a two-stage resampling step (global and local) while RNA involves local resampling followed by a particle exchange step. These algorithms still involve a certain degree of centralised planning and information exchange. RPA provides a better estimation while RNA has a simpler design. In a later work [Bolić et al., 2010], they compare an FPGA implementation of a standard particle filter with that of a Gaussian particle filter[1] The presented results indicate that the Gaussian particle filter, while being faster than a standard particle filter, is equally accurate for (near-)Gaussian problems.

A number of the previously mentioned algorithms (GDPF, RNA, RPA, Gaussian particle filter) are compared in [Rosén et al., 2010] using a parallel implementation on a multi-core CPU. The comparison goes only as far as experiments with 10000 particles. Nevertheless, the Gaussian particle filter outperforms all other algorithms in Gaussian estimation problems. RNA can achieve near-linear speedup with respect to the number of cores, which is much better than the other non-Gaussian filters.

A particle filter implementation on a GPU is presented in [Hendeby et al., 2010]. For the resampling step, the particles are divided into multiple subsets. Each subset is resampled independently and the results are redistributed into different sets (similar to the RNA algorithm). Pseudo-random numbers are generated on the host CPU and transferred to the GPU. This severely impacts the performance of the filter as about 85% of the total runtime is spent on generating pseudo-random numbers and transferring them to the GPU. The GPU implementation is not suitable for real-time estimation for complex problems and is only marginally faster than a CPU implementation.

None of the studies we reviewed in this section have been able to demonstrate a working particle filter implementation for complex real-time estimation problems. The literature we examined (with the exception of [Hendeby et al., 2010]) mention experiments with particle filters running up to 10000 particles. This is insufficient for most complex problems. Some of the presented algorithms divide the particle population into smaller subsets without any form of communication or information exchange except for calculating global estimates. The only reference to a large scale particle filter implementation was by [Hendeby et al., 2010] which mentions experiments with up to a million particles. There is, however, no mention of any real-time application for the proposed particle filter implementation.

3.3 Filter Design

The particle filter algorithm, as described in Algorithm 2, consists of three main stages:

i. Sampling (prediction),

[1] Gaussian particle filters [Kotecha and Djuric, 2003] are a special kind of particle filter which, similar to the extended and unscented Kalman filters, approximate the posterior with a normal distribution. They do not require a resampling step.

ii. Importance weight calculation (update),

iii. Resampling

The first two steps can be considered trivially parallel, as they involve operations on a single particle and do not require any global knowledge of the whole particle population. A data parallel approach would be applicable in this case. The resampling step, however, does require information from the entire particle population which cannot be trivially executed in parallel.

3.3.1 Particle Filter Network

For a truly distributed design we will use the particle filter scheme from [Simonetto and Keviczky, 2009]. The main idea behind this approach is to construct a network of smaller particle filters. Instead of communicating estimates or other aggregate data, these particle filters exchange only a few representative particles in each round. Such an approach is inherently scalable, as instead of increasing the size of individual particle filters, more particle filters can be added to the network. Each particle filter runs the algorithm as described in Algorithm 3.

With this distributed scheme, each particle filter can be executed on a different computation unit. Depending on the underlying hardware memory model and architecture, a suitable network scheme can be chosen for efficient (particle) data transfer amongst the computation units. Figure 3.1 depicts a number of possible configurations.

Suppose we have a network of N particle filters, each maintaining m particles. In every iteration of the estimation procedure, each particle filter sends its t likeliest particles to its direct neighbours. Furthermore, the received particles replace the local particles with the smallest weights. The number of neighbouring filters depends on the network topology. From the configurations presented in Figure 3.1 the star network is a special case. It can be considered a semi-distributed or hybrid approach. In this network, each particle filter writes its t likeliest particles to a global data structure (black circle in the diagram). These particles are subsequently sorted to find the t particles with the highest weight. All the particle filters add these t particles to their local particle set.

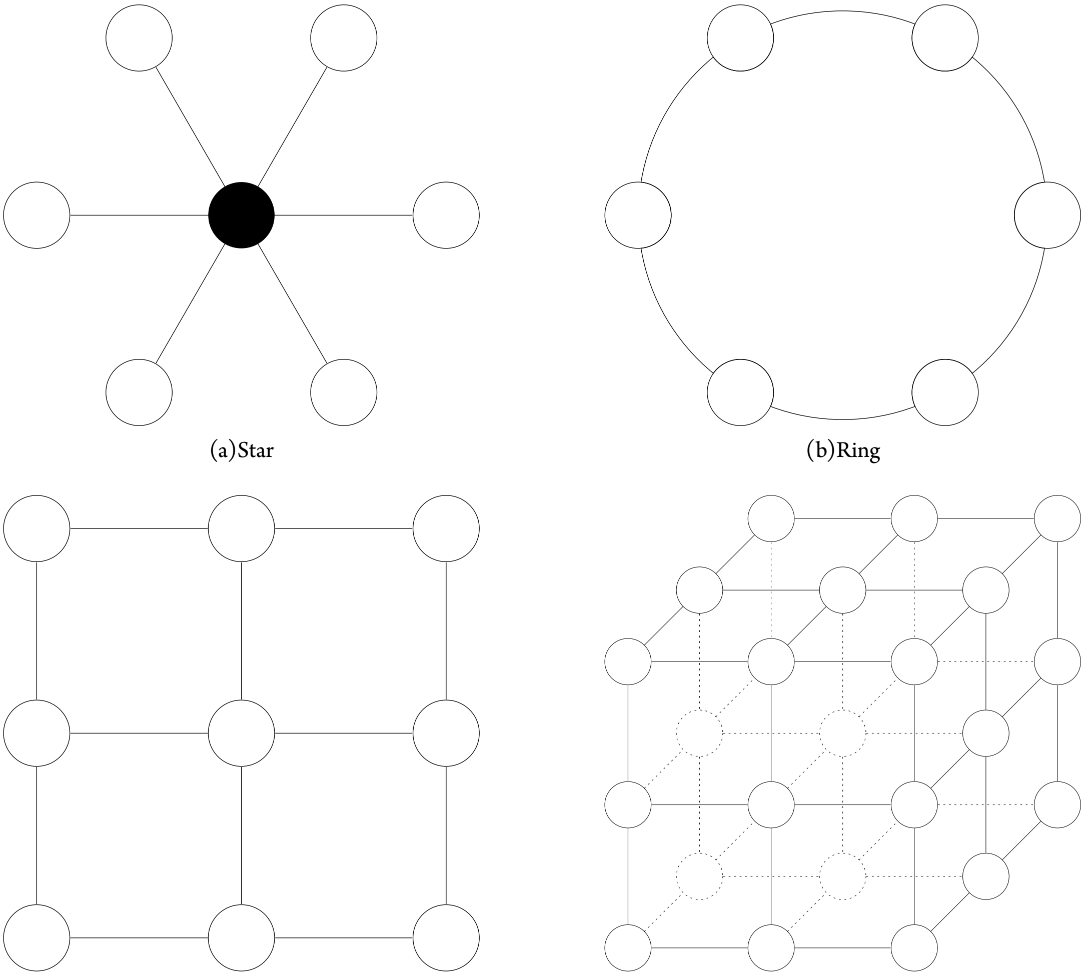
We know from [Simonetto and Keviczky, 2009] that the accuracy of such a distributed particle filter network approaches that of a centralised particle filter with mN nodes , even when t is much smaller than the number of particles in each node m.

3.3.2 Resampling

The usual problems with parallel resampling are avoided with this approach. Each particle filter performs local resampling on its own particles. One of the main concerns with resampling is that it can lead to a significant loss of variation amongst the particle population. Therefore, much attention is needed in order to avoid resampling too often. We propose a simple probabilistic resampling scheme with parameter $r \in [0, 1]$ indicating the probability of performing the resampling step at each round. Each particle filter draws independently $u \sim U(0, 1)$ and only performs the resampling step if $u < r$.

3.3.3 Global Estimate

One of the main challenges in the design of a network of particle filters is the calculation of a global estimate. Whether such a global estimate can be efficiently calculated highly depends on the underlying hardware architecture and specifically, the cost of communication amongst the computation units running the particle filters. We know from the simulation results presented in [Simonetto and Keviczky, 2009] that the worst local



(a)Star

(b)Ring

(c)2D Mesh

(d)3D Mesh

Figure 3.1: Possible configurations for a particle filter network

```
$X'_t \leftarrow \emptyset$
for $i$ in $1 : M$ do
    sample $x_t^{[i]} \sim p(x_t \mid x_{t-1}^{[i]})$
    $w_t^{[i]} \leftarrow p(z_t \mid x_t^{[i]})$
    $X'_t \leftarrow X'_t + x_t^{[i]}$
end for
sort $X'_t$ according to weight
send $< x_t^{[1]}, w_t^{[1]} > \cdots < x_t^{[n]}, w_t^{[n]} >$ to neighbours
receive $< x_t^{[M-n]}, w_t^{[M-n]} > \cdots < x_t^{[M]}, w_t^{[M]} >$ from neighbours
$X_t \leftarrow \emptyset$
for $i$ in $1 : M$ do
    draw $r \sim U[0 : \sum w_t]$
    $j, s \leftarrow 0$
    while s < r do
        $s \leftarrow s + w_t^{[j]}$
        $j \leftarrow j + 1$
    end while
    $X_t \leftarrow X_t + x_t^{[j]}$
end for
```

Algorithm 3: Distributed Particle Filter

estimate in such a network is still close to the best local estimate. The trade-off between the extra communication cost and the potential estimation improvement is highly dependant on the application as well as the hardware.

3.4 Filter Parameters

While traditional centralised particle filters are characterised only by the total number of particles, our distributed particle filter has a number of parameters which influence its behaviour. In this section we will summarise these parameters, which where discussed throughout this chapter.

Number of particles The total number of particles for each particle filter.

Number of particle filters The number of particle filters in the network.

Particle filter network topology Defines how the particle filters are connected, and how information propagates through the network.

Number of exchanged particles How many particles are exchanged between neighbouring particle filters.

Resampling frequency How often resampling is performed.

3.5 Discussion

In this chapter, we reviewed existing literature on the subject of distributed particle filtering. None of these attempts have been able to demonstrate a large scale (millions of particles) particle filter implementation for real-time estimation problems. We presented a fully distributed particle filter algorithm based on the work of [Simonetto and Keviczky, 2009] which can be implemented on different hardware architectures.

The general idea behind this design is to construct a large particle filter by forming a network of smaller particle filters. The smaller particle filters operate independently with only a limited communication amongst neighbouring nodes. Finally, we discussed a number of parameters which influence the behaviour of the particle filter.

4 General-Purpose Computation on GPUs

In recent years, Graphics Processing Units (GPUs) have evolved into highly parallel processors with massive computational capacity. Modern GPUs offer a peak performance of over a thousand GFLOPS; an order of magnitude higher than that of conventional multi-core CPUs. The reason for this massive performance gap lies in the special workloads the GPU is designed to handle. This substantial computational capacity, together with high availability, has led to the wide adaptation of GPUs beyond their original purpose of graphics rendering.

We will start with an overview of the GPU architecture in Section 4.1. In Section 4.2, we will discuss the programming model for general purpose computation on GPUs. Finally, in Section 4.3 we will take a closer look at a number of GPUs which we will use for the experiments described in Chapter 7 and Chapter 8.

4.1 GPU Architecture

In order to better understand the GPU architecture, we briefly examine the computer graphics processing pipeline which has been dictating the design of GPUs. Furthermore, we discuss the main characteristics of throughput-oriented architectures, of which GPUs are a prime example.

4.1.1 Graphics Processing Pipeline

Real-time computer graphics has been the primary target for the development of the GPU architecture. The *graphics processing pipeline* consists of a number of stages which can be classified as either fixed-function or programmable [Fatahalian and Houston, 2008]. This pipeline converts a set of primitives (e.g. lines, polygons) to actual pixel values. The programmable stages of the pipeline are defined by *shader functions*, which are usually expressed in a high-level language. The *shader* code is subsequently compiled into a binary which can run on the GPU. The shader functions are applied to hundreds or thousands of graphical entities (e.g. pixels, vertices) which offers great opportunity for data parallelism.

The fixed function stages are, however, difficult to parallelise as they involve interaction between multiple entities. Therefore, these stages are separated from the programmable parts and are (usually) implemented in hardware. These fixed function stages include texture filtering[1] and rasterisation[2].

4.1.2 Throughput-Oriented Architectures

The GPU architecture has been classified as a *throughput-oriented architecture* [Garland and Kirk, 2010]. In contrast to traditional (multi-core) CPUs, throughput-oriented processors sacrifice the latency of individual

[1] Texture filtering determines the texture value given a specific coordinate for texture mapped pixels.

[2] Rasterisation involves transforming primitives into pixel values.

tasks in order to maximise total throughput. The main assumption for this class of processors is that they will be presented with highly parallel workloads.

With the ever increasing gap between processor speeds and memory access latencies, often referred to as the *Memory Wall* [Wulf and McKee, 1995], throughput oriented architectures employ a different strategy to overcome this problem. While traditional CPUs use extensive caching, sophisticated branch prediction and out-of-order execution to hide memory access latencies, throughput oriented processors rely on the execution of a vast number of threads which can resume work while data is being fetched from the much slower off-chip memory. This results in a much simpler processing core design which allows for more chip resources to be dedicated to processing elements.

Three main characteristics have been identified for throughput-oriented processors: Focusing on many simple processing cores, hardware multithreading and SIMD execution [Garland and Kirk, 2010]. We already discussed the concept of having many simple processing cores when dealing with memory access latency. Hardware multithreading refers to a special case of multithreading in which the execution context of the threads are maintained on-chip. This allows switching context at no cost which allows for greater exploitation of instruction- and thread-level parallelism. Single Instruction, Multiple Data (SIMD) [Flynn, 1966], refers to a class of computer architectures in which a single instruction stream operates on multiple data streams. The SIMD architecture allows for a single control unit to work with multiple arithmetic units, which results in effectively dedicating more chip resources (e.g. transistors) to arithmetic units.

4.1.3 NVIDIA GPU Architecture

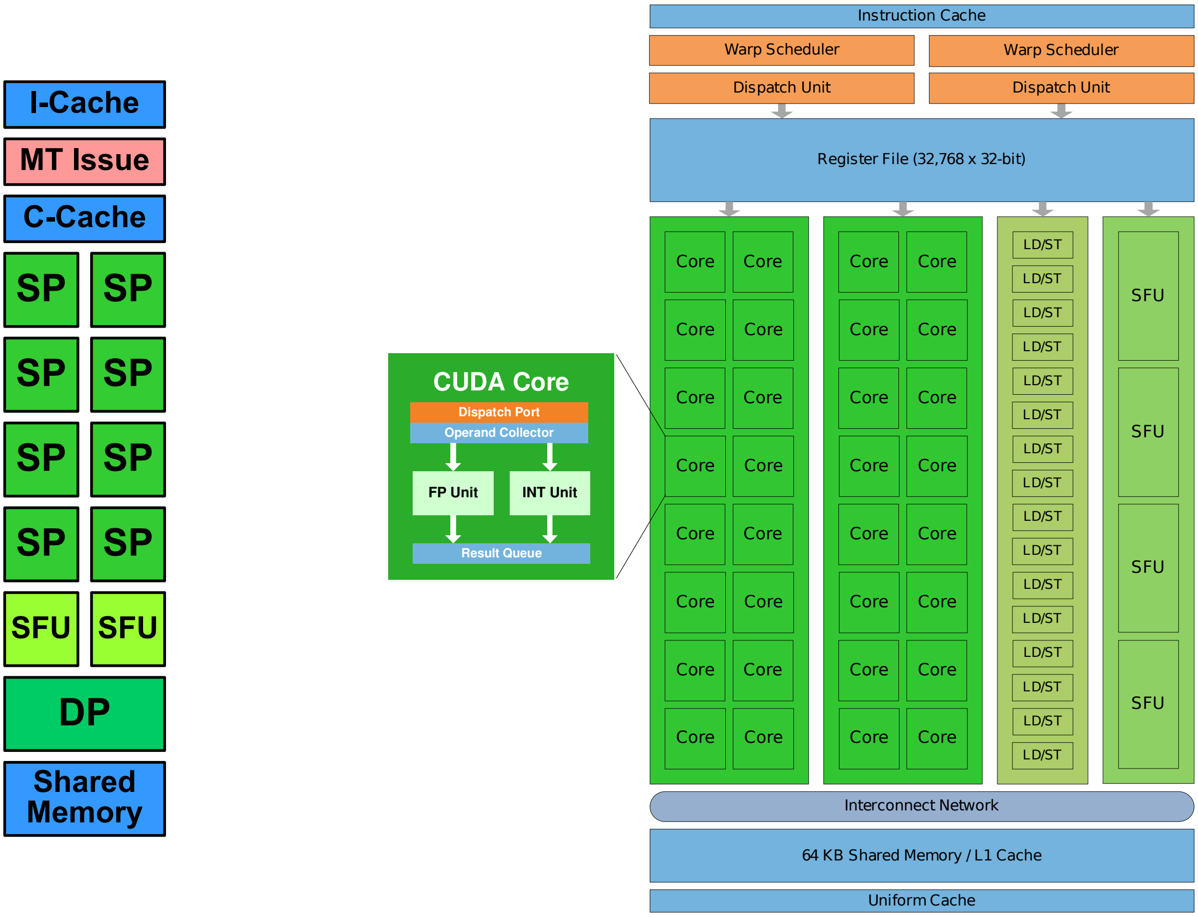
We will now examine the *CUDA* architecture [NVIDIA, 2010a], introduced by NVIDIA in 2006. The G80 microprocessor was the first NVIDIA GPU to move from the traditional pipeline execution model to a unified shader model [NVIDIA, 2006]. The unified shader architecture allows different shaders (e.g. pixel shaders, vertex shaders) to run on the same computation units, marking the end of the traditional pipeline architectures.

CUDA-based GPUs are based on a scalable array of processors, called *Streaming Multiprocessors* (SM) [Lindholm et al., 2008]. A streaming multiprocessor is a collection of simple processing cores with integer and floating point units, combined with on-chip memory. Each multiprocessor can maintain hundreds or even a thousand resident threads, depending on the hardware generation. The SM contains thousands of registers divided amongst all resident threads and few tens of kilobytes of fast shared shared memory. Figure 4.1 gives a high level overview of the SM design for two NVIDIA architectures.

SIMT Execution Model

In order to efficiently run thousands of concurrent threads, CUDA employs what is called a *Single-Instruction, Multiple-Thread* or SIMT architecture. In this architecture, threads are organised into *warp*s, which are basically a group of 32 threads. Each warp is executed independently, with the warp scheduler choosing a warp with active threads which is ready for execution.

Although similar to the SIMD architecture described in the previous section, SIMT has two main differences: programming model and independent branching. With classic SIMD vector machines, the programmer explicitly uses vector operations on (fixed width) vectors, whereas SIMT enables the programmer to specify the behaviour of an individual thread. Threads in SIMT can branch independently, but within a warp, this leads to the serialisation of the different execution paths.



(a)GT200

(b)GF100 (Fermi)

Figure 4.1: NVIDIA Streaming Multiprocessor (SM), source: NVIDIA

4.2 Programming Model

With the introduction of programmable shaders for GPUs, numerous (high level) programming languages have been developed in order to ease the task of programming these shaders. All these shading languages are designed according to the needs of modern graphics pipelines. The increased popularity of the GPU outside the graphics domain has led to the introduction of general purpose languages specifically designed for this purpose. In this section we will examine the programming model of *CUDA*, introduced by NVIDIA in 2006 for its line of GPUs. A competing standard from the Khronos Group, called OpenCL [Khronos Group, 2010], was released in 2008. OpenCL, an open and royalty-free standard, is a programming framework for heterogeneous platforms including multi-core CPUs as well as GPUs. The programming and memory model is, however, comparable to that of CUDA.

A CUDA program is divided into a *host* and a *device* part. The host program runs on the CPU and the device program which runs on the GPU consists of one or more *kernels* [NVIDIA, 2010a]. Each kernel typically consists of numerous threads grouped into *blocks*. Blocks of threads are, furthermore, grouped into a *grid*. Blocks of threads can be executed concurrently on different multiprocessors.

Threads within each block are organised in a one-, two- or three-dimensional fashion. Each thread has a unique ID within a block which can be accessed in the kernel code by the built-in variable `threadIdx`.

Similarly, blocks are grouped into one- or two-dimensional grids, with a unique ID accessible through the variable `blockIdx`. Threads from the same block can synchronise their execution by defining synchronisation points with `__syncthreads()`.

4.2.1 Memory Model

Memory is managed in a hierarchical way, with each thread having access to private local memory. All threads within a blocks have access to the same shared memory, which as with thread-local memory has only the lifetime of a single kernel execution. Global memory is accessible to all threads from all blocks and persists through kernel calls. Figure 4.2 gives an illustration of the CUDA memory hierarchy.

Global memory is stored off-chip, and is the slowest available memory on the GPU. It is accessed through 32-, 64- or 128 byte memory transactions. Whenever one or more threads within a warp access global memory, a number of memory transactions are issued. If the memory accesses within a warp meet certain requirements the requests are *coalesced* into a single memory transaction. This is the most efficient way to access global memory. The requirements for memory coalescing are different between devices of different generations, but in general aligned and especially contiguous (unit stride) loads/stores are the most efficient.

Shared memory is a fast on-chip memory available to all threads within a thread-block. It is much faster than the off-chip global memory, but is generally much smaller (8-, 16- or 32KB per SM depending on the hardware generation) and does not persist through kernel calls. The manner in which shared memory is accessed can have a huge impact on the performance. Shared memory consists of a number of (16 or 32) *banks*. A read or write request to shared memory can only reach the optimum performance if all the requested locations from a single warp fall into distinct banks. If k addresses from a request fall in a single bank, we have a k-way *bank conflict*. It is, therefore, important to consider the bank sizes and access strides in order to prevent bank conflicts.

4.3 Hardware Overview

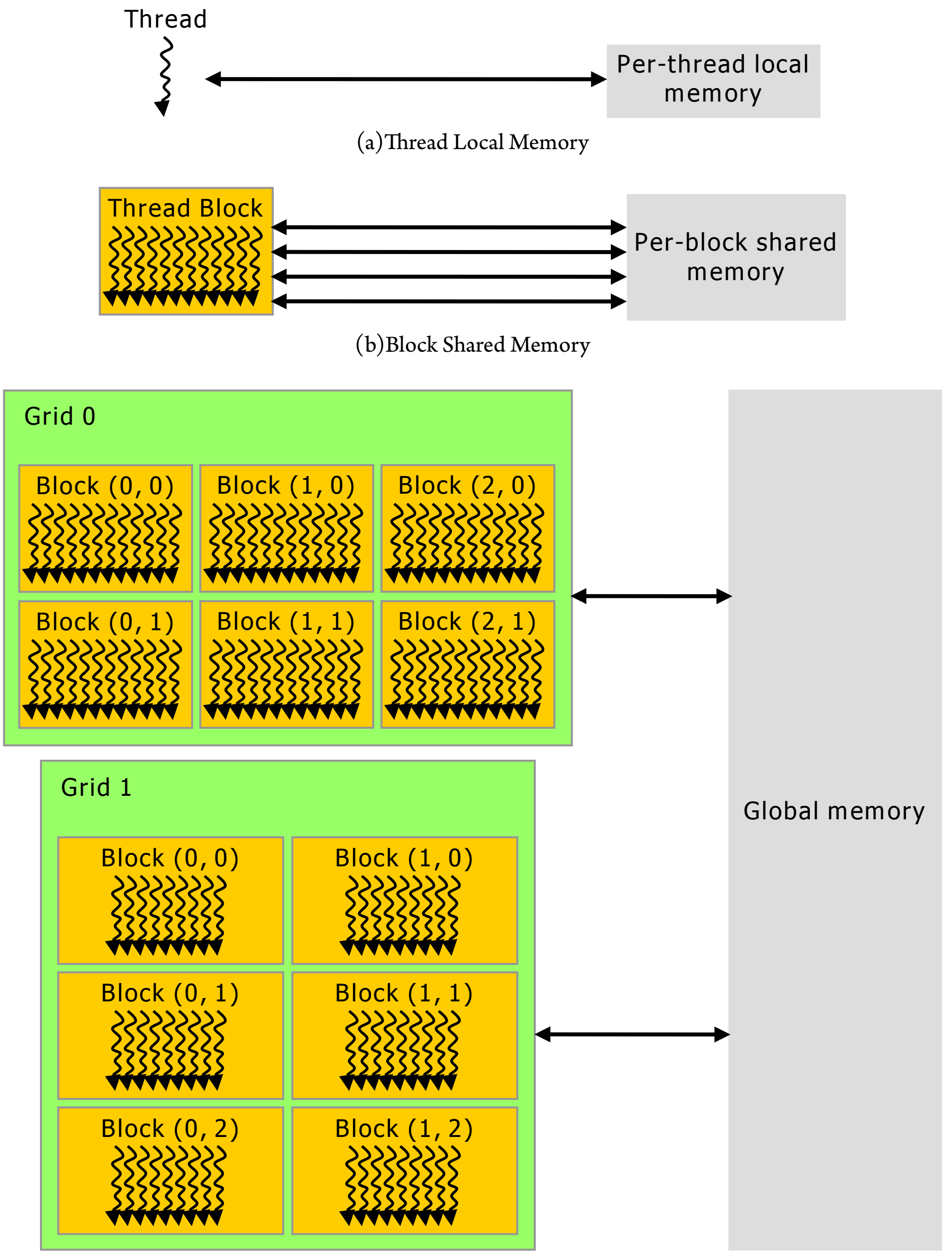
In this section we will examine two NVIDIA GPUs which we use for the experiments described in Chapter 7 and Chapter 8. The first GPU is the GTX280, introduced in 2008, which is based on the GT200 microarchitecture. It contains 240 CUDA cores divided amongst 30 SMs clocked at 1296 MHz. As depicted in Figure 4.1a Each SM consists of 8 processing cores (SPs) and 2 Special Function Units (SFUs). The SPs can execute one single-precision floating-point operation per clock cycle or two multiply-add (MAD) instructions. The SFUs handle transcendental math and interpolation but can also compute four floating-point operations per cycle. This leads to a theoretical maximum throughput of:

$$1.296 \text{ GHz} \times 30 \times (8 \times 2 \text{ FLOP} + 2 \times 4 \text{ FLOP}) = 933.12 \text{ GFLOP/s}$$

This assumes that instructions are executed on both SPs and SFUs (dual-issuing). However, if there are no instructions issued to the SFUs, this figure drops to 622.08 GFLOP/s. Similarly, if no MAD instructions are issued, it further drops to 311.04 GFLOP/s. The GTX280 has 1GB of main memory with a throughput of 141.7 GB/s. Each SM has 16KB of shared memory together with 16K 32-bit wide registers.

The second NVIDIA GPU we use for the experiments is the GTX480, based on the GF100/Fermi architecture introduced in 2010. It has 480 CUDA cores clocked at 1401 MHz, partitioned into 15 SMs. Each SM has 32 cores and 4 SFUs as shown in Figure 4.1b. The GTX480 delivers a single-precision peak performance of:

$$1.401 \text{ GHz} \times 15 \times (32 \times 2 \text{ FLOP}) = 1344.96 \text{ GFLOP/s}$$



(a) Thread Local Memory

(b) Block Shared Memory

(c) Global Memory

Figure 4.2: NVIDIA CUDA Memory Hierarchy, source: NVIDIA

	Processor count	SM count	DRAM	Memory bandwidth	Single-precision FLOPS
GTX280	240	30	1024 MB	141.7 GB/s	933.12 GFLOPS
GTX480	480	15	1536 MB	177.4 GB/s	1344.96 GFLOPS

Table 4.1: NVIDIA GPU Overview

As with the GTX280, this depends on the presence of MAD instructions. The maximum throughput drops to 672.48 GFLOP/s without MAD instructions. The GTX480 has 1536MB of memory with a maximum throughput of 177.4 GB/s. Each SM in the GTX480 has 32K 32-bit wide registers and 48KB of shared memory, which can be dynamically reconfigured as an L1 cache.

Table 4.1 gives an overview of the two NVIDIA cards described above.

Part II

Implementation and Experiments

5

Two Applications: Robot Localisation and Robotic Arm

In order to test, verify and benchmark our GPU-based distributed particle filter implementation, based on the ideas presented in Chapter 3, we use the following two estimation problems: (i) Unicycle robot localisation, and (ii) Robotic arm.

The unicycle localisation problem, discussed in Section 5.1, is a relatively small estimation problem which does not allow us to test the performance and the accuracy of the filter under different conditions. We use this application in order to illustrate the model-specific aspects of the particle filter implementation in Chapter 6. Section 5.2 presents the robotic arm estimation problem. We use this estimation problem in order to measure the performance of the filter in Chapter 7 and the accuracy in Chapter 8.

5.1 Unicycle Robot Localisation

In this estimation problem we use noisy range measurements from fixed sensors to localise a mobile robot which moves freely on a 2D plane. The state consists of the tuple (x, y, θ) where (x, y) denote the coordinates of the robot, and θ refers to its orientation. The dynamical state equation is as follows:

$$x(t) = x(t-1) + \frac{\hat{v}}{\hat{r}} \sin(\theta(t-1) + \hat{r}\Delta t) - \sin(\theta(t-1))$$

$$y(t) = y(t-1) - \frac{\hat{v}}{\hat{r}} \cos(\theta(t-1) + \hat{r}\Delta t) - \cos(\theta(t-1))$$

$$\theta(t) = \theta(t-1) + \hat{r}\Delta t + \gamma\Delta t$$

Note that $\hat{v}$ and $\hat{r}$ represent (noisy) control inputs and γ is a noise term. The weights can be calculated according to a Gaussian probability distribution function. The sensor data consists of the set of measured distances to the fixed sensors.

5.2 Robotic Arm

The robotic arm, in this experiment, has a number of joints which can be controlled independently. It has one degree of freedom per joint. Each joint has a sensor to measure the angle. There is a camera mounted at the end of the arm. This camera is used for tracking an object which is moving on a fixed 2D plane. Figure 5.1 gives an illustration of this robotic arm.

The state equations for both the robotic arm joint movements as well as the moving object are as follows:

$$\theta_i(t) = \theta_i(t-1) + \Delta t u_i + \omega_\theta \quad \forall i \in \{0, \dots, N\}$$

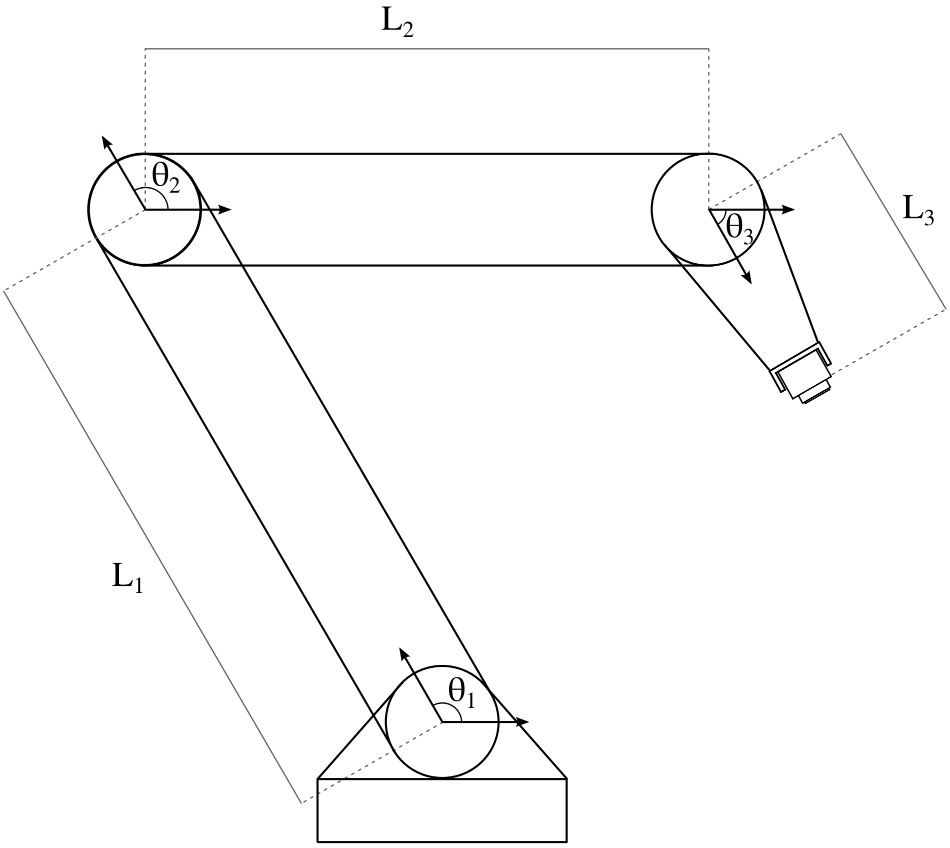


Figure 5.1: Robotic Arm with a Mounted Camera

$$\begin{aligned} x(t) &= x(t-1) + v_x(t-1)\Delta t + \omega_x \\ y(t) &= y(t-1) + v_y(t-1)\Delta t + \omega_y \\ v_x(t) &= v_x(t-1) + \omega_{vX} \\ v_y(t) &= v_y(t-1) + \omega_{vY} \end{aligned}$$

Here θ_i represents the angle of joint i and u_i represents the control signal for the actuator of joint i. (x, y) and (v_x, v_y) are the position and velocity vectors of the moving object. ω is the noise term.

The camera detects the object in its own frame of reference. Therefore, in order to translate between the global coordinates and the camera reference frame, we need to apply, for each joint, the rotation and shift matrices according to the joint angle and link length:

$$\begin{aligned} \begin{bmatrix} u(t) \\ v(t) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} & \left(\begin{bmatrix} 0 \\ 0 \\ -L_N \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_N) & -\sin(\theta_N) \\ 0 & \sin(\theta_N) & \cos(\theta_N) \end{bmatrix} \right. \\ & \left(\begin{bmatrix} 0 \\ 0 \\ -L_{N-1} \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_{N-1}) & -\sin(\theta_{N-1}) \\ 0 & \sin(\theta_{N-1}) & \cos(\theta_{N-1}) \end{bmatrix} \right. \\ & \vdots \\ & \left(\begin{bmatrix} 0 \\ 0 \\ -L_1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_1) & -\sin(\theta_1) \\ 0 & \sin(\theta_1) & \cos(\theta_1) \end{bmatrix} \begin{bmatrix} \cos(\theta_0) & \sin(\theta_0) & 0 \\ -\sin(\theta_0) & \cos(\theta_0) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \right) \ldots \Bigg) \Bigg) + \begin{bmatrix} \omega_{uX} \\ \omega_{uY} \end{bmatrix} \end{aligned}$$

θ_i refers to the angle of joint i, L_i refers to the length of link i and ω is the noise term. The angle measurements for the individual joints are according to the following equation:

$$\begin{bmatrix} \widehat{\theta_0}(t) \\ \widehat{\theta_1}(t) \\ \vdots \\ \widehat{\theta_N}(t) \end{bmatrix} = \begin{bmatrix} \theta_0(t) \\ \theta_1(t) \\ \vdots \\ \theta_N(t) \end{bmatrix} + \omega_k$$

The noise term is depicted by ω. For the sake of simplicity, all noise terms are assumed to be Gaussian, although particle filters do not impose such a restriction.

The main advantage of this robotic arm model is its parametric form which by adjusting the number of joints, allows for increasing and/or decreasing the state dimensions.

6 Particle Filtering on the GPU

In order to have an efficient particle filter implementation on the GPU, we need to modify the standard particle filter algorithm. We presented a distributed particle filtering scheme in Chapter 3 which serves as basis for our GPU implementation. We use the unicycle robot localisation application, presented in Chapter 5, to illustrate the model-specific aspects of the filter implementation. The code for the unicycle implementation is listed in Appendix A.

In this chapter we will discuss how this distributed filter design translates to a concrete implementation on the GPU. Section 6.1 discusses the general aspects of our GPU implementation, while Section 6.2 specifically discusses the memory organisation. Section 6.3 discusses the implementation of each kernel.

6.1 Filter Design

The distributed particle filtering scheme, discussed in Chapter 3, forms the basis of our GPU implementation. The basic idea of this approach is to build a network of small particle filters instead of a single large particle filter. This network of particle filters has three main characteristics:

1. Individual particle filters remain small, which allows for efficient resampling.
2. Excluding the particle exchange step, the remainder of the filter algorithm can be executed independently for each particle filter.
3. Scaling is achieved by adding more particle filters, instead of increasing the size of individual particle filters.

Recall, from Chapter 4, that with the CUDA programming model kernels run on the GPU as blocks of threads. These blocks are executed concurrently on different Streaming Multiprocessors (SM). Threads within a block can communicate using fast on-chip memory and synchronise their execution using barrier instructions. However, there are a number of limits which we need to consider. We discussed in Chapter 4 that the total number of threads within each block is limited to 512 for 1.x devices, and 1024 for 2.x devices. Furthermore, there is also a limit on the available shared memory (16KB for 1.x devices, and 48KB for 2.x devices).

Based on this, we choose the following mapping:

1. Each particle is represented by a single CUDA thread;
2. Each particle filter is represented by a single CUDA thread block.

This particular mapping has a number of advantages. The sorting and resampling steps, which require a lot of communication are limited to a single thread block. Therefore, we can utilise the fast shared memory, as well as the synchronisation mechanisms for an efficient implementation of these steps.

Parameter	Symbol
Number of Particles	m
Number of Particle Filters	N
Particle Filter Network Topology	Star/Ring/2D Mesh/2D Torus/3D Mesh
Number of Exchanged Particles	t
Resampling Frequency	r

Table 6.1: Particle Filter Parameters

There are, however, a number of disadvantages to this approach. The amount of available shared memory as well as the limit on the number of threads within a thread block restrict the size of an individual particle filter. Nevertheless, this is in line with our design strategy, in which scalability is achieved by increasing the total number of particle filters, instead of the size of individual particle filters.

Based on the details of Algorithm 3 we identify the following steps for our particle filter implementation.

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

The generation of pseudo-random numbers is listed here as a separate step. Pseudo-random numbers are used both for the sampling as well as the resampling step. We have chosen to generate these numbers separately at the start of each iteration. Table 6.1 lists the available parameters affecting the behaviour of the filter. We will explain these parameters further when we discuss the corresponding kernel.

6.2 Data Layout

There are three main data types for holding the data needed by the particle filter. The `state` structure is the main data structure containing the data of a single particle. These are stored in an array to the length of the whole particle population[1]. These structures are initialised on the GPU at the start of the execution. There are also the `control` and `measurement` structures, holding respectively, the control (actuation) values and the (noisy) measurements. The particle weights are stored separately in a single-precision floating point array.

Listing 6.1 shows these three data structures for the unicycle model discussed in Chapter 5.

6.3 Individual Kernels

In this section we will discuss, in detail, the implementation of each step of the particle filter algorithm.

[1] We actually allocate twice the required memory for the particle data. This is explained in the discussions for the sorting kernel

Listing 6.1: Particle Filter Data

```
typedef struct _state
{
        float x;
        float y;
        float theta;
} state;
typedef struct _control
{
        float velocity;
        float angular_velocity;
} control;
typedef struct _measurement
{
        float sensor_data[NUM_SENSORS];
} measurement;

struct state* particle_data;
float*        particle_weights;
```

6.3.1 Pseudo-Random Number Generation

As with any Monte Carlo simulation technique, particle filters rely heavily on random numbers. *Pseudo-random number generators* (PRNGs) are, therefore, an essential part of any Monte Carlo simulation. Not all PRNGs are suitable for parallel execution on GPUs. We will now examine two PRNGs which we use for our experiments. For a detailed overview of GPU implementations of various PRNGs, see [Demchik, 2011].

Mersenne Twister

Mersenne Twister [Matsumoto and Nishimura, 1998b] is a highly popular pseudo-random number generator, characterised by its large period. Although not directly suitable for cryptographic purposes, Mersenne Twister is used in a wide range of applications, including Monte Carlo simulations. A distributed scheme is proposed in [Matsumoto and Nishimura, 1998a], which dynamically creates Mersenne Twister parameters based on process IDs. This enables concurrent Mersenne Twister PRNGs to produce uncorrelated number sequences. The NVIDIA CUDA SDK provides a CUDA implementation [Podlozhnyuk, 2007] based on this scheme with a period of $2^{607} - 1$. We will, however, use an optimised CUDA implementation from [Saito, 2010] which is claimed to be 1.5 times faster and has a period of 2^{11213}-1.

This CUDA implementation produces uniform floating point numbers in the range of [1,2). We use the Box-Muller transformation [Box and Muller, 1958] to transform these uniformly distributed numbers to a normal distribution.

Xorshift

Xorshift [Marsaglia, 2003] is a family of PRNGs which are based on the repeated use of a simple construct, namely the exclusive-or (xor) of a number with a shifted version of itself. Compared to Mersenne Twister, these PRNGs are much simpler and faster. Xorshift PRNGs, however, have generally much smaller periods (e.g. 2^{64}, 2^{128}, 2^{192}).

The NVIDIA CUDA Toolkit provides the *CURAND* library [NVIDIA, 2010b]. CURAND is a CUDA

Listing 6.2: Sampling/Importance Weights Kernel

```
__global__ void sampling_importance_weights
                  (state* const d_particle_data,
                   float* const d_particle_weights,
                   const control control_data,
                   const measurement measurement_data
                   const float* const d_random,
                   const float dt);
```

implementation of *xorwow*, which is a member of the xorshift PRNG family. It is claimed to have period greater than 2^{190}, which should be enough for our experiments. CURAND can produce uniformly distributed numbers in the range of $(0,1]$ and normally distributed numbers with a given mean and standard deviation.

6.3.2 Sampling/Importance Weights

The sampling step, as described in Section 2.3, involves generating new particles $x_t^{[m]}$ from the previous particles $x_{t-1}^{[m]}$. This is achieved by sampling from the state transition distribution, taking into account any possible control values. The pseudo-random numbers generated in the previous step are used here for generating samples from the state transition distribution.

The importance weights calculation, which uses measurement data to assign weights to the particles, is combined with sampling into a single CUDA kernel.

The measurement and control structures are directly passed to this kernel as parameters. The CUDA compiler places these structures in constant memory, which is cached on each SM. In the applications we have encountered so far, the measurement and control structures have easily fit the constant memory (64KB). If for a particular application, these data structures exceed the constant memory size, they need to be placed either in texture memory or global memory.

As mentioned in the previous section, the pseudo-random numbers are generated prior to the execution of this kernel. A pointer to the data structure holding these numbers is passed along to the kernel as well.

6.3.3 Local Sorting

Each particle filter needs to internally sort its particles according to their weights. This serves two purposes: First of all, for the particle exchange step, each particle filter needs to send its t particles with the highest weight, and for each neighbour, replace t particles with the lowest weight with the particles received. Secondly, in order to determine the global estimate, the local winner from each block needs to be calculated.

Listing 6.3: Particle Sort Kernel

```
__global__ void block_sort(
        state* const d_particle_data_sorted,
        float* const d_particle_weights,
        const state* const d_particle_data_unsorted,
        const int num_particles);
```

We use a *bitonic sort* [Batcher, 1968] implementation from the NVIDIA CUDA SDK, which has a complexity of $O(n\ log^2(n))$. Bitonic sort is an example of a *sorting network*, which uses a fixed sequence of comparisons in order to sort the input. This makes them attractive for parallel sorting.

The particles are generally too large to entirely fit in shared memory. Therefore, we only store the weights in shared memory and sort them separately. We keep track of the sorted indices with an index array (also stored in shared memory). Once the weights are sorted, each thread reads from global memory from the position stored in the indices array, and writes back to its own location. Recall from Chapter 4 that in order to achieve high memory throughput, all accesses need to be contiguous. This is only the case for the writes. Nevertheless, this allows us to apply our filter implementation to much larger problems.

As a result of not using shared memory for the actual particle data, we cannot sort the particle data in-place in global memory. We need to allocate two arrays: one for holding the unsorted particles and another one for the sorted particles. This prevents any conflict between reads and writes from different threads within each block.

6.3.4 Global Estimate

For this step we need to implement a reduction operation in order to calculate an estimate given a probability density function. Whether to use a weighted average of all the particle values or to choose the single most representative particle as the estimate depends on the particular application. We consider the single particle with the highest weight to be global estimate of the particle filter. In order to calculate this, we need to find the particle with the highest weight amongst the local winners from each particle filter.

To this end we use an external library called Thrust[2]. Thrust is an STL-like library providing a high-level interface for CUDA implementations of many useful algorithms. We use `thrust::max_element()` to find the particle with the highest weight amongst the local winners from each block.

6.3.5 Exchange

Listing 6.4: Particle Exchange Kernel

```
__global__ void exchange_ring(
        const state_data d_particle_data,
        float* const d_particle_weights,
        const int num_particles,
        const int num_blocks);
__global__ void exchange_2dtorus(
        const state_data d_particle_data,
        float* const d_particle_weights,
        const int num_particles,
        const int num_blocks);
```

There are two parameters influencing the behaviour of this kernel. The particle filter network topology determines which blocks exchange particles. We have implementations for the following networks: (i) Star, (ii) Ring and (iii) 2D Torus. With the star topology, all blocks write their local winners to a global data structure. These particles are subsequently sorted, which determines the global estimate. Furthermore, all blocks read the same particles back. Other topologies, such as the ring network, are more distributed in that exchanged particles are unique to direct neighbouring blocks.

There is also the parameter t, the number of exchanged particles, which determines how many particles are transferred in a single exchange step.

[2] http://code.google.com/p/thrust/

6.3.6 Resampling

Listing 6.5: Particle Resampling Kernel

```
__global__ void resampling(
        const state_data d_particle_data,
        const state_data d_particle_data_tmp,
        const float* const d_particle_weights,
        const float* const d_random,
        const int num_particles,
        const float resampling_freq);
```

The resampling step generates a new particle set by drawing randomly (with replacement) from the original set according to the particle weights. In order to implement this, we first need to calculate the cumulative sum of all the weights within each thread block. Our implementation is based on the parallel prefix sum from [Harris et al., 2007] which consists of two steps: the up-sweep phase and the down-sweep phase. In the up-sweep phase, the elements of the array are traversed "upwards" in a binary tree fashion in order to calculate the partial sums. In the down-sweep phase, the tree is traversed back down to calculate the sums in place. Both phases are logarithmic in time.

With the cumulative sum calculated, each threads picks a uniformly distributed number in the range of [0, 1) (generated separately) and multiplies that with the total sum of the weights. With a custom binary search implementation (also logarithmic), we find the highest index with a weight non-larger than the random selection. This index points to the chosen particle.

Similar to the reasoning used in Section 6.3.3 for the sorting kernel, the actual particle data is not stored in shared memory as it is not guaranteed to fit. Therefore, the reads will be irregular and non-contiguous while the writes remain contiguous and aligned.

6.4 Generic Particle Filtering

Listing 6.6: Model Specific Code

```
__device__  void sampling(
        state* const particle_data,
        const control control_data,
        const float* const d_random,
        const float dt);

__device__ float importance_weight(
        const state* const particle_data,
        const measurement measurement_data);
```

In this chapter we have reviewed our implementation of the distributed particle filter algorithm on the GPU. In order to be able to reuse this filter implementation for different applications we have tried to separate the model-specific parts from generic filter code.

The data structures needed to store the particle data as well as other model-specific elements (i.e. measurement, control) are depicted in Listing 6.1. From the kernels discussed in this chapter, the only model-specific kernel is `sampling_importance()`. For convenience, this kernel calls two device functions which are responsible for the actual calculations of the particle state (sampling) and the importance weights. Listing 6.6 shows the signature of these two device functions. The complete implementation of the unicycle model for our particle filtering framework is presented in Appendix A.

There are a number of consideration for the implementation of custom applications with this framework:

- There should be enough global memory available for allocating data structures for holding the pseudo-random numbers, the weights and twice the particle data.
- The measurement and control structures should together fit into the constant memory.
- The particle with the highest weight is considered to be the global estimate. Some additional steps are needed if a particular applications needs to extract more information from the particle population (e.g. weighted average).

7 Performance Analysis and Optimisations

The GPU implementation of the generic particle filtering framework is discussed in Chapter 6. In this chapter, we evaluate the performance of the the GPU implementation using the robotic arm application, discussed in Chapter 5, on two CUDA enabled NVIDIA GPUs. Based on the results, we perform a number of optimisations in order to improve the implementation efficiency.

First, in Section 7.1, we discuss a performance model we use throughout this chapter for visualising the performance of the different kernels. Next, in Section 7.2, we examine the performance of the basic implementation from Chapter 6 and identify the bottlenecks. In Section 7.3, we perform a number of optimisations in order to improve the efficiency of the implementation. We examine the scalability of the filter implementation in a number of directions in Section 7.4, and finally, in Section 7.5 we will conclude this chapter.

7.1 Performance Model

In order to visualise the performance of our implementation, we use the *roofline model* [Williams et al., 2009]. Focusing on floating-point performance, the roofline model sets an upper bound on the performance given a certain *operational intensity*. In contrast to arithmetic intensity, operational intensity refers to the number of operations per byte of DRAM traffic, ignoring any traffic between the processor and the caches.

The roofline is calculated according to:

$$\begin{aligned}\text{Attainable GFLOPs/sec} = min(&\text{Peak Floating-Point Performance,}\\ &\text{Peak Memory Bandwidth} \times \text{Operational Intensity})\end{aligned}$$

The roofline is presented on a two-dimensional plot with the operational intensity on the x-axis and the attainable GFLOPs on the y-axis. The intersection point of the diagonal line (memory bound) and the horizontal line (computation bound) represents the point of peak computation performance and memory bandwidth. In order to find the upper bound of the floating-point performance for a given kernel, we draw a vertical line from its operational intensity. The intersection point with the roofline represents the upper bound. If this intersection point lies on the diagonal line, the kernel is memory bound on this platform, otherwise it is computation bound.

7.1.1 Platform Bounds

In order to draw an accurate roofline model for the GPU cards used in our experiments, we need to calculate realistic upper bounds. We use a specific benchmark kernel in order to measure the attainable memory bandwidth. This kernel only issues aligned and contiguous memory accesses. The achieved results relative to the specifications of the hardware are presented in Table 7.1.

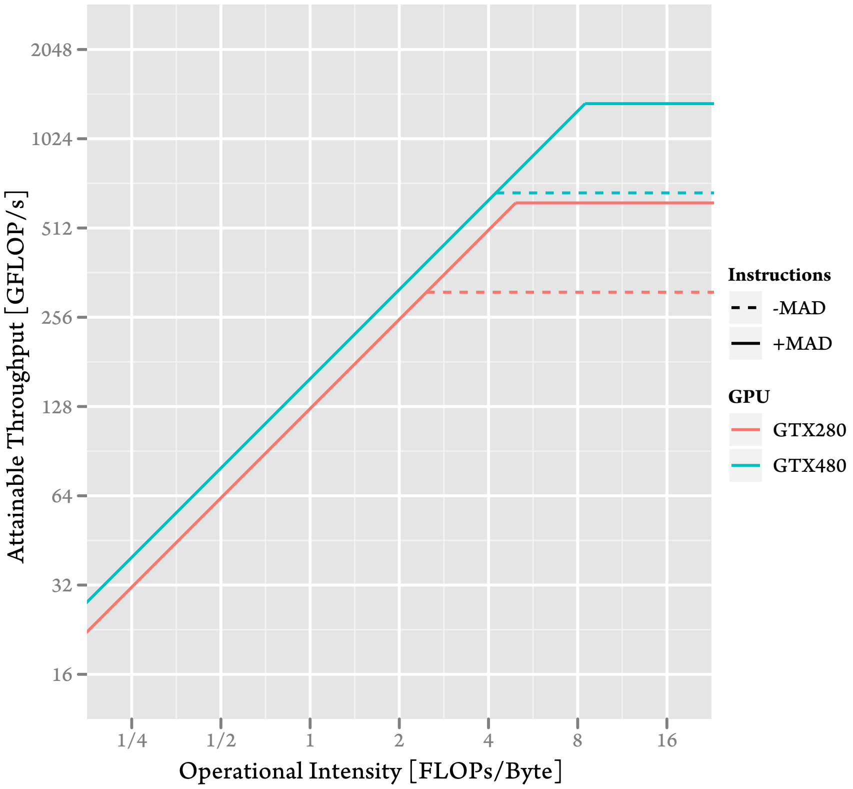


Figure 7.1: Roofline Model

GPU	Attained Bandwidth	Specified Bandwidth	Ratio
GTX480	158.72 GB/s	177.4 GB/s	89.5%
GTX280	125.9 GB/s	141.7 GB/s	88.8%

Table 7.1: Attained vs Specified Memory Bandwidth

In Chapter 4, we calculated different values for the overall instruction throughput for both the GTX480 and the GTX280 depending on whether MAD instructions are executed (or dual-issuing on the SFUs for the GTX280). Figure 7.1 depicts the roofline model for the GTX480 and the GTX280. For both cards we have drawn separate computation bounds depending on whether MAD instructions are being issued. We ignore the GTX280 dual-issuing on the SFUs as it can only happen in limited circumstances which do not apply to our implementation.

In the following section, we will identify the kernels contributing most to the total runtime and use the roofline model to better understand their behaviour.

7.2 Baseline Performance

In this section we measure the runtime of the different parts of the particle filter algorithm using the robotic arm application, as described in Chapter 5. The GPU implementation details are examined in Chapter 6. For this experiment we use the filter and model parameters listed in Table 7.2.

Number of particles	512
Number of blocks	2048
Particle network configuration	Ring
Number of exchanged particles	1
Resampling frequency	1.0
Number of joints	5
State dimension (#joints + 4)	9

Table 7.2: Parameters for performance measurements

The timing measurements are done using a host CPU timer (based on `gettimeofday()`). The results were verified against the GPU execution times reported by the NVIDIA CUDA profiler. Table 7.3 presents these results for both the GTX480 as well as the GTX280. From these results we can conclude that the global reduction step (thrust::max_element) in order to determine the actual estimate as well as the particle exchange kernel do not significantly contribute to the total runtime of the filter. Although we used a ring network topology for this particular experiment, other network topologies (e.g. star, 2D torus) perform similarly. While individual particle filter blocks only exchange a single particle in this experiment, other experiments have shown that increasing the number of exchanged particles does not significantly increase the runtime of the particular kernel. Therefore, for the remainder of this chapter we will focus only on the PRNGs as well as the following three kernels: `sampling_importance()`,`block_sort()` and `resampling()`.

Kernel	Runtime (µs)	Contrib.
curand (xorwow)	819	5.70%
sampling_importance	5889	41.30%
block_sort	4984	35.00%
thrust::max_element	181	1.30%
exchange	45	0.30%
resampling	2329	16.30%
total	14247	

(a)GTX480

Kernel	Runtime (µs)	Contrib.
curand (xorwow)	1310	4.90%
sampling_importance	9659	36.10%
block_sort	8185	30.60%
thrust::max_element	242	0.90%
exchange	151	0.60%
resampling	7238	27.00%
total	26785	

(b)GTX280

Table 7.3: Breakdown

In the following sections we will further examine each of these steps in order to better understand these performance figures.

7.2.1 Pseudo-Random Number Generators

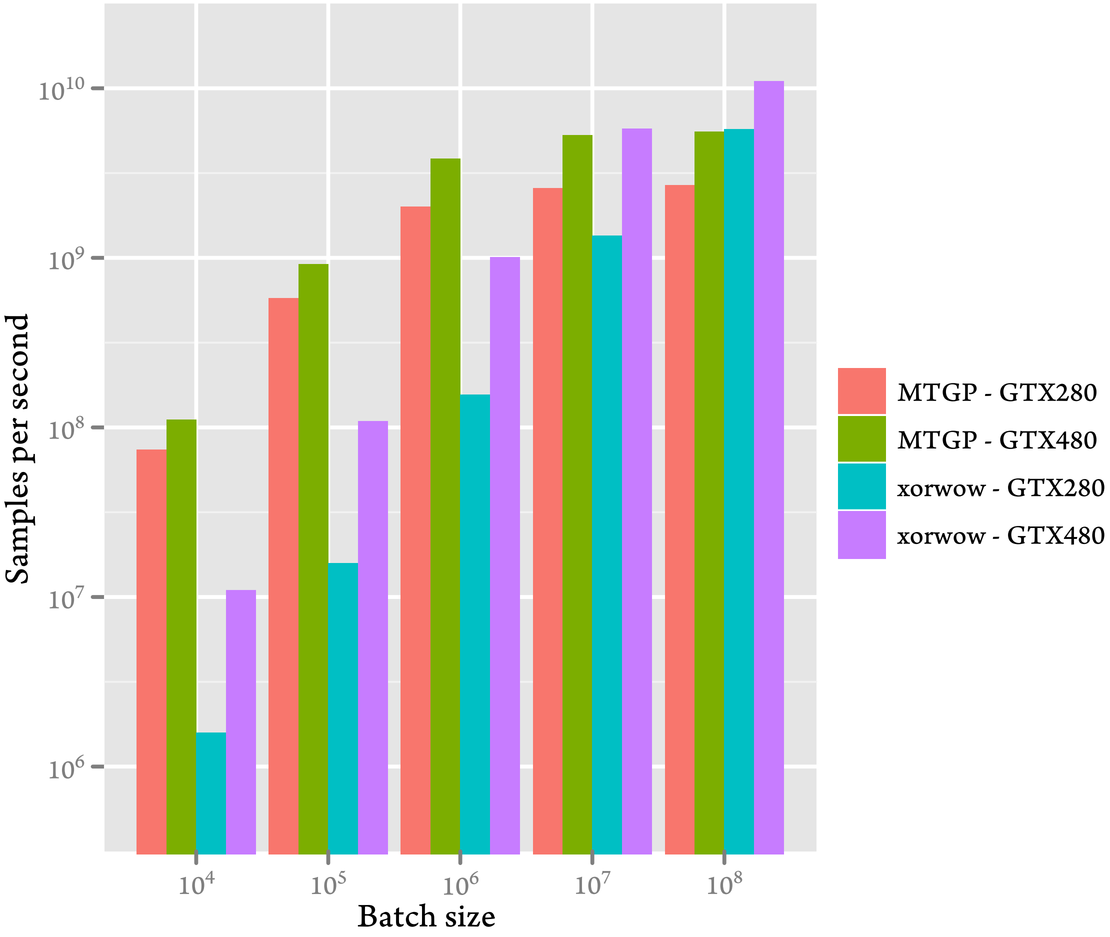


Figure 7.2: Performance comparison of two PRNG implementations

One of the most crucial steps in the particle filter algorithm is the generation of pseudo-random numbers. We discussed two PRNGs in Chapter 6: (i) MTGP, an optimised Mersenne Twister CUDA implementation, and (ii) xorwow, a Xorshift-based PRNG from NVIDIA.In this experiment, we use both PRNGs to produce a batch of standard normally distributed random numbers. As mentioned before, MTGP only produces uniformly distributed numbers. Therefore, we use a custom Box-Muller transformation in order to transform these number into a normal distribution.

We have noticed that the batch size has a great influence on the performance of both PRNGs. Figure 7.2 presents the results of this experiment with different batch sizes. We expect xorwow to outperform MTGP as it is a much simpler algorithm. The experiment results confirm this to be the case only for large batch sizes. MTGP outperforms xorwow in all the test cases with smaller batch sizes. Since NVIDIA released their xorwow implementation with the 3.2 release of CUDA in late 2010, future releases might be better optimised for smaller batch sizes.

7.2.2 Individual Kernel Analysis

Now that we have identified the main kernels which contribute to the overall runtime, we will examine each kernel in detail. To this purpose we will calculate, for each kernel, the total number of bytes transferred from and to global memory as well as the instruction count. This allows us to calculate the utilisation of the GPU resources, and to identify performance bottlenecks.

In contrast to global memory traffic, it is much more difficult to come up with a definitive figure for the instruction count as a lot of the architectural features of modern GPUs are relatively unknown. Some features of the GT200 (e.g. GTX280) have been uncovered through microbenchmarks [Wong et al., 2010]. In particular, they have calculated the latencies and throughput of various floating-point and integer instructions. The results of these benchmarks are limited to the GTX280. However, as the code for the microbenchmarks is made available online[1], we have been able to run the benchmark code on the GTX480 with minor modifications.

The results of our experiments with the microbenchmark code for measuring the throughput of the various instructions are presented in Table 7.4. The measured throughput on the GTX280 is identical to those presented in [Wong et al., 2010]. We have also extended the benchmark to additionally measure the throughput of integer and floating-point (Boolean) comparison operations.

	throughput (ops/clock)	throughput (ops/clock)
Instruction type	GTX280	GTX480
fadd	7.9	28.1
fmul	11.2	28.1
iadd	7.9	16.0
imul	1.7	16.0
ixor	7.9	28.1
iand	7.9	28.1
`__sinf()`	2.0	4.0
`__cosf()`	2.0	4.0
`__expf()`	2.0	4.0
icmp	4.0	15.7
fcmp	2.7	15.7

Table 7.4: Instruction throughput microbenchmark results for GTX280 and GTX480

In order to calculate the total FLOP count when we are dealing with mixed integer and floating-point operation types, we use the measured throughput figures. The floating-point addition instruction is considered a single FLOP while that of other instructions is scaled according to the relative difference in throughput. Since we observe a different behaviour on the GTX480 for certain instruction types, we calculate separate FLOP counts for each platform.

`sampling_importance()`

The `sampling_importance()` kernel, as discussed in Chapter 6, calls two model-specific device functions for updating the particles and calculating the weights. However, in order to negate any possible effects of caching as both functions need to load certain particle variables, we have combined these two device functions into one.

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

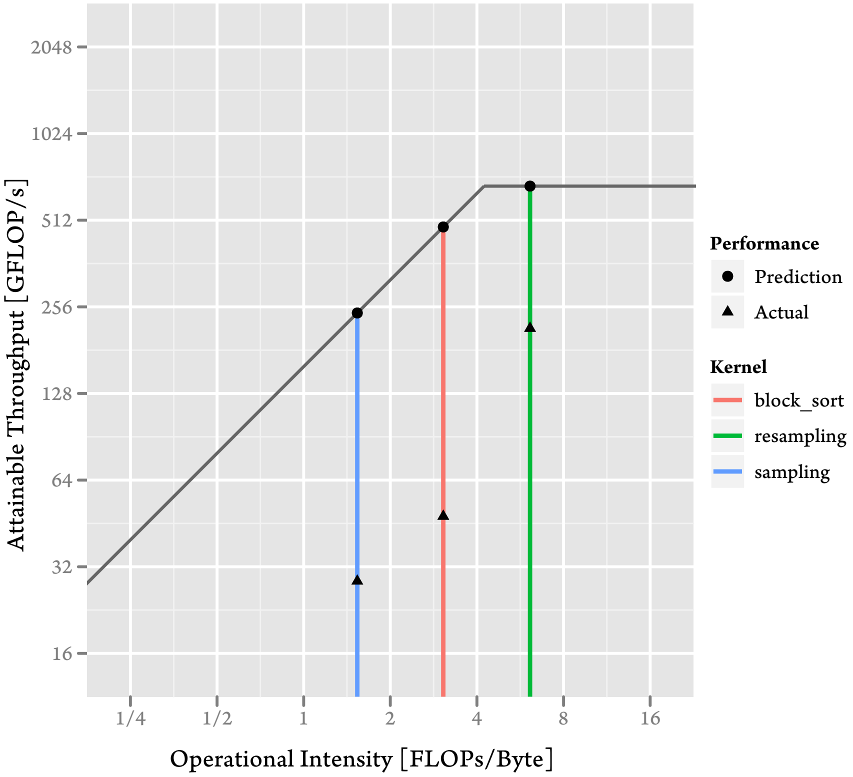


Figure 7.3: Roofline Model and Baseline Results - GTX480

Each thread reads one instance of the `state_data` structure, and writes back an updated version. With the chosen experiment parameters, this structure contains 9 single-precision floating-point variables (36B). For each state variable, we have a normally distributed random number (4B). Each thread also writes the weight of the particle to global memory, which is again a single-precision floating-point number (4B). This leads to a total of 36B + 36B = 72B of data read per thread and 36B + 4B = 40B of data written. Therefore, we have, in total, 112B of read/writes per thread.

Instruction type	count (parametric)	count	FLOP count GTX280	FLOP count GTX480
fadd	$12 + 6 \times \#\text{joints}$	42	42	42
fmul	$10 + 8 \times \#\text{joints}$	50	~36	50
`__sinf()`	#joints	5	~20	~36
`__cosf()`	#joints	5	~20	~36
`__expf()`	1	1	~4	~8
		total	122	172

Table 7.5: Instruction count for the `sampling_importance()` kernel

The instructions for the `sampling_importance()` kernel is listed in Table 7.5. This operational intensity is, according to these figures, $\frac{122}{112} = 1.09$ FLOPs/byte on the GTX280 and $\frac{172}{112} = 1.54$ FLOPs/byte

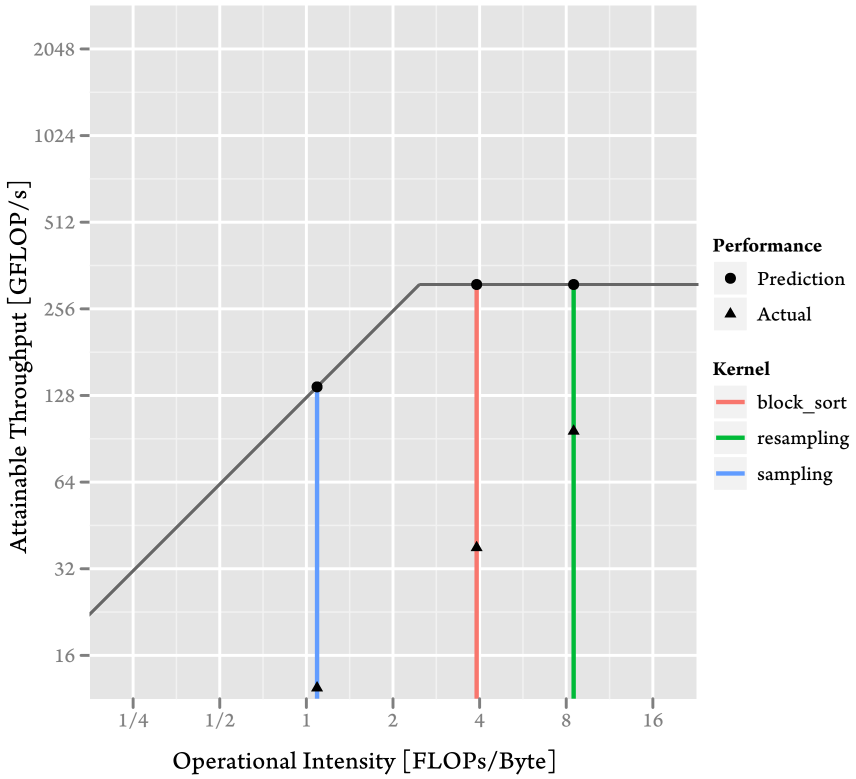


Figure 7.4: Roofline Model and Baseline Results - GTX280

on the GTX480. Both these figures are well below the point where the memory access latencies are hidden by computation. This kernel will ultimately be bound by the available memory throughput.

In order to figure our which roofline applies to this kernel, we examined the compiled binary using the NVIDIA provided disassembler. We noticed a great number of MAD instructions (FMA instructions on the GTX480) which can, in theory, run at twice the rate of other floating-point instructions. Nevertheless, because this kernel is ultimately memory bound, it does not change the predicted outcome.

The results of our measurements are presented in Table 7.8 and Table 7.9 and on the roofline model in Figure 7.3 and Figure 7.4. The measured memory throughput on both cards is far lower than what we expect to achieve. The reason for this is that none of the memory accesses are coalesced. Therefore, all the read from within a (half-)warp are serialised. In Section 7.3.1, we will discuss our modifications in order to achieve better memory throughput.

7.2.3 Local sort kernel: `block_sort()`

`block_sort()` uses a bitonic sorting network in order to sort the particles within each block according to their weight. Because the `state_data` structure is not guaranteed to fit the shared memory, we have opted to store only the weights as well as an index in shared memory. Once the weights are sorted, we apply the index to the actual particle data. In our bitonic sorting network, each thread handles two particles. Therefore, each threads needs to read 2× the `state_data` structure as well 2× the weight (`float`) and eventually write them back to global memory. This leads to a total of 160B read/writes per thread.

Table 7.6 presents the number of FLOPs per thread for this kernel. From this we can calculate the oper-

Instruction type	count	FLOP count GTX280	FLOP count GTX480
imul	53	~247	~94
iadd	90	90	~158
ixor	8	8	8
iand	53	53	53
icmp	53	~105	~95
fcmp	45	~132	~81
	total	635	489

Table 7.6: Instruction count for the `block_sort()` kernel

ational intensity: $\frac{635}{160} = 3.97$ FLOPs/byte for the GTX280 and $\frac{489}{160} = 3.06$ FLOPs/byte for the GTX480. In order to determine the computational bound for this kernel we examine whether MAD instructions are issued. Unlike the `sampling_importance()` kernel, we encountered no MAD instructions when we examined the CUDA binary with a disassembler. Therefore, we consider the lower roofline to be a realistic upper bound for the performance of this kernel.

Although these figures suggest that this kernel is compute bound on the GTX280 and memory bound on the GTX480, the irregular read access patterns prevent us from fully utilising the available memory bandwidth.

The experiment results are listed in Table 7.8 and Table 7.9. These results are also visualised on the roofline model in Figure 7.3 and Figure 7.4. These results indicate a very low memory bandwidth utilisation. As mentioned above, irregular memory access pattern prevents us from fully utilising the available bandwidth. In Section 7.3.2, we will discuss the options for reducing the uncoalesced memory transaction overhead.

7.2.4 Resampling kernel: `resampling()`

Each thread in the `resampling()` kernel reads 3× `float` and 1 `state_data` structure and writes 1 `state_data` structure to global memory. Therefore, we have to total of 84B read/writes per thread.

Instruction type	count	FLOP count GTX280	FLOP count GTX480
imul	108	~502	~190
iadd	108	108	~190
icmp	27	~54	~97
fmul	1	~1	1
fadd	18	18	18
fcmp	10	~30	~18
	total	713	514

Table 7.7: Instruction count for the `resampling()` kernel

The number of instructions for this kernel is presented in Table 7.7. From this we can calculate the operational intensity for the GTX280 as $\frac{713}{84} = 8.49$ FLOPs/byte and for the GTX480 as $\frac{514}{84} = 6.12$ FLOPs/byte. Similar to the `block_sort()` kernel, no MAD instructions are used, which results in this kernel being compute bound on both the GTX480 and GTX280. However, the memory accesses are highly irregular and thus prevent from fully utilising the available bandwidth. The experiment results listed in Table 7.8 and Table 7.9 as well as the roofline model in Figure 7.3 and Figure 7.4 confirm this to be the case. We will discuss optimising uncoalesced memory transactions in Section 7.3.2.

Kernel	Runtime	Mem throughput	Mem Util	Instr Throughput	Instr Util
sampling_importance	5889 µs	18.57 GB/s	11.5%	28.52 GFLOP/s	2.1%
block_sort	4984 µs	15.68 GB/s	9.7%	47.91 GFLOP/s	7.1%
resampling	2329 µs	35.22 GB/s	21.8%	215.52 GFLOP/s	32.0%

Table 7.8: Memory and instruction throughput and utilisation - GTX480

Kernel	Runtime	Mem throughput	Mem Util	Instr Throughput	Instr Util
sampling_importance	9659 µs	11.32 GB/s	9.0%	12.33 GFLOP/s	1.3%
block_sort	8185 µs	9.54 GB/s	7.6%	37.88 GFLOP/s	12.2%
resampling	7238 µs	11.33 GB/s	9.0%	96.20 GFLOP/s	30.9%

Table 7.9: Memory and instruction throughput and utilisation - GTX280

7.3 Optimisations

The analysis in the previous section indicates that even for compute bound kernels, irregular memory transactions resulted in suboptimal performance. In this section we focus our attention specifically to memory optimisations.

7.3.1 Enabling Memory Coalescing

Recall from Chapter 4, where we discussed the technical details of the GPU architecture, that the optimum memory bandwidth can only be achieved if memory accesses follow certain patterns which allow for coalesced memory loads. In this section, we will attempt to modify the memory layout of the data-structures to allow for better coalescing.

Depending on the particular GPU architecture, different requirements have to be met in order for global memory accesses withing a (half-)warp to be coalesced. For devices with compute capability 1.3 (e.g. GTX280) the word size needs to be either 1, 2, 4, 8 or 16 bytes and all the accessed elements within a half-warp need to lie in a single segment (segment size is 32-128 bytes depending on the word size).

Coalescing in 2.x devices is quite different because of the L1 and L2 caches. Global memory requests are broken down into 128-byte L1 cache requests. There are no alignment requirements.

The `state_data` structure which holds the particle data, even with a single joint, is larger than the maximum word size for coalescing on 1.x devices. Therefore, we need to modify the way we store this data. A common technique is to store the data in a *Structure of Arrays* (SoA) instead of the more obvious *Array of Structures* (AoS). In this scheme, each member of the original structure is stored in a separate array. All the variables in our model are of the type `float` (4B) which is guaranteed to be aligned, and therefore suitable for coalesced reads and writes.

Listing 7.1: Particle data stored in a SoA fashion

```
typedef struct _state
{
        float* angles;
        float* x;
        float* y;
        float* vX;
        float* vY;
} state;
```

One particular thing we need to consider is that since the `angles` array is actually two-dimensional, we need to store the angle values according to particle order and not angle order. The reason for this is that when all the threads from a single (half-)warp access a certain angle value, these reads must be contiguous for coalescing. The same reasoning holds for the array holding the generated random values.

Having discussed the SoA structure, we need to remember that from the three kernels discussed so far, only the `sampling_importance()` kernel lends itself particularly well to coalescing. In this step, each threads reads its own particle data and eventually writes it back without any communication with other threads. The other two kernels (`block_sort()`, and `resampling()`), however, have scattered read patterns which are not suited for coalescing.

The results of the experiments with the particle data stored in a SoA fashion are presented in Table 7.10 for the GTX480 and in Table 7.11 for the GTX280.

Kernel	Orig Runtime	Opt1 Runtime	Speedup	Mem Throughput	Mem Util	Instr Throughput	Instr Util
sampl_imp	5889 µs	809 µs	7.28×	135.20 GB/s	83.7%	207.63 GFLOP/s	15.4%
block_srt	4984 µs	1928 µs	2.59×	40.52 GB/s	25.1%	123.84 GFLOP/s	18.4%
resampling	2329 µs	1463 µs	1.59×	56.07 GB/s	34.7%	343.10 GFLOP/s	51.0%
total	14247 µs	5296 µs	2.69×				

Table 7.10: Memory optimisations: Speedup, throughput and utilisation - GTX480

Kernel	Orig Runtime	Opt1 Runtime	Speedup	Mem Throughput	Mem Util	Instr Throughput	Instr Util
sampl_imp	9659 µs	1038 µs	9.31×	105.37 GB/s	81.9%	114.78 GFLOP/s	18.5%
block_srt	8185 µs	6323 µs	1.29×	12.36 GB/s	9.8%	49.04 GFLOP/s	15.8%
resampling	7238 µs	5941 µs	1.22×	13.81 GB/s	11.0%	117.20 GFLOP/s	37.7%
total	26785 µs	14946 µs	1.79×				

Table 7.11: Memory optimisations: Speedup, throughput and utilisation - GTX280

As expected, the `sampling_importance()` kernel benefits the most from this particular optimisation. The results show that, in this kernel, we are now utilising most of the available memory bandwidth. In the following section, we will examine the ways we can optimise the other two kernels.

Listing 7.2: Particle data packed in float4s

```
typedef struct _state
{
        float4* angles1;
        float*  angles2;
        float4* pos;
} state;
```

7.3.2 Improving Uncoalesced Memory Accesses

The SoA structure discussed in the previous section had only a limited impact on the overall performance of the implementation as two important kernels have scattered read access patterns.

Recall from earlier discussion that global memory transactions have a minimum size of 32 bytes. Whenever we are reading scattered floats, we are effectively only using $\frac{1}{4}$ of the available bandwidth. One way to reduce this overhead is to store the data in 16 byte chunks. This will reduce the overhead to $\frac{1}{2}$ of the bandwidth. To this purpose we will use the built-in CUDA vector type float4 vector types, although the same could also be achieved with a custom structure and explicit alignment specifiers.

We modify the state_data structure to use float4 for storing the attributes. As mentioned in Table 7.2, All the experiments in this chapter have been run with 5 joints for the robotic arm. Therefore we still end up with a single float variable.

Kernel	Opt1 Runtime	Opt2 Runtime	Speedup	Mem Throughput	Mem Util	Instr Throughput	Instr Util
sampl_imp	809 µs	799 µs	1.01×	136.89 GB/s	84.7%	210.22 GFLOP/s	15.6%
block_srt	1928 µs	1924 µs	1.00×	40.61 GB/s	25.1%	124.10 GFLOP/s	18.5%
resampling	1463 µs	1536 µs	0.95×	53.41 GB/s	33.1%	326.79 GFLOP/s	48.6%
total	5296 µs	5312 µs	1.00×				

Table 7.12: Uncoalesced memory optimisations: Speedup, throughput and utilisation - GTX480

Kernel	Opt1 Runtime	Opt2 Runtime	Speedup	Mem Throughput	Mem Util	Instr Throughput	Instr Util
sampl_imp	1038 µs	1466 µs	0.71×	74.61 GB/s	59.4%	81.27 GFLOP/s	13.1%
block_srt	6323 µs	4172 µs	1.52×	18.73 GB/s	14.9%	74.32 GFLOP/s	23.9%
resampling	5941 µs	2927 µs	2.03×	28.03 GB/s	22.3%	237.88 GFLOP/s	76.5%
total	14946 µs	10161 µs	1.47×				

Table 7.13: Uncoalesced memory optimisations: Speedup, throughput and utilisation - GTX280

The results of these modifications are presented in Table 7.12 and Table 7.13. These results, together with the earlier discussed optimisations, are also visualised on the roofline model in Figure 7.5 and Figure 7.6. With these modifications, we do not see any improvement on the GTX480, but as expected, on the GTX280, the block_sort() and resampling() kernels benefit the most from this memory layout. However,

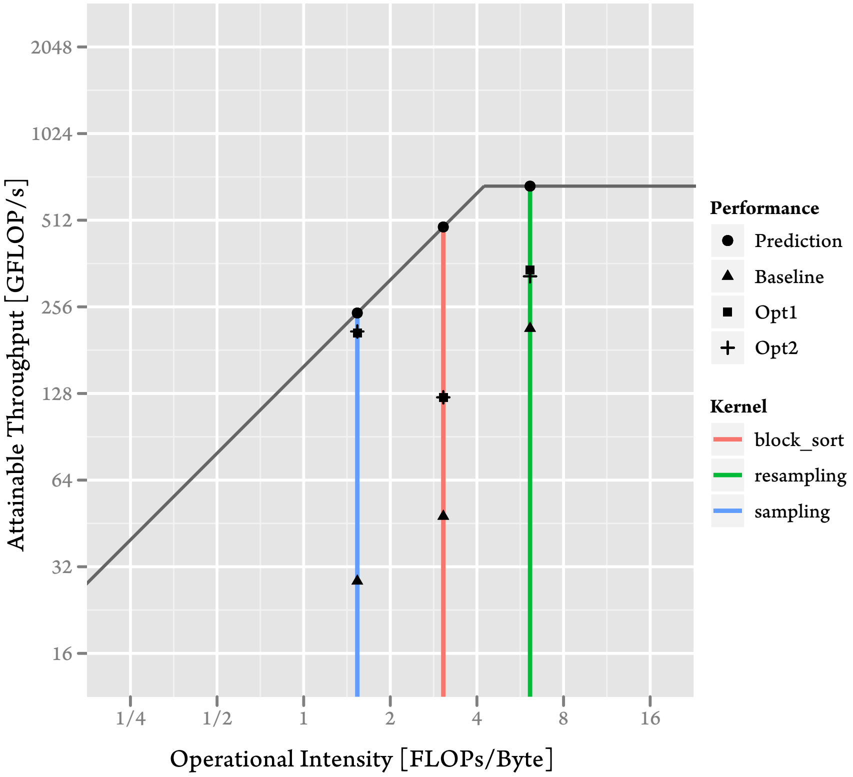


Figure 7.5: Roofline Model for Each Kernel - GTX480

on the GTX280, we observe a drop in performance in the `sampling()` where all memory accesses are coalesced. We were able to reproduce this drop in performance on the GTX280 with a simple `float4_copy()` kernel which only copies `float4` values from one array to another. We could only achieve 76% of the bandwidth of a similar kernel which copied 4× `float`s. This is consistent with the measured bandwidth for the `sampling()` kernel. This, however, does not occur with the `float2` data type, nor on the GTX480.

In order to confirm whether irregular memory access patterns are actually causing any performance degradation for `block_sort()` and `resampling()`, we modify both kernels to remove all irregular access patterns. We are careful that none of the computations for sorting in the `block_sort()` kernel, and the cumulative sum and binary search for the `resampling()` kernel are optimised away by the compiler. On the GTX480 this reduces the runtime of `block_sort()` to 1427 µs and for the GTX280 to 3440 µs. The runtime of `resampling()` is reduced to 1051 µs on the GTX480 and to 1634 µs on the GTX280.

These results, first of all, indicate that the performance hit of irregular global memory reads is far greater on the GTX280 than on the GTX480. The sophisticated two-layer global memory caching on the Fermi architecture clearly reduces the impact of scattered reads. More importantly, we can conclude from these results that there are other factors contributing to the gap between the theoretical maximum and the achieved throughput for these two kernels. Considering both these kernels extensively use the shared memory for sorting as well as parallel scan implementations, it is the most likely place to find potential inefficiencies.

In Chapter 4, we discussed that one of main issues regarding the use of shared memories are bank conflicts. If threads within a warp access memory locations on the same physical bank, a bank conflict occurs and these request are serialised. The CUDA profiler from NVIDIA can measure the number of occurred bank conflicts. Running the application through this profiler confirms a large number of shared memory bank conflicts for

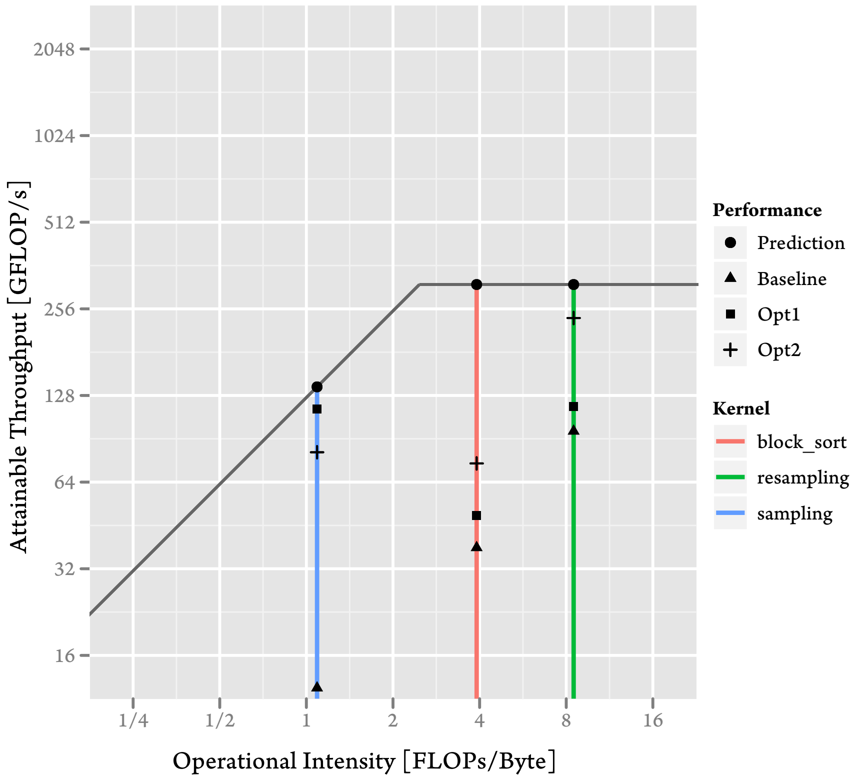


Figure 7.6: Roofline Model for Each Kernel - GTX280

both the `block_sort()` and `resampling()` kernels. Solving this issue is much easier for the parallel scan part where the access strides are much more regular. We do not, however, investigate this any further.

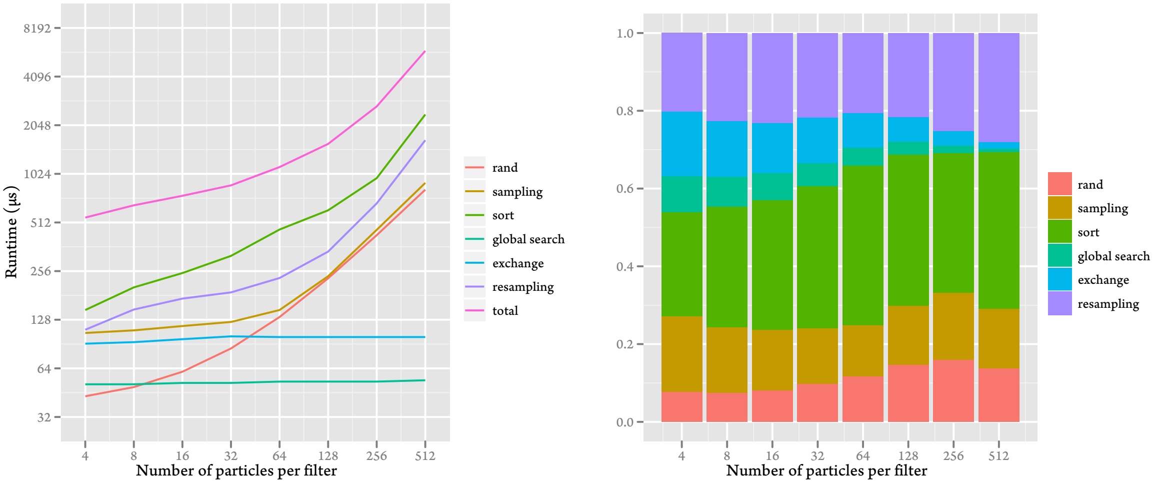
7.4 Scalability

In the previous sections, we have analysed the performance of the various parts of the particle filter algorithm with a fixed particle filter configuration. We now explore the performance of our implementation under varying conditions. We consider scaling in the following three directions: (i) number of particles per filter, (ii) number of particle filter blocks, and (iii) state dimensions. The results presented in this section are obtained on the GTX480, although the same scaling trends can be observed on the GTX280 as well.

7.4.1 Scaling the Number of Particles per Filter

The number of particles in a particle filter plays a crucial role in its estimation quality. In our distributed particle filter design, discussed in Chapter 3, we have introduced a two-level hierarchy for the particle population: (i) groups of particles forming a particle filter block, and (ii) a network of smaller particle filters. This design was introduced in order to overcome the inherent limitation of scaling individual particle filter.

The limited amount of shared memory and registers available to SMs, as well as the limitation of the number of threads within each thread-block are architectural constraints restricting the number of threads for each particle filter block. Nevertheless, we examine the performance of our filter implementation with the number of particles ranging from 4 to 512. The number of blocks is 2048.



(a)Runtime for each kernel

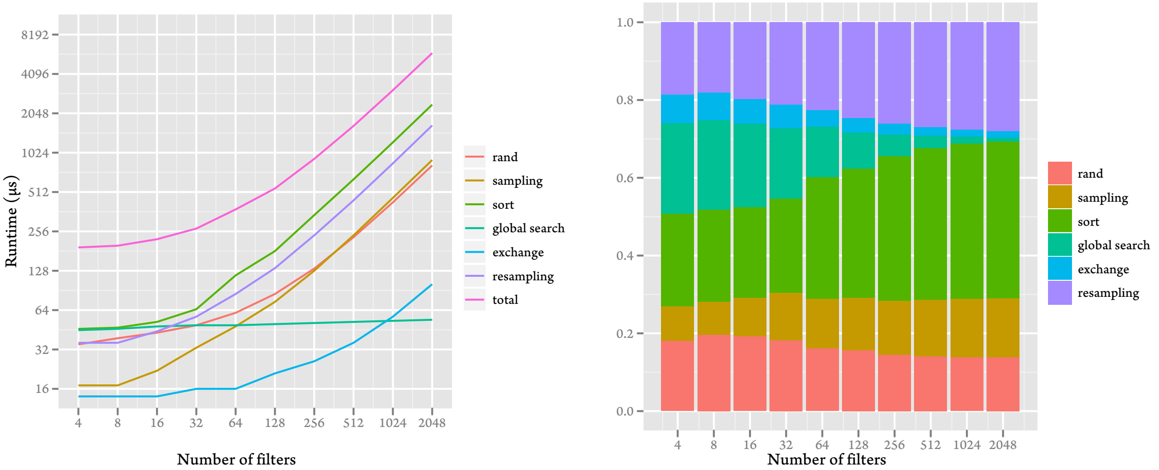
(b)Relative runtimes

Figure 7.7: Filter performance with varying number of particles per block

The results presented in Figure 7.7 indicate that the compute-heavy sorting and resampling stages clearly dominate the runtime when the number of particles increases. The particle exchange as well as the global search stages are, as expected, unaffected by the number of particles.

7.4.2 Scaling the Number of Particle Filter Blocks

In contrast to the number of particles per block, scaling the number of blocks is not bound by the same limitations. The usual boundary in this case is the amount of particle data we can store on global memory. For this experiment, we range the number of blocks from 4 to 2048. The number of particles per thread remains 512 for the whole experiment. The total number of particles, therefore, ranges from 2048 to 1048576.



(a)Runtime for each kernel

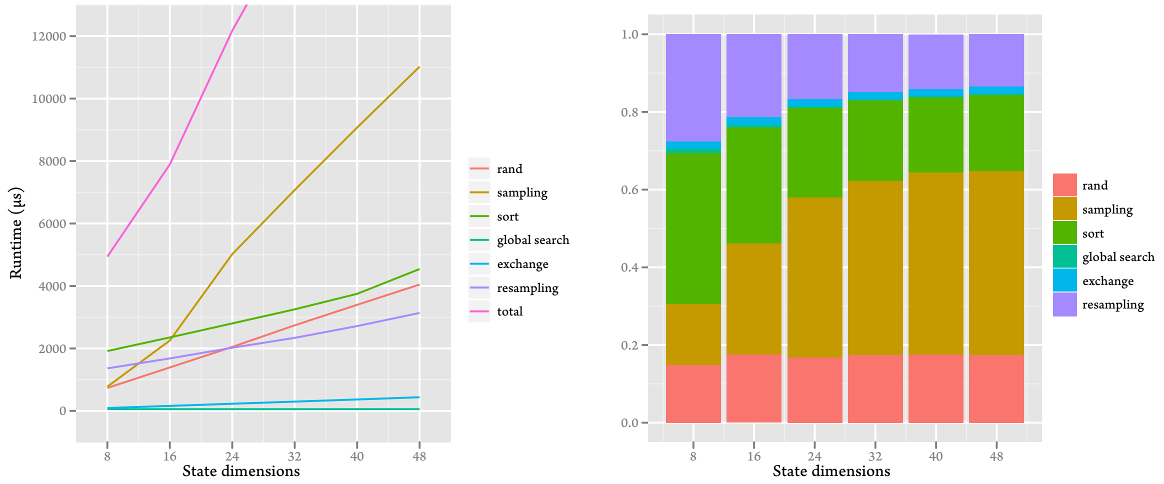
(b)Relative runtimes

Figure 7.8: Filter performance with varying number of particle filter blocks

Figure 7.8 depicts the results of this experiment. As with the previous section, the compute-heavy sorting and resampling stages dominate the runtime. The results confirm the runtime to scale linearly with respect to the number of particle filter blocks.

7.4.3 Scaling the State Dimensions

The scaling directions discussed in the previous two section involved filter parameters. There is, however, another important aspect which is the model-specific part of the filter (the `sampling_importance()` kernel). How efficiently this can be implemented on the GPU architecture depends on the model itself. The robotic arm application, used throughout this chapter, is particularly well suited for this purpose. By adjusting the number of joints of the robotic arm, we can simulate a smaller and/or larger problem. In this experiment, the state dimension ranges from 8 (4 joints) to 48 (44 joints).



(a)Runtime for each kernel

(b)Relative runtimes

Figure 7.9: Filter performance with varying state dimensions

From the experiment results, presented in Figure 7.9, we can conclude that with an increasing state dimension the efficient `sampling_importance()` becomes the dominant factor in the overall performance. As the problem becomes more complex, the "filtering" aspect becomes less relevant and the efficiency of the model implementation determines the overall runtime.

7.5 Discussion

In this chapter we examined the performance of our particle filter implementation on two GPU platforms. Using the roofline model, we determined performance upper bounds for each kernel. This enabled us to identify major performance bottlenecks.

Furthermore, we applied two memory optimisations in order to increase the implementation efficiency. The result of these two optimisations is a speedup of 2.69× for the GTX480 and 2.63× for the GTX280.

We analysed the scaling behaviour in three directions: (i) increasing the size of individual filters in the network, (ii) increasing the number of filters in the network and (iii) increasing problem size (state dimensions). The size of each particle filter can only increase up to a certain point as a result of platform limitations. The number of particle filters, however, does not suffer from the same limitations. The amount of global memory is the first platform limit we encounter when scaling up the number of particle filters. The runtime scales linearly with respect to the number of particle filters. With an increasing problem size, the model-specific sampling and weight calculations become the dominant factor in the total runtime. From this we can conclude that as the problem becomes more complex, the efficiency of the model implementation determines the overall runtime.

8 Filter Accuracy

In this chapter we use the robotic arm application described in Chapter 5 in order to examine the behaviour of our particle filter implementation under different scenarios. We are mainly interested in the filter estimation quality, and the effect of various model and filter parameters.

First, in Section 8.1, we will discuss the overall experiment setup used throughout this chapter. Section 8.2 presents the experiment results for a standard centralised particle filter implementation as a baseline for the other results, while Section 8.3 discusses that of our distributed particle filter implementation on the GPU. We conclude this chapter in Section 8.4 with an analysis of the results presented in this chapter. Appendix B presents a detailed overview of the results of the experiments of this chapter.

8.1 Filtering Scenario

We use the robotic arm application, described in Chapter 5, for the experiments throughout this chapter in order to examine the filter behaviour under different conditions. The chosen parameters for this model are listed in Table 8.1.

Number of joints	5
State dimension (#joints + 4)	9
Arm length	0.5m

Table 8.1: Robotic Arm Parameters

The simulated noise parameters are listed in Table 8.2. All noise terms are chosen to be Gaussian. This, however, is not a restriction imposed by the particle filter.

ω_θ	$\mathcal{N}(0, 0.1)$ rad/s
ω_k	$\mathcal{N}(0, 0.1)$ rad
$\omega_{uX}, \omega_{uY}, \omega_x, \omega_y$	$\mathcal{N}(0, 0.1)$ m
ω_{vX}, ω_{vY}	$\mathcal{N}(0, 0.1)$ m/s

Table 8.2: Simulation Noise Parameters

The object tracked by the camera follows a lemniscate path. The filter is run at 25 Hz. In order to compare different filter results, we calculate the estimation error e according to:

$$e = (x - \mu)^T C^{-1} (x - \mu)$$

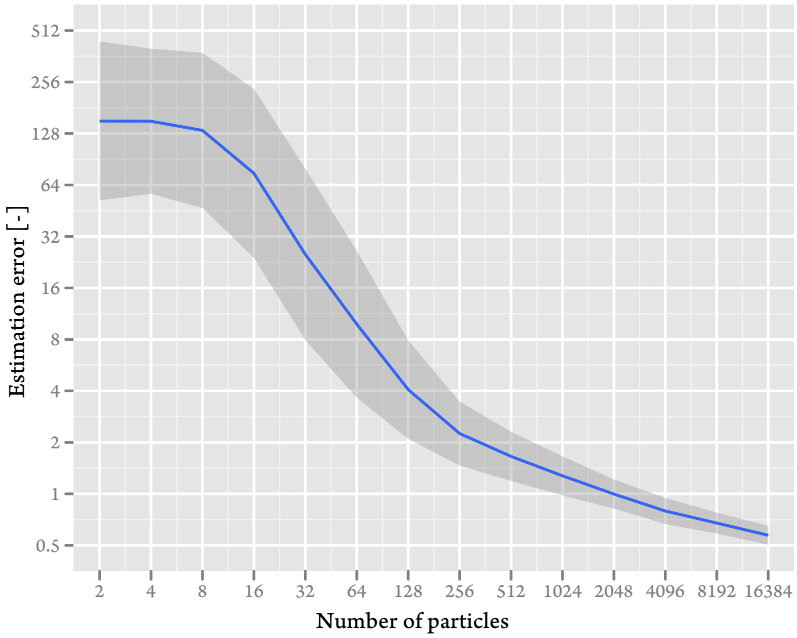


Figure 8.1: Estimation Error for a Centralised Particle Filter

where x is the state estimation vector, μ the actual state vector and C is a diagonal matrix with the expected standard deviations of the measurement errors. We consider $e < 1.0$ to be an acceptable estimation error for this experiment.

8.2 Centralised Particle Filter Results

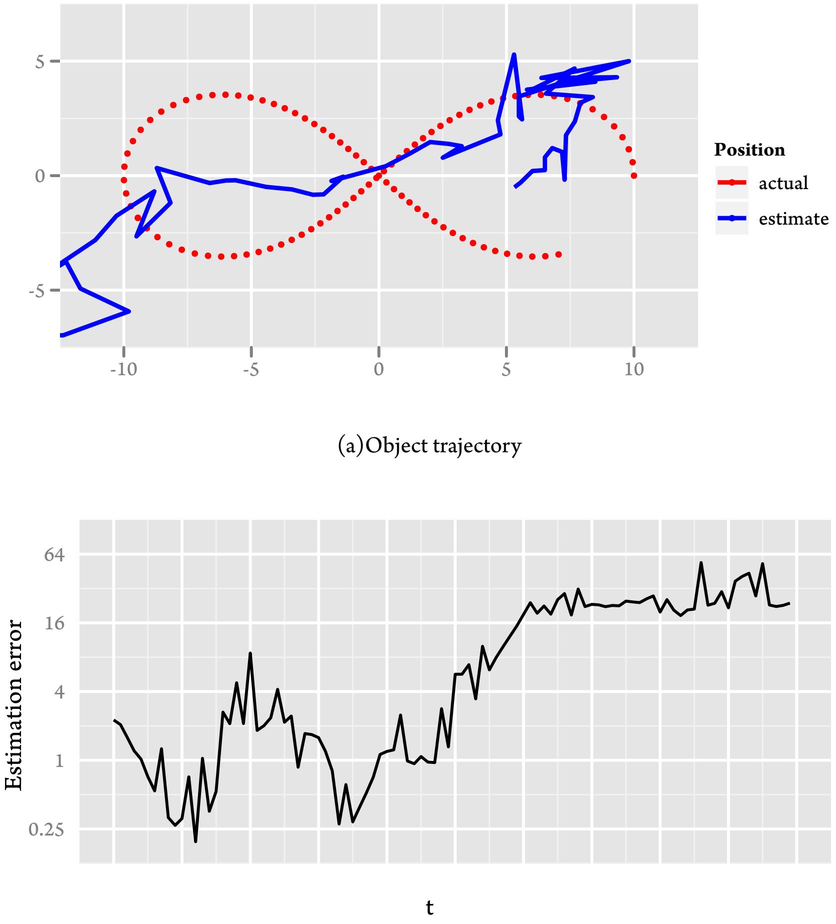
In order to have a reference for the filter performance, we start with a standard Sequential Importance Resampling particle filter implementation as depicted in Algorithm 2. We run this experiment with the number of particle varying between 2 and 16384.

Figure 8.1 presents the mean estimation error ± the standard deviation of 100 runs for each particle filter configuration. From these results it is clear that we need at least 4096 particles for an acceptable estimation. Although this sequential particle filter implementation is not optimised, with even 8192 particles, it is, on a modern desktop processor, a factor of 100 slower than what is needed for real-time estimation.

8.3 Distributed Particle Filter Results

In this section we will focus on the GPU particle filter implementation discussed in Chapter 6 and Chapter 7. The following filter parameters, explained in Section 3.4, each influence the filter estimation results in different ways:

- Filter network topology
- Filter network dimension
- Number of exchanged particles



(a)Object trajectory

(b)Overall estimation error

Figure 8.2: Filtering scenario with a low number of particles

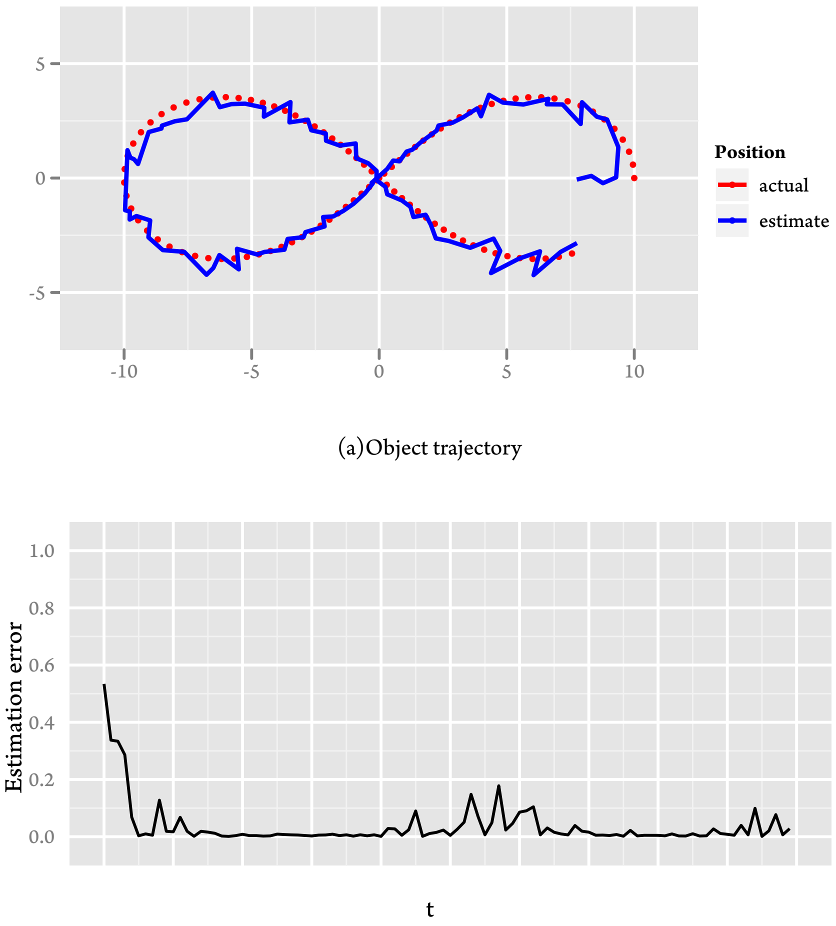
- Resampling frequency

With the exception of the first two parameters, we consider each parameter in isolation in order to gain a better understanding of its implications for the filtering process.

8.3.1 Filter Convergence

In this section we discuss an experiment in order to examine the filter convergence. As discussed in the experiment setup, the tracking object follows a lemniscate path. We initialise the filter to an incorrect state (one of the foci of the lemniscate). This experiment tests whether the filter converges to the actual path and can follow the object through the curves. The simulated noise terms are as described in Table 8.2. The filter parameters are listed in Table 8.3.

Figure 8.2 depicts the results of this experiment in a low particle count setting (16 filters each running 16 particles), while Figure 8.3 depicts those for a high particle count (256 filters each running 256 particles). These figures include a visual illustration of the actual and estimated object trajectory as well as the total estimation error (including joint angle measurements). This confirms our earlier results that this particular



(a) Object trajectory

(b) Overall estimation error

Figure 8.3: Filtering scenario with a high number of particles

problem requires a large number of particles for an accurate estimation, and furthermore, validates the filter correctness.

8.3.2 Filtering Frequency

One of our original goals was to enable real-time particle filtering for complex estimation problems. The runtime performance of the particle filter depends not only on the application-specific model update and importance weight calculations, but also on the filter configuration (e.g. number of concurrent filters, network topology). Chapter 7 contains a detailed analysis on the performance of the various parts of the algorithm.

The filter parameters are listed in Table 8.3. The number of particles for each particle filter is fixed at 512, while the number of filters ranges from 4 to 4096.

Figure 8.4 presents the results in the form of maximum achievable frequency for two GPU platforms. The achievable frequency is well beyond the requirements for this experiment. We can reach 1KHz frequencies with around 100000 particles in total.

Filter network topology	Ring
Number of exchanged particles	1
Resampling frequency	1.0

Table 8.3: Particle Filter Parameters

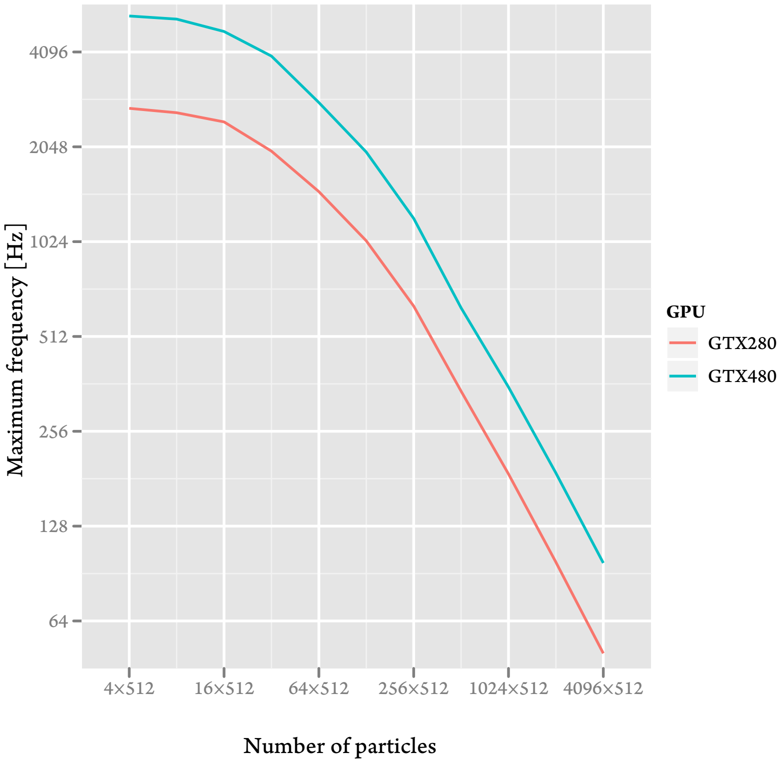


Figure 8.4: Maximum achievable particle filter frequency

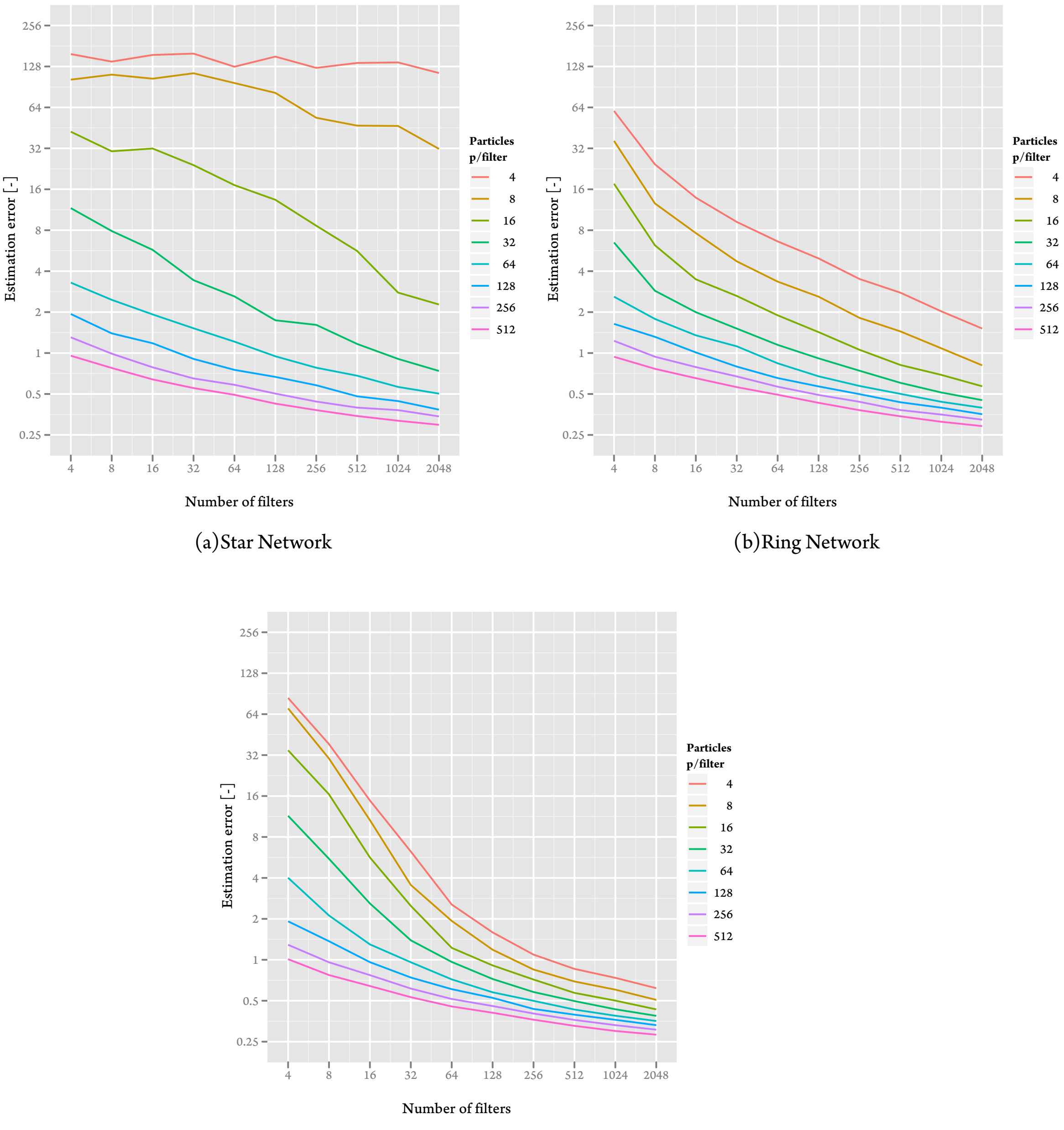
8.3.3 Filter Network Size and Topology

With centralised particle filters, the total number of particles is the most defining parameter for the estimation quality. This is equally true for distributed particle filters. Complex estimation problems with a large dimensional state vector require a huge number of particles in order for the random sampling process to produce particles near the correct state. In this section we perform a number of experiments in order to determine the behaviour of our distributed filter implementation with various filter dimensions.

Recall from Chapter 3 where we proposed a number of possible configurations for particle exchange within the particle filter network. We have observed that the way particles are exchanged highly effects the filter scaling behaviour. Therefore, we will study the effects of these two parameters together. We have chosen for the following network topologies:

1. Star

2. Ring



(a)Star Network

(b)Ring Network

(c)2D Torus Network

Figure 8.5: Estimation error with varying particle filter network size and topology

3. 2D Torus

The experiment setup is discussed in Section 8.1. The remaining filter parameters are chosen as follows: number of exchanged particles $t = 1$ and the resampling frequency $r = 1.0$.

The results of this experiment are presented in Figure 8.5. The presented estimation error is the average error of 100 runs for each configuration from the same simulation trace. From this we can clearly conclude that the star network topology performs the worst. The star network configuration results in a loss of diversity amongst the whole particle population as the same particles are fed into all filters. This loss of diversity results in a decreased filter performance.

The most interesting observation from both the ring and 2D torus network is that in all cases, a low number of particles can be compensated by adding more filters. This validates our strategy of dividing larger particle filters into a network of smaller filters. We also observe that with a low number of filters, the ring network outperform the 2D torus, while, a large number of filters favours the 2D torus network. The extra connections present in the 2D torus network allow a faster propagation of more likely particles in larger networks, while resulting in duplicate particles (and the loss of diversity) in smaller networks.

8.3.4 Particle Exchange

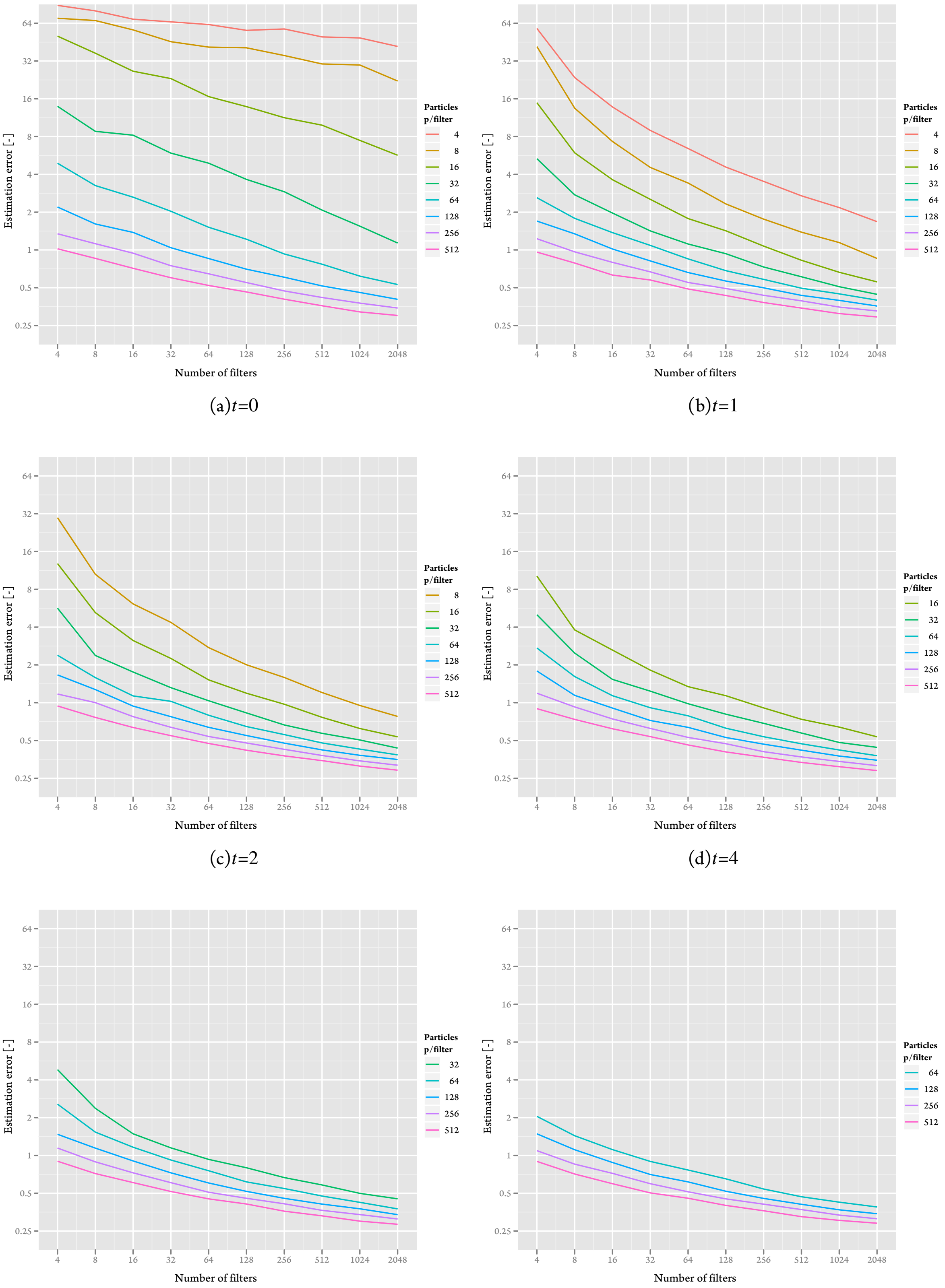
In this section we will examine the effects of the number of particles exchanged in each round t on the filter behaviour. To this end, we use the same robotic arm experiment with a ring network. The estimation error is calculated in the same fashion. We run this experiment for a number of different values for t, the number of exchanged particles for each filter with its neighbours. In a ring network, each filter receives $2t$ particles and sends back t particles. In order to prevent neighbouring filters from overriding each others particles, the number of particles for each filter, m, should be greater than the total number of particles sent and received: $m > 3t$. Therefore, certain parameter configurations are invalid.

Figure 8.6 illustrates the results of this experiment. The benefits of particle exchange is evident when we compare the results of the case where no particles are exchanged with the other results. Exchanging many particles does offer some minor improvements, but exchanging a single particle is generally sufficient for the likely particles to spread across the filter network.

8.3.5 Resampling Frequency

We discussed resampling in Section 2.3.1, where we mentioned the loss of diversity amongst the particle population as one of its major implications. In order to determine the effects of sampling too often or too little, we run the same experiment from the two previous sections with different resampling frequencies r. We run this experiment for a number of different values for r, ranging from $r = 0$ (no resampling) to $r = 1$ (always resampling). As with the previous two experiments, we calculate the average estimation error for 100 runs from the same trace for each configuration.

The results of this experiment are presented in Figure 8.7. It is quite evident from these results that resampling is an essential step to the whole particle filtering process. When no resampling takes place, adding more particles or filters does not improve the estimation significantly. Even infrequent resampling greatly improves the filter performance. The optimum resampling frequency, however, depends on the filter network size. Lower resampling frequencies favour low particle and low filter count settings (top left corner in the plots), while other combinations favour frequent resampling.



(a) t=0

(b) t=1

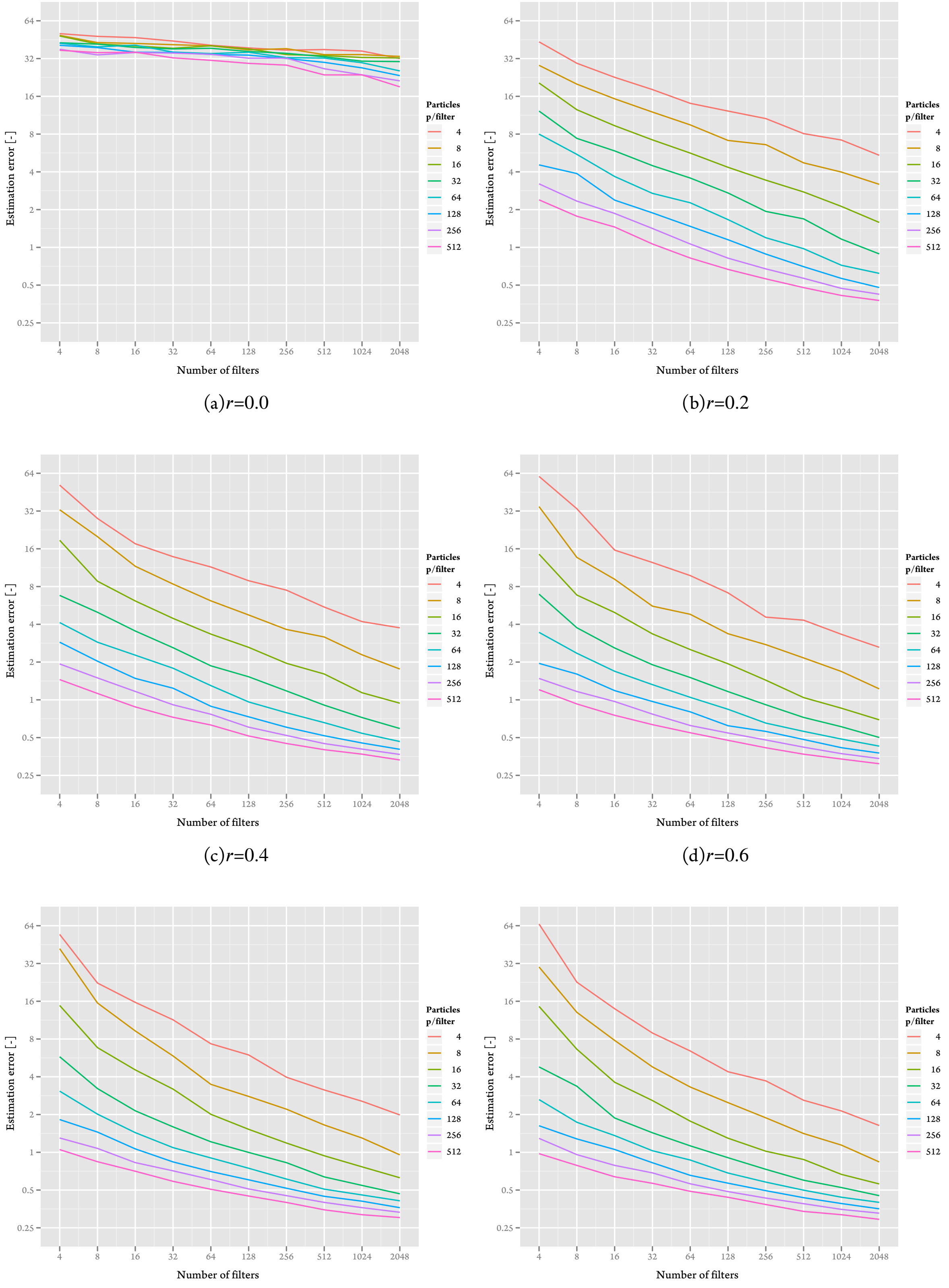
(c) t=2

(d) t=4

(e) t=8

(f) t=16

Figure 8.6: Estimation error with varying number of exchanged particles



(a)*r*=0.0

(b)*r*=0.2

(c)*r*=0.4

(d)*r*=0.6

(e)*r*=0.8

(f)*r*=1.0

Figure 8.7: Estimation error with varying resampling frequency

8.4 Discussion

In this chapter we examined various aspects of our particle filter implementation using the robotic arms application discussed in Chapter 5. Although unusable for any real-time purposes, the centralised particle filter implementation serves as a baseline for the distributed particle filter results.

The presented results confirm our GPU implementation to be suitable for real-time estimation for even larger problem sizes. Perhaps the most important results of this chapter is that, given a proper filter network topology, a network of particle filters can match (or even outperform) a single large centralised particle filter. Even minimal communication amongst the individual filters is sufficient to spread likely particles throughout the network.

There is, however, not a clear optimal configuration. Nevertheless, the general trend we have observed is that in low particle settings infrequent resampling, limited communication and a low connectivity network (e.g. ring) gives the best results. High particle settings tend to perform better with a more connected network (e.g. 2D torus), frequent resampling and increased communication.

9 Conclusions and Future Work

Real-time particle filtering, even with estimation problems which require millions of particles is possible with current generation consumer-grade GPUs. The results presented in this thesis attest to this. The suitability of the GPU for particular estimation problems also depends on problem-specific characteristics which are orthogonal to the choice of filter type.

In this thesis we have analysed the particle filter, a powerful Monte Carlo based Bayesian estimation technique, in order to find the right amount of parallelism for an efficient implementation on modern GPUs. Based on an earlier work of [Simonetto and Keviczky, 2009], we introduced algorithmic changes to the "classical" particle filter which allows for distributing the computation on separate execution units. Our modifications are not specific to GPUs, and thus can be used for efficient implementations on other multi-core platforms or even clusters.

With our changes to the particle filter algorithm, we introduced a number of parameters which affect the behaviour of the filter. With a number of experiments, we have been able to quantify the effects of these parameters, both on the estimation quality as well as the runtime.

We have analysed the performance of our GPU implementation, both analytically and empirically. Our analytical models are based on calculating performance upper bounds according to the operational intensity of the different kernels in our application. Empirically, we measured the effective utilisation of the GPU resources (i.e. memory bandwidth and instruction throughput) on two GPUs from different hardware generations. Using these results as a guideline for optimisation targets, we managed to increase the efficiency of our implementation using memory optimisations.

We believe this work can be extended in three directions. The first direction relates to the realisation of our particle filter design on other hardware platforms. These platform could range from conventional multicore processors to embedded platforms to grids. Each platform will present new challenges with regards to performance portability.

Another direction for future work is case studies with different types of estimation problems. We expect to gain a better understanding of the particle filter design with data available from more experiments.

Yet another direction for further research is focusing on the implementation efficiency, as there are still a number of kernels underutilising the available resources. This requires a better understanding of the GPU hardware.

Bibliography

[Arulampalam et al., 2002] Arulampalam, M. S., Maskell, S., Gordon, N., and Clapp, T. (2002). A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188. Cited on page 5.

[Bashi et al., 2003] Bashi, A. S., Jilkov, V. P., Li, X. R., and Chen, H. (2003). Distributed Implementations of Particle Filters. In *Proceedings of the Sixth International Conference of Information Fusion*, pages 1164–1171. Cited on page 12.

[Batcher, 1968] Batcher, K. E. (1968). Sorting networks and their applications. In *Proceedings of the 1968 spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA. ACM. Cited on page 32.

[Bolić et al., 2010] Bolić, M., Athalye, A., Hong, S., and Djurić, P. (2010). Study of Algorithmic and Architectural Characteristics of Gaussian Particle Filters. *Journal of Signal Processing Systems*, 61(2):205–218. Cited on page 12.

[Bolić et al., 2005] Bolić, M., Djurić, P. M., and Hong, S. (2005). Resampling Algorithms and Architectures for Distributed Particle Filters. *IEEE Transactions on Signal Processing*, 53(7):2442–2450. Cited on page 12.

[Box and Muller, 1958] Box, G. E. and Muller, M. E. (1958). A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610–611. Cited on page 31.

[Brun et al., 2002] Brun, O., Teuliere, V., and Garcia, J. M. (2002). Parallel Particle Filtering. *Journal of Parallel and Distributed Computing*, 62(7):1186–1202. Cited on pages 11 and 12.

[Demchik, 2011] Demchik, V. (2011). Pseudo-random number generators for Monte Carlo simulations on ATI Graphics Processing Units. *Computer Physics Communications*, 182(3):692–705. Cited on page 31.

[Doucet et al., 2000] Doucet, A., Godsill, S., and Andrieu, C. (2000). On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, 10:197–208. Cited on page 8.

[Fatahalian and Houston, 2008] Fatahalian, K. and Houston, M. (2008). A Closer Look at GPUs. *Communications of the ACM*, 51(10):50–57. Cited on page 17.

[Flynn, 1966] Flynn, M. J. (1966). Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909. Cited on page 18.

[Garland and Kirk, 2010] Garland, M. and Kirk, D. B. (2010). Understanding Throughput-Oriented Architectures. *Communications of the ACM*, 53(11):58–66. Cited on pages 17 and 18.

[Gordon et al., 1993] Gordon, N. J., Salmond, D. J., and Smith, A. F. M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F*, 140(2):107–113. Cited on page 7.

[Harris et al., 2007] Harris, M., Sengupta, S., and Owens, J. D. (2007). Parallel Prefix Sum (Scan) with CUDA. In Nguyen, H., editor, *GPU Gems 3*, chapter 39. Addison-Wesley Professional. Cited on page 34.

[Hendeby et al., 2010] Hendeby, G., Karlsson, R., and Gustafsson, F. (2010). Particle Filtering: The Need for Speed. *EURASIP Journal on Advances in Signal Processing*, 2010. Cited on page 12.

[Kalman, 1960] Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME - Journal of Basic Engineering*, 82:35–45. Cited on page 6.

[Khronos Group, 2010] Khronos Group (2010). *The OpenCL Specification*. Cited on page 19.

[Kotecha and Djuric, 2003] Kotecha, J. H. and Djuric, P. M. (2003). Gaussian Particle Filtering. *IEEE Transactions on Signal Processing*, 51(10):2592–2601. Cited on page 12.

[Lindholm et al., 2008] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. (2008). NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55. Cited on page 18.

[Marsaglia, 2003] Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software*, 8(14). Cited on page 31.

[Matsumoto and Nishimura, 1998a] Matsumoto, M. and Nishimura, T. (1998a). Dynamic Creation of Pseudorandom Number Generators. In *Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, pages 56–69. Cited on page 31.

[Matsumoto and Nishimura, 1998b] Matsumoto, M. and Nishimura, T. (1998b). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30. Cited on page 31.

[NVIDIA, 2006] NVIDIA (2006). *NVIDIA GeForce 8800 GPU Architecture Overview*. NVIDIA Corporation, Santa Clara, CA, USA. Cited on page 18.

[NVIDIA, 2010a] NVIDIA (2010a). *CUDA C Programming Guide*. NVIDIA Corporation, Santa Clara, CA, USA. Cited on pages 18 and 19.

[NVIDIA, 2010b] NVIDIA (2010b). *CUDA CURAND Library*. NVIDIA Corporation, Santa Clara, CA, USA. Cited on page 31.

[Podlozhnyuk, 2007] Podlozhnyuk, V. (2007). *Parallel Mersenne Twister*. NVIDIA Corporation, Santa Clara, CA, USA. Cited on page 31.

[Rosén et al., 2010] Rosén, O., Medvedev, A., and Ekman, M. (2010). Speedup and Tracking Accuracy Evaluation of Parallel Particle Filter Algorithms Implemented on a Multicore Architecture. In *2010 IEEE International Conference on Control Applications (CCA)*, pages 440–445. Cited on page 12.

[Saito, 2010] Saito, M. (2010). A Variant of Mersenne Twister Suitable for Graphic Processors. http://arxiv.org/abs/1005.4973. Cited on page 31.

[Simonetto and Keviczky, 2009] Simonetto, A. and Keviczky, T. (2009). Recent Developments in Distributed Particle Filtering: Towards Fast and Accurate Algorithms. In *1st IFAC Workshop on Estimation and Control of Networked Systems*. Cited on pages 2, 11, 13, 16, and 63.

[Swerling, 1958] Swerling, P. (1958). A proposed stagewise differential correction procedure for satellite tracking and prediction. Technical report, RAND Corporation, Boston, MA, USA. Cited on page 6.

[Thrun et al., 2005] Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic Robotics*. Intelligent robotics and autonomous agents. The MIT Press. Cited on page 5.

[West and Harrison, 1997] West, M. and Harrison, J. (1997). *Bayesian Forecasting and Dynamic Models*. Springer Series in Statistics. Springer-Verlag, New York, second edition. Cited on page 5.

[Williams et al., 2009] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76. Cited on page 37.

[Wong et al., 2010] Wong, H., Papadopoulou, M.-M., Sadooghi-Alvandi, M., and Moshovos, A. (2010). Demystifying GPU Microarchitecture through Microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 235–246. Cited on page 41.

[Wulf and McKee, 1995] Wulf, W. A. and McKee, S. A. (1995). Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24. Cited on page 18.

A

Unicycle Robot Localisation Implementation

The unicycle robot localisation problem has been introduced in Chapter 5. The complete implementation of this model for our CUDA-based particle filtering framework is presented.

Listing A.1: Unicycle Localisation Application

```
const int NUM_SENSORS = 16;

const float NOISE_VELOCITY = 0.2;
const float NOISE_ANGULAR_VELOCITY = 0.1;
const float NOISE_GAMMA = 0.01;

typedef struct _state
{
        float x;
        float y;
        float theta;
}
state;

typedef struct _control
{
        float velocity;
        float angular_velocity;
}
control;

typedef struct _measurement
{
        float distance[NUM_SENSORS];
}
measurement;


__device__  void sampling(
        state* const particle_data,
        const control control_data,
        const float* const d_random,
        const float dt)
{
        const float x = particle_data->x;
        const float y = particle_data->y;
        const float theta = particle_data->theta;

        const float velocity = control_data.velocity +
                NOISE_VELOCITY * d_random[0];
        const float angular_velocity =
                control_data.angular_velocity +
                NOISE_ANGULAR_VELOCITY * d_random[1];
        const float gamma = NOISE_GAMMA * d_random[2];

        particle_data->x = x +
                velocity / angular_velocity *
                (sinf(theta+angular_velocity*dt) - sinf(theta));
        particle_data->y = y -
                velocity / angular_velocity *
                (cosf(theta+angular_velocity*dt) - cosf(theta));
        particle_data->theta = theta + angular_velocity*dt + gamma*dt;
}

__device__ float importance_weight(
        const state* const particle_data,
        const measurement measurement_data)
{
        const float x = particle_data->x;
        const float y = particle_data->y;

        const float norm_factor =
                1.0f / powf(2.0f*((float)M_PI), NUM_SENSORS/2);

        float value=0;

        for (int i=0; i<NUM_SENSORS; ++i)
        {
                const float d1=x-d_sensor_position_x[i];
                const float d2=y-d_sensor_position_y[i];
                const float d3=  d_sensor_position_z[i];

                const float vect=sqrtf(d1*d1+d2*d2+d3*d3) -
                        measurement_data.distance[i];
                value += vect*vect;
        }

        return norm_factor*expf(-value);
}
```

B Robotic Arm Filtering Results

The results presented here are obtained with the robotic arm application discussed in Chapter 5. The experiment setup and parameters are presented in Chapter 8.

B.1 Centralised Particle Filter

Number of particles						
2	4	8	16	32	64	128
$\mu = 242.4884$	$\mu = 229.8312$	$\mu = 222.7339$	$\mu = 139.313$	$\mu = 46.21315$	$\mu = 16.99719$	$\mu = 5.2109$
$\sigma = 217.788$	$\sigma = 207.014$	$\sigma = 247.090$	$\sigma = 177.815$	$\sigma = 57.767$	$\sigma = 27.148$	$\sigma = 4.287$
256	512	1024	2048	4096	8192	16384
$\mu = 2.486124$	$\mu = 1.749702$	$\mu = 1.320456$	$\mu = 1.017577$	$\mu = 0.8061942$	$\mu = 0.681618$	$\mu = 0.5773173$
$\sigma = 1.226$	$\sigma = 0.602$	$\sigma = 0.360$	$\sigma = 0.218$	$\sigma = 0.149$	$\sigma = 0.096$	$\sigma = 0.075$

Table B.1: Estimation error for centralised particle filter

B.2 Distributed Particle Filter

		Number of particles per filter							
		4	8	16	32	64	128	256	512
	4	$\mu = 227.9524$ $\sigma = 207.918$	$\mu = 173.0753$ $\sigma = 175.547$	$\mu = 74.24901$ $\sigma = 108.088$	$\mu = 27.85484$ $\sigma = 69.770$	$\mu = 4.471158$ $\sigma = 4.331$	$\mu = 2.110396$ $\sigma = 0.884$	$\mu = 1.353138$ $\sigma = 0.380$	$\mu = 0.9899719$ $\sigma = 0.274$
	8	$\mu = 222.0341$ $\sigma = 223.819$	$\mu = 165.2306$ $\sigma = 144.396$	$\mu = 60.2252$ $\sigma = 83.377$	$\mu = 13.35894$ $\sigma = 20.534$	$\mu = 3.02162$ $\sigma = 2.514$	$\mu = 1.486802$ $\sigma = 0.556$	$\mu = 1.032647$ $\sigma = 0.317$	$\mu = 0.7919986$ $\sigma = 0.164$
	16	$\mu = 224.4179$ $\sigma = 193.451$	$\mu = 172.2834$ $\sigma = 178.332$	$\mu = 63.76622$ $\sigma = 83.083$	$\mu = 14.04086$ $\sigma = 55.207$	$\mu = 2.225712$ $\sigma = 1.289$	$\mu = 1.255043$ $\sigma = 0.459$	$\mu = 0.8117802$ $\sigma = 0.213$	$\mu = 0.6481496$ $\sigma = 0.104$
	32	$\mu = 214.5085$ $\sigma = 167.475$	$\mu = 165.6956$ $\sigma = 151.601$	$\mu = 64.13282$ $\sigma = 98.222$	$\mu = 5.006643$ $\sigma = 5.906$	$\mu = 1.681571$ $\sigma = 0.889$	$\mu = 0.9575123$ $\sigma = 0.338$	$\mu = 0.6675079$ $\sigma = 0.159$	$\mu = 0.5603728$ $\sigma = 0.094$
	64	$\mu = 209.0884$ $\sigma = 224.764$	$\mu = 169.2819$ $\sigma = 158.555$	$\mu = 42.90776$ $\sigma = 56.475$	$\mu = 3.295621$ $\sigma = 2.933$	$\mu = 1.29523$ $\sigma = 0.462$	$\mu = 0.7818636$ $\sigma = 0.232$	$\mu = 0.5961169$ $\sigma = 0.123$	$\mu = 0.498006$ $\sigma = 0.072$
Num filters	128	$\mu = 214.903$ $\sigma = 195.308$	$\mu = 146.8859$ $\sigma = 146.550$	$\mu = 35.76919$ $\sigma = 56.555$	$\mu = 2.110745$ $\sigma = 1.475$	$\mu = 1.023817$ $\sigma = 0.471$	$\mu = 0.692682$ $\sigma = 0.188$	$\mu = 0.5128466$ $\sigma = 0.102$	$\mu = 0.4293393$ $\sigma = 0.063$
	256	$\mu = 204.1097$ $\sigma = 197.087$	$\mu = 130.7894$ $\sigma = 149.786$	$\mu = 24.03496$ $\sigma = 41.596$	$\mu = 1.921983$ $\sigma = 1.343$	$\mu = 0.8180299$ $\sigma = 0.257$	$\mu = 0.5975755$ $\sigma = 0.155$	$\mu = 0.4474109$ $\sigma = 0.086$	$\mu = 0.3841442$ $\sigma = 0.053$
	512	$\mu = 200.337$ $\sigma = 176.888$	$\mu = 127.8728$ $\sigma = 191.216$	$\mu = 19.45182$ $\sigma = 42.733$	$\mu = 1.280827$ $\sigma = 0.558$	$\mu = 0.7182555$ $\sigma = 0.255$	$\mu = 0.4910907$ $\sigma = 0.103$	$\mu = 0.403475$ $\sigma = 0.074$	$\mu = 0.3469468$ $\sigma = 0.043$
	1024	$\mu = 199.3808$ $\sigma = 173.191$	$\mu = 121.1254$ $\sigma = 129.980$	$\mu = 8.217067$ $\sigma = 23.422$	$\mu = 1.015018$ $\sigma = 0.577$	$\mu = 0.582461$ $\sigma = 0.155$	$\mu = 0.4532942$ $\sigma = 0.098$	$\mu = 0.3862672$ $\sigma = 0.068$	$\mu = 0.3208877$ $\sigma = 0.044$
	2048	$\mu = 190.4131$ $\sigma = 202.827$	$\mu = 115.5985$ $\sigma = 173.571$	$\mu = 5.947686$ $\sigma = 15.493$	$\mu = 0.8089744$ $\sigma = 0.421$	$\mu = 0.5169247$ $\sigma = 0.127$	$\mu = 0.391295$ $\sigma = 0.076$	$\mu = 0.34602$ $\sigma = 0.053$	$\mu = 0.2990087$ $\sigma = 0.037$

Table B.2: Estimation error for star network

		Number of particles per filter							
		4	8	16	32	64	128	256	512
Num filters	4	$\mu = 102.5398$ $\sigma = 133.782$	$\mu = 62.50196$ $\sigma = 86.901$	$\mu = 34.0523$ $\sigma = 76.424$	$\mu = 10.53582$ $\sigma = 14.836$	$\mu = 3.00443$ $\sigma = 1.839$	$\mu = 1.727579$ $\sigma = 0.612$	$\mu = 1.280384$ $\sigma = 0.385$	$\mu = 0.957506$ $\sigma = 0.195$
	8	$\mu = 38.71661$ $\sigma = 46.364$	$\mu = 20.66024$ $\sigma = 21.509$	$\mu = 8.169698$ $\sigma = 6.709$	$\mu = 3.37399$ $\sigma = 2.257$	$\mu = 1.926268$ $\sigma = 0.864$	$\mu = 1.384282$ $\sigma = 0.475$	$\mu = 0.9588313$ $\sigma = 0.195$	$\mu = 0.7770571$ $\sigma = 0.138$
	16	$\mu = 17.8298$ $\sigma = 11.624$	$\mu = 10.62569$ $\sigma = 8.902$	$\mu = 4.550879$ $\sigma = 4.080$	$\mu = 2.167783$ $\sigma = 0.977$	$\mu = 1.433354$ $\sigma = 0.557$	$\mu = 1.03543$ $\sigma = 0.237$	$\mu = 0.7995505$ $\sigma = 0.137$	$\mu = 0.6614142$ $\sigma = 0.109$
	32	$\mu = 11.31487$ $\sigma = 6.711$	$\mu = 5.714578$ $\sigma = 4.024$	$\mu = 2.939995$ $\sigma = 1.650$	$\mu = 1.584928$ $\sigma = 0.497$	$\mu = 1.163994$ $\sigma = 0.342$	$\mu = 0.8065948$ $\sigma = 0.137$	$\mu = 0.6794544$ $\sigma = 0.095$	$\mu = 0.5659443$ $\sigma = 0.072$
	64	$\mu = 7.539837$ $\sigma = 4.148$	$\mu = 3.657734$ $\sigma = 1.616$	$\mu = 2.013334$ $\sigma = 0.685$	$\mu = 1.189128$ $\sigma = 0.322$	$\mu = 0.8561569$ $\sigma = 0.169$	$\mu = 0.6632572$ $\sigma = 0.110$	$\mu = 0.5694271$ $\sigma = 0.067$	$\mu = 0.4966007$ $\sigma = 0.053$
	128	$\mu = 5.502683$ $\sigma = 2.637$	$\mu = 2.841592$ $\sigma = 1.238$	$\mu = 1.509556$ $\sigma = 0.560$	$\mu = 0.9354803$ $\sigma = 0.217$	$\mu = 0.686675$ $\sigma = 0.138$	$\mu = 0.5727136$ $\sigma = 0.071$	$\mu = 0.4951787$ $\sigma = 0.057$	$\mu = 0.4326917$ $\sigma = 0.043$
	256	$\mu = 3.805812$ $\sigma = 1.595$	$\mu = 1.958428$ $\sigma = 0.818$	$\mu = 1.088706$ $\sigma = 0.283$	$\mu = 0.7570641$ $\sigma = 0.163$	$\mu = 0.5775182$ $\sigma = 0.075$	$\mu = 0.5029114$ $\sigma = 0.066$	$\mu = 0.4397668$ $\sigma = 0.039$	$\mu = 0.3819917$ $\sigma = 0.039$
	512	$\mu = 2.943974$ $\sigma = 0.980$	$\mu = 1.510945$ $\sigma = 0.493$	$\mu = 0.8394746$ $\sigma = 0.205$	$\mu = 0.6115893$ $\sigma = 0.100$	$\mu = 0.5046428$ $\sigma = 0.060$	$\mu = 0.4367984$ $\sigma = 0.043$	$\mu = 0.3818363$ $\sigma = 0.029$	$\mu = 0.3439244$ $\sigma = 0.029$
	1024	$\mu = 2.142178$ $\sigma = 0.717$	$\mu = 1.132082$ $\sigma = 0.342$	$\mu = 0.7044835$ $\sigma = 0.148$	$\mu = 0.5165899$ $\sigma = 0.063$	$\mu = 0.4399009$ $\sigma = 0.041$	$\mu = 0.3983444$ $\sigma = 0.039$	$\mu = 0.3543179$ $\sigma = 0.030$	$\mu = 0.3133846$ $\sigma = 0.026$
	2048	$\mu = 1.601057$ $\sigma = 0.561$	$\mu = 0.8341214$ $\sigma = 0.204$	$\mu = 0.5756661$ $\sigma = 0.086$	$\mu = 0.4530789$ $\sigma = 0.049$	$\mu = 0.3974585$ $\sigma = 0.034$	$\mu = 0.3564218$ $\sigma = 0.031$	$\mu = 0.3246974$ $\sigma = 0.026$	$\mu = 0.2911823$ $\sigma = 0.023$

Table B.3: Estimation error for ring network

		Number of particles per filter							
		4	8	16	32	64	128	256	512
Num filters	4	$\mu = 154.575$ $\sigma = 192.636$	$\mu = 124.0169$ $\sigma = 159.728$	$\mu = 70.97812$ $\sigma = 111.622$	$\mu = 22.24618$ $\sigma = 34.030$	$\mu = 6.950467$ $\sigma = 12.642$	$\mu = 2.149334$ $\sigma = 1.136$	$\mu = 1.371315$ $\sigma = 0.515$	$\mu = 1.036454$ $\sigma = 0.249$
	8	$\mu = 70.73076$ $\sigma = 102.110$	$\mu = 62.1337$ $\sigma = 145.064$	$\mu = 32.89054$ $\sigma = 52.159$	$\mu = 8.205879$ $\sigma = 7.703$	$\mu = 2.445901$ $\sigma = 1.618$	$\mu = 1.471483$ $\sigma = 0.617$	$\mu = 0.9980664$ $\sigma = 0.310$	$\mu = 0.7907979$ $\sigma = 0.177$
	16	$\mu = 20.97834$ $\sigma = 16.383$	$\mu = 16.71037$ $\sigma = 21.835$	$\mu = 8.571457$ $\sigma = 8.565$	$\mu = 3.577263$ $\sigma = 4.152$	$\mu = 1.395995$ $\sigma = 0.539$	$\mu = 1.003476$ $\sigma = 0.313$	$\mu = 0.7978615$ $\sigma = 0.228$	$\mu = 0.6503318$ $\sigma = 0.108$
	32	$\mu = 9.07093$ $\sigma = 7.748$	$\mu = 5.217487$ $\sigma = 5.410$	$\mu = 3.675169$ $\sigma = 4.443$	$\mu = 1.520823$ $\sigma = 0.731$	$\mu = 1.016102$ $\sigma = 0.380$	$\mu = 0.7578095$ $\sigma = 0.169$	$\mu = 0.6224237$ $\sigma = 0.103$	$\mu = 0.5358646$ $\sigma = 0.073$
	64	$\mu = 3.462152$ $\sigma = 3.460$	$\mu = 2.687894$ $\sigma = 3.535$	$\mu = 1.392487$ $\sigma = 1.324$	$\mu = 1.008158$ $\sigma = 0.324$	$\mu = 0.7404839$ $\sigma = 0.197$	$\mu = 0.6173912$ $\sigma = 0.110$	$\mu = 0.5190756$ $\sigma = 0.065$	$\mu = 0.4568045$ $\sigma = 0.052$
	128	$\mu = 1.737855$ $\sigma = 0.855$	$\mu = 1.262515$ $\sigma = 0.495$	$\mu = 0.9536437$ $\sigma = 0.315$	$\mu = 0.7441259$ $\sigma = 0.194$	$\mu = 0.588911$ $\sigma = 0.128$	$\mu = 0.5301779$ $\sigma = 0.075$	$\mu = 0.4602782$ $\sigma = 0.055$	$\mu = 0.4103854$ $\sigma = 0.045$
	256	$\mu = 1.144598$ $\sigma = 0.391$	$\mu = 0.8779246$ $\sigma = 0.244$	$\mu = 0.7322191$ $\sigma = 0.159$	$\mu = 0.5869851$ $\sigma = 0.096$	$\mu = 0.5037322$ $\sigma = 0.069$	$\mu = 0.4387583$ $\sigma = 0.059$	$\mu = 0.4047316$ $\sigma = 0.042$	$\mu = 0.3638851$ $\sigma = 0.028$
	512	$\mu = 0.885039$ $\sigma = 0.239$	$\mu = 0.7029075$ $\sigma = 0.125$	$\mu = 0.5769665$ $\sigma = 0.082$	$\mu = 0.502267$ $\sigma = 0.071$	$\mu = 0.4334683$ $\sigma = 0.050$	$\mu = 0.3967512$ $\sigma = 0.038$	$\mu = 0.3625083$ $\sigma = 0.030$	$\mu = 0.3282409$ $\sigma = 0.030$
	1024	$\mu = 0.7553266$ $\sigma = 0.177$	$\mu = 0.6104205$ $\sigma = 0.096$	$\mu = 0.505615$ $\sigma = 0.069$	$\mu = 0.4353883$ $\sigma = 0.049$	$\mu = 0.3897054$ $\sigma = 0.039$	$\mu = 0.3634693$ $\sigma = 0.034$	$\mu = 0.3317855$ $\sigma = 0.027$	$\mu = 0.3007419$ $\sigma = 0.027$
	2048	$\mu = 0.6287862$ $\sigma = 0.116$	$\mu = 0.5112716$ $\sigma = 0.066$	$\mu = 0.4342013$ $\sigma = 0.046$	$\mu = 0.3893534$ $\sigma = 0.043$	$\mu = 0.3559662$ $\sigma = 0.030$	$\mu = 0.332184$ $\sigma = 0.031$	$\mu = 0.3072191$ $\sigma = 0.025$	$\mu = 0.282078$ $\sigma = 0.019$

Table B.4: Estimation error for 2D torus network