PARALLEL RESAMPLING METHODS FOR PARTICLE FILTERS ON GRAPHICS PROCESSING UNIT

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÖZCAN DÜLGER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

DECEMBER 2017

Approval of the thesis:

PARALLEL RESAMPLING METHODS FOR PARTICLE FILTERS ON GRAPHICS PROCESSING UNIT

submitted by **ÖZCAN DÜLGER** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Halit Oğuztüzün _____
Head of Department, **Computer Engineering**

Prof. Dr. Halit Oğuztüzün _____
Supervisor, **Computer Engineering Department, METU**

Prof. Dr. Mübeccel Demirekler _____
Co-supervisor, **Electrical and Electronics Eng. Dept., METU**

Examining Committee Members:

Assist. Prof. Dr. Emre Akbaş
Computer Engineering Department, METU _____

Prof. Dr. Halit Oğuztüzün
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Murat Manguoğlu
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Özcan Öztürk
Computer Engineering Department, Bilkent University _____

Assist. Prof. Dr. Adnan Özsoy
Computer Engineering Department, Hacettepe University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ÖZCAN DÜLGER

Signature :

ABSTRACT

PARALLEL RESAMPLING METHODS FOR PARTICLE FILTERS ON GRAPHICS PROCESSING UNIT

Dülger, Özcan

Ph.D., Department of Computer Engineering

Supervisor : Prof. Dr. Halit Oğuztüzün

Co-Supervisor : Prof. Dr. Mübeccel Demirekler

December 2017, 140 pages

This thesis addresses the implementation of the resampling stage of the particle filter on graphics processing unit (GPU). Some of the well-known sequential resampling methods are the Multinomial, Stratified and Systematic resampling. They have dependency in their loop structure which impedes their parallel implementation. Although such impediments were overcome on their GPU implementation, these algorithms suffer from numerical instability due to the accumulation of rounding errors when single precision is used. Rounding errors arise in cumulative summation over the weights of the particles when the weights differ widely or the number of particles is large. There are resampling algorithms such as Metropolis and Rejection, which do not suffer from numerical instability as they only calculate ratios of weights pairwise rather than perform collective operations over the weights. They are more suitable for the GPU implementation of the particle filter. However, they suffer from non-

coalesced global memory access patterns which cause their speed deteriorate rapidly as the number of particles gets large. In the first part of this thesis, we offer solutions for this problem of the Metropolis resampling. We introduce two implementation techniques, designated Metropolis-C1 and Metropolis-C2, and compare them with the original Metropolis resampling on NVIDIA Tesla K40 board. In the first scenario where these two techniques achieve their fastest execution times, Metropolis-C1 is faster than the others, but yields the worst results in quality. However, Metropolis-C2 is closer to the Metropolis resampling in quality. In the second scenario where all three algorithms yield similar quality, although Metropolis-C1 and Metropolis-C2 are slower, they are still faster than the original Metropolis resampling. In the second part of the thesis, we introduce a new resampling method, designated Uphill resampling, which is free from numerical instability as it avoids the accumulation of rounding errors. We make comparisons with the Systematic, Metropolis and Rejection resampling methods with respect to quality and speed. We achieve similar results with the Metropolis and Rejection resampling. Furthermore, we devise a coalesced version of the Uphill resampling, designated Uphill-CA, which does not undergo non-coalesced global memory access patterns. With Uphill-CA, we achieve faster results with quality similar to the original Uphill. Thus, the Uphill resampling provides the users of particle filters with a spectrum of fast alternatives on the GPU that is comparable, in terms of quality, with other methods.

Keywords: Graphics Processing Unit, Parallel Resampling, Particle Filter, Resampling Methods

ÖZ

PARÇACIK SÜZGEÇLERİ İÇİN GRAFİK İŞLEME BİRİMİNDE PARALEL YENİDEN ÖRNEKLEME YÖNTEMLERİ

Dülger, Özcan

Doktora, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Prof. Dr. Halit Oğuztüzün

Ortak Tez Yöneticisi : Prof. Dr. Mübeccel Demirekler

Aralık 2017 , 140 sayfa

Bu tezde, parçacık süzgecinin yeniden örnekleme adımının grafik işleme biriminde gerçeklenmesi ile ilgili konular ele alınmıştır. Sistematik, Katmanlı ve Çok-terimli yeniden örnekleme yöntemleri literatürde çok tanınmış bazı yeniden örnekleme yöntemleridir. Döngü çevrimleri arasındaki bağımlılık bu yeniden örnekleme yöntemlerinin paralel gerçeklenmesinde engel çıkarmaktadır. Bu engeller ortadan kaldırılmış olsa da tek kesinliğe sahip kayan noktalı sayılar kullanıldığında, bu yöntemler biriken yuvarlama hatalarından kaynaklanan sayısal kararsızlık problemiyle karşılaşmaktadırlar. Yuvarlama hataları ağırlıklar arası göreceli sapmanın yüksek yada parçacık sayısının büyük olduğu durumlarda ağırlıklar üzerindeki birikimli toplam değerlerinden kaynaklanmaktadır. Sayısal kararsızlık probleminden etkilenmeyen yöntemler vardır. Metropolis ve Rejection yeniden örnekleme yöntemleri ağırlıkların tümünü kullan-

mak yerine onları sadece bire bir orantısal hesaplamalarda kullandıklarından sayısal kararsızlık probleminden etkilenmemektedir. Parçacık süzgecinin grafik işleme biriminde gerçeklenmesi için çok uygundurlar. Buna karşın, bu yöntemler dağınık ana bellek erişim desenlerine sahip olduğundan parçacık sayısı arttıkça yavaşlamaktadır. Bu tezin ilk kısmında, Metropolis yeniden örnekleme yöntemi için bu problemi çözen iki yöntem önerdik. Metropolis-C1 ve Metropolis-C2 olarak isimlendirdiğimiz bu iki yöntemi Metropolis yöntemi ile NVIDIA Tesla K40 ekran kartında karşılaştırdık. Her iki yönteminde en hızlı sonuçları elde ettiği ilk senaryoda, Metropolis-C1 hepsinden hızlı olurken, kalite olarak en kötü kaliteyi elde etmiştir. Buna karşın, Metropolis-C2 Metropolis'e göre kalite olarak yakın sonuçlar elde etmiştir. Üç yönteminde birbirine yakın kalitede sonuçlar elde ettiği ikinci senaryoda ise, Metropolis-C1 ve Metropolis-C2 yavaşlasa da Metropolis'ten hala hızlı olmaktadırlar. Tezin ikinci kısmında, Uphill yeniden örnekleme isminde, yuvarlama hatalarından sakındığından dolayı sayısal kararsızlık problemi olmayan yeni bir yeniden örnekleme yöntemi önerdik. Sistematik, Metropolis ve Rejection yöntemleri ile kalite ve hız açısından karşılaştırma yaptık. Metropolis ve Rejection ile yakın sonuçlar elde ettik. Ek olarak, dağınık ana bellek erişim desenlerine sahip olmayan Uphill yönteminin birleşik versiyonu olan Uphill-CA yöntemini önerdik. Uphill-CA ile, Uphill yöntemi ile yakın kalitede daha hızlı sonuçlar elde ettik. Böylelikle, Uphill yeniden örnekleme parçacık süzgeci kullanıcılarına grafik işleme biriminde kalite olarak diğer yöntemlerle karşılaştırılabilinen hızlı bir alternatif sunmaktadır.

Anahtar Kelimeler: Grafik İşleme Birimi, Paralel Yeniden Örnekleme, Parçacık Süzgeci, Yeniden Örnekleme Yöntemleri

To my family

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Prof. Dr. Halit Oğuztüzün for his great support and guidance. He also motivated and encouraged me every-time during this period. I am also grateful to my co-supervisor Prof. Dr. Mübeccel Demirekler for her valuable support and guidance. It was great honor to work with them during my PhD. I would like to thank to my thesis advisory committee members Assoc. Dr. Murat Manguoğlu and Assoc. Dr. Özcan Öztürk for their comments and suggestions on my thesis. I would also like to thank to Prof. Füsun Özgüner who supervised me during my visit to The Ohio State University. Finally, I would like to thank to the chairman of our department Prof. Dr. Adnan Yazıcı for providing suitable environment to study efficiently in the department.

I would like to thank to Serdar Çiftçi, Alperen Eroğlu, Muhammed Çağrı Kaya, Mehmet Akif Akkuş, Hilal Arslan, Murat Koçak, Merve Asiler, Hüsnü Yıldız, Alper Karamanlıoğlu, Anıl Çetinkaya, Alperen Dalkıran, Tuğberk İşyapar, Cengizhan Türk, Murat Demir, Gürler Gülçe and all other valuable friends for their sincere friendship and support. I also want to thank to Faisal Alanazi, Sarah Al-Shareeda and Menna El-Shaer for their sincere friendship and hospitality during my visit to The Ohio State University.

I gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU board used in this thesis. This thesis was supported in part by Republic of Turkey Ministry of Development, Turkey under grant BAP-08-11-DPT2002K120510. I also gratefully acknowledge the support of The Scientific and Technological Research Council of Turkey (TUBITAK) under program 2211/C scholarship and under program 2214/A International Research Fellowship.

I would like to thank to my valuable relatives Yasemin Konuk, Nazmi Konuk for their support and belief in me. I also want to thank to Hüseyin Konuk, Mustafa Ekmekçi for their help to me. I would like to thank to Prof. Dr. Halise Devrimci Özgüven for

her valuable support. Finally, I want to offer my special thanks to my mother Güner Dülger and to my father Murat Dülger. They grew me and believed me. They were with me everytime I need.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

B.S.	Bachelor of science
BR	Branch resampling
C1	Metropolis-C1 resampling
C2	Metropolis-C2 resampling
CI	Confidence interval
CPU	Central processing unit
CU	Central unit
CUDA	Compute Unified Device Architecture
CURAND	CUDA Random Number Generation
DART	Distributed particle filter algorithm with resampling tree
FPGA	Field Programmable Gate Array
FRIM	Finite-redraw importance-maximizing
GLSL	OpenGL Shading Language
GNN	Global nearest neighbor
GPU	Graphics Processing Unit
HR	Hierarchical resampling
IIDF	Iterated importance density function
IPF	Iterated particle filter
IR	Intermediate resampling
JPDA	Joint probabilistic data association
KB	Kilobyte
LBPR	Load balanced particle replication
M	Metropolis Resampling

M.S.	Master of science
Metropolis-C1	Metropolis-Coalesced 1
Metropolis-C2	Metropolis-Coalesced 2
MHT	Multi-hypotheses tracker
MSE	Mean squared error
NN	Nearest neighbor
PDA	Probabilistic data association
PDF	Probability density function
PE	Processing Element
PF	Particle Filter
Ph.D.	Doctor of philosophy
PR	Parallel resampling
PRNG	Pseudo-random number generator
RCR	Residual cumulative resampling
RMSE	Root mean squared error
RNA	Resampling with non-proportional allocation
RPA	Resampling with proportional allocation
RPG	Random permutation generator
RR	Root resampling
RTS	Resampling tree scheme
SE	Squared error
SIR	Sampling importance resampling
SIS	Sequential importance sampling
SMC	Sequential Monte Carlo
SMSR	Shared memory systematic resampling
SMX	Streaming multiprocessor
UC1	Uphill-C1

UCA	Uphill-CA
Uphill-C1	Uphill-Coalesced 1
Uphill-CA	Uphill-Coalesced Access
UR	Unitary resampling

CHAPTER 1

INTRODUCTION

In this chapter, we put forward the purpose, scope and context of the thesis.

1.1 Motivation

Particle filter is a serial Monte-Carlo estimation method that is particularly suitable when the system or measurement model is highly non-linear and uncertainties are large. The main idea is to approximate the required posterior density function with random samples (particles) that have associated weights. As time progresses, the normalized weight of one particle becomes nearly one and the normalized weights of the remaining particles become nearly zero. This is called degeneracy problem. One of the effective solutions of the degeneracy problem is resampling. In resampling, the particles with higher weights are replicated and those with lower weights are eliminated [32].

Due to have many particles, the particle filter has high computational cost and this makes the adoption of the particle filter for real-time applications prohibitive. Due to the many cores in its architecture, graphics processing unit (GPU) offers promise for fast parallel implementation of particle filters. We have an experiment in Section 6.10 about the speed up of the GPU implementation of particle filter over the CPU implementation of it. The execution times of resampling stage has a big impact on the speed up. There are many resampling methods in the literature. Some of the well-known methods are the Multinomial, Stratified and Systematic resampling. Their computational costs are linear in the number of particles. There are bottlenecks

in the cumulative summation of the weights and inside the while loop in the parallel implementation of the Stratified and Systematic resampling methods. The cumulative summation entails interactions between weights. The iterations of the while loop are dependent on the previous iterations [32, 22]. Murray and co-workers found solutions to remove bottlenecks for the Stratified and Systematic resampling methods [22]. Gong and co-workers also found solutions for the same problem [10]. However, these methods suffer from the numerical instability problem for large number of particles or large weight variance when single-precision units are used. Whenever an algorithm ends up adding a large number with a very small number, the small number may be lost in the machine precision of the operation [22]. In the context of this thesis, an algorithm is numerically stable in the sense that the sequence of arithmetic operations embodying the algorithm does not lead to a detrimental buildup of rounding error [30]. In the Stratified and Systematic resampling algorithms, there is an accumulation of rounding errors when using single-precision units. Furthermore, collective operations on the weights cause difficulty in parallelization of these resampling methods. Murray and co-workers propose two resampling algorithms, namely, Metropolis and Rejection resampling. These methods do not suffer from the numerical instability problems as they do not need cumulative sum of the weights of the particles. They are fast in theory, but in the parallel implementation of the Metropolis and Rejection resampling on the GPU, non-coalesced global memory access patterns occur [22]. In the GPU, the access to the global memory is performed segment by segment. Due to the randomized access of Metropolis and Rejection over the weight set, the access operations become serial. This serialization causes the speed of Metropolis and Rejection deteriorate rapidly. Hence, they can not achieve their target of being fast resampling method. We focus to overcome this problem in the implementation of Metropolis on the GPU. We introduce two techniques that eliminate the non-coalesced global memory access problem of the Metropolis resampling without sacrificing quality much. Furthermore, we devise a new resampling method which does not suffer from the numerical instability problem as it does not require cumulative summation of the weights. It is simple to implement on the GPU and completely parallelizable. The theoretical analyses and proofs are also clear and revealing.

1.2 Contributions of the Thesis

In this thesis, we first introduce two techniques, designated Metropolis-C1 (abbreviated C1) and Metropolis-C2 (abbreviated C2), to ameliorate the non-coalesced global memory access problem of the Metropolis resampling (abbreviated M). We define s-segment concept to achieve coalesced access of the particles. We force the adjacent particles to access the contiguous region of the global memory of the GPU by mapping the regions to the physical segments of the GPU. Thus, the particles can read their weights from the segments of global memory in a coalesced way. We compare these two techniques with the original Metropolis resampling in terms of quality and speed on the GPU. Metropolis-C1 achieves the fastest results (up to 9.6x speed up over Metropolis) but worst in quality. Metropolis-C2 achieves faster results (up to 3.1x speed up over Metropolis) than Metropolis along with similar quality. We argue that these two techniques enable users to adopt the Metropolis resampling, a numerically stable method, for their applications that demand speed, by reducing quality in a controlled manner.

Furthermore, we propose a new resampling algorithm, called Uphill resampling, which is suitable for the GPU. It only compares the weights of two particles and selects the greater one as the candidate particle to replicate. It does this comparison B times where B is the parameter in the Uphill resampling. Since it only compares the weights of two particles, it does not suffer from the numerical instability problem that plagues other methods and does not require cumulative sum among the weights. However, it suffers from the non-coalesced global memory access problem like Metropolis. We propose a coalesced version of the Uphill resampling, called Uphill-CA, that confines the non-coalesced global memory access problem on the GPU. Uphill-CA achieves faster results (up to 7.2x speed up over Uphill) than Uphill. It is same with Uphill in theory, but in practice it is closer to Uphill in quality with some variance. We also introduce generic version of the Uphill resampling called Uphill-C1. It behaves like a local approach. The behavior of Uphill-C1 varies depending on the size of s-segment. Uphill is a special case of Uphill-C1 when s-segment covers the whole weight set. Uphill-C1 achieves the fastest results (up to 10.4x speed up over Uphill) with worse quality compared to Uphill. The tracking performance of the proposed

methods are very close to the performance of existing resampling methods. Hence, we make contributions to the tracking applications by proposing fast and numerically stable methods on the GPU.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows. In Chapter 2, we describe the basis of the particle filter. We mention some problems of particle filter and their solutions. We give the stages of the most well-known sampling importance resampling particle filter. And we give some background about CUDA and GPU architecture. Furthermore, we give some basic concepts of target tracking. And we conclude the chapter by describing our experimental environment, statistical measures and distributions we use along with a non-linear equation in the literature which we use to measure the performance of the resampling methods. In Chapter 3, we give the pseudo-code of the well-known resampling methods in the literature such that the Systematic, Metropolis and Rejection resampling. In Chapter 4, we discuss some of the studies about parallel implementation of resampling on the GPU and on other architectures. In Chapter 5, we describe our proposed methods Metropolis-C1 and Metropolis-C2 in detail. We show how C1 and C2 solve the non-coalesced global memory access problem of the Metropolis resampling. We compare them with Metropolis on the same benchmark and discuss the experimental results. In Chapter 6, we describe our new resampling method Uphill resampling in detail along with its coalesced version Uphill-CA and its generic version Uphill-C1. And we discuss the experimental results of them. In Chapter 7, we compare above-mentioned resampling methods on an aircraft tracking problem with two different scenarios. We discuss the experimental results and show how we benefit from the proposed resampling methods. Finally, Chapter 8 concludes the thesis pointing out some future research directions. There is also appendix A to show the details of the theoretical analysis of Uphill, Uphill-CA and Uphill-C1. Furthermore, the source codes and data sets can be downloaded from `https://user.ceng.metu.edu.tr/~odulger/`.

CHAPTER 2

BACKGROUND

In this chapter, we give the theoretical basis of the particle filters in detail. We mention some problems of particle filter and their solutions. We show how sampling importance resampling (SIR) particle filter is derived from sequential importance sampling algorithm. And we give some basic concepts of CUDA and GPU architecture to inform the readers before we describe our proposed methods. Furthermore, we give some basic concepts of target tracking. Finally, we describe our experimental environment, statistical measures and distributions we use along with a simple end-to-end application which we use to measure the estimation performance of the resampling methods.

2.1 Particle Filter

Particle filter is a sequential Monte Carlo estimation method that represents probability densities with particles. The main idea is to approximate the required posterior density function with random samples (particles) that have associated weights. Estimated state is calculated by using these random samples and their weights. As the number of particles increases, this method will yield an equivalent representation of posterior probability density function (pdf). And it will also become an optimal solution for the Bayesian estimation [32]. The theoretical aspects of the particle filters and the design of efficient particle filters are discussed briefly in the following sections.

2.1.1 Monte Carlo Integration

The basis of the sequential Monte Carlo methods is the Monte Carlo integration. Suppose that we have the following multi-dimensional integration [32]:

$$I = \int g(x)dx \tag{2.1}$$

where $x \in \mathbb{R}^{n_x}$. Monte Carlo methods factorize $g(x) = f(x)\pi(x)$ where $\pi(x)$ is a probability density that satisfies $\pi(x) \geq 0$ and $\int \pi(x)dx = 1$. The N distributed samples $x^i; i = 1, \ldots, N$ can be drawn from $\pi(x)$. The Monte Carlo estimation of integral is [32]:

$$I = \int f(x)\pi(x)dx \tag{2.2}$$

The sample mean is [32]:

$$I_N = \frac{1}{N}\sum_{i=1}^{N} f(x^i) \tag{2.3}$$

If the samples x^i are independent, I_N will be unbiased and converge to the I according to the law of large numbers. The variance of $f(x)$ is [32]:

$$\sigma^2 = \int (f(x) - I)^2 \pi(x)dx \tag{2.4}$$

If the variance is finite, then the estimation error converges to following distribution according to the central limit theorem [32]:

$$\lim_{N\to\infty} \sqrt{N}(I_N - I) \sim N(0, \sigma^2) \tag{2.5}$$

The error of the estimation, $e = I_N - I$, is of order $O(N^{-\frac{1}{2}})$, which means the convergence rate of the estimation is independent of the dimension of n_x [32].

The $\pi(x)$ is the posterior density in the Bayesian estimation. However, it is not usually possible to sample from this posterior density, because it is multivariate, non-standard and cannot be observed in some systems when it is needed. One of the possible solutions is the importance sampling method [32].

2.1.1.1 Importance Sampling

Generating samples from $\pi(x)$ is the ideal sampling operation. But it is not usually possible to obtain $\pi(x)$ before the sampling. We can use a density $q(x)$, which is similar to $\pi(x)$, to generate samples. With a correct weighting of the samples, it still makes the Monte Carlo estimation similar to estimation with the $\pi(x)$. The pdf $q(x)$ is called as the importance or proposal density. The similarity between the $\pi(x)$ and $q(x)$ is given by the following condition [32]:

$$\pi(x) > 0 \Rightarrow q(x) > 0 \text{ for all } x \in \mathbb{R}^{n_x} \tag{2.6}$$

If the condition is valid, the estimation of the integral can be written as [32]:

$$I = \int f(x)\pi(x)dx = \int f(x)\frac{\pi(x)}{q(x)}q(x)dx \tag{2.7}$$

The estimation of I is computed by the N independent samples $x^i; i = 1, \ldots, N$ which are distributed based on $q(x)$ and the weighted sum is formed as below [32]:

$$I_N = \frac{1}{N}\sum_{i=1}^{N} f(x^i)\tilde{w}(x^i) \tag{2.8}$$

where $\tilde{w}(x^i)$ is the importance weight of the ith sample and is defined as below[32]:

$$\tilde{w}(x^i) = \frac{\pi(x^i)}{q(x^i)} \tag{2.9}$$

If the normalization factor of $\pi(x)$ is unknown, we need to make normalizations.

Then the weighted sum becomes [32]:

$$I_N = \frac{\frac{1}{N}\sum_{i=1}^{N} f(x^i)\tilde{w}(x^i)}{\frac{1}{N}\sum_{j=1}^{N} \tilde{w}(x^j)} = \sum_{i=1}^{N} f(x^i)w(x^i) \tag{2.10}$$

where $w(x^i)$ is the normalized importance weight of the ith sample and is calculated as below [32]:

$$w(x^i) = \frac{\tilde{w}(x^i)}{\sum_{j=1}^{N} \tilde{w}(x^j)} \tag{2.11}$$

2.1.2 Sequential Importance Sampling

Sequential importance sampling (SIS) algorithm is a basis for the particle filters. The main idea is to approximate the required posterior density function with random samples (particles) that have associated weights. Estimated state is calculated by using these random samples and their weights. As the number of particles increases, this method will yield an equivalent representation of the posterior pdf [32].

Suppose the joint posterior density at time k is given as $p(X_k|Z_k)$ where X_k is all the targets states up to time k and Z_k is all the measurements up to time k. Its marginal is denoted as $p(x_k|Z_k)$. This joint posterior density can be approximated at time k as follows [32]:

$$p(X_k|Z_k) \approx \sum_{i=1}^{N} w_k^i \delta(X_k - X_k^i) \tag{2.12}$$

where $X_k^i, i = 1, \ldots, N$ is the set of particles with associated normalized weights $w_k^i, i = 1, \ldots, N$. If these samples are drawn from an importance density $q(X_k|Z_k)$, then the normalized weight of the ith particle is calculated according to (2.9)[32]:

$$w_k^i \propto \frac{p(X_k^i|Z_k)}{q(X_k^i|Z_k)} \tag{2.13}$$

Suppose the joint posterior density at time $k-1$ is $p(X_{k-1}|Z_{k-1})$ and we try to

approximate the joint posterior density at time k, with the received measurement z_k and the new set of samples $X_k^i, i = 1, \ldots, N$. If we factorize the importance density $q(X_k|Z_k)$ such that [32]:

$$q(X_k|Z_k) \triangleq q(x_k|X_{k-1}, Z_k)q(X_{k-1}|Z_{k-1}) \tag{2.14}$$

we can draw samples $X_k^i \sim q(X_k|Z_k)$ by augmenting X_{k-1}^i with x_k^i where $X_{k-1}^i \sim q(X_{k-1}|Z_{k-1})$ and $x_k^i \sim q(x_k|X_{k-1}, Z_k)$. To obtain the equation for the normalized weight update, the pdf $p(X_k|Z_k)$ is first expressed as [32]:

$$p(X_k|Z_k) \propto p(z_k|x_k)p(x_k|x_{k-1})p(X_{k-1}|Z_{k-1}) \tag{2.15}$$

and then, after substitution (2.14) and (2.15) into (2.13), the normalized weight update equation is expressed such that [32]:

$$w_k^i \propto \frac{p(z_k|x_k^i)p(x_k^i|x_{k-1}^i)p(X_{k-1}^i|Z_{k-1})}{q(x_k^i|X_{k-1}^i, Z_k)q(X_{k-1}^i|Z_{k-1})} \tag{2.16}$$

$$w_k^i = w_{k-1}^i \frac{p(z_k|x_k^i)p(x_k^i|x_{k-1}^i)}{q(x_k^i|X_{k-1}^i, Z_k)} \tag{2.17}$$

If $q(x_k|X_{k-1}, Z_k) = q(x_k|x_{k-1}, z_k)$, then the importance density will be dependent only to x_{k-1} and z_k. Thus, we do not need to store all the particles from time 0 to time $k-1$ and all the measurements from time 0 to time $k-1$. This is useful when we need only the filtered estimate of the posterior density $p(x_k|Z_k)$ at each time step. Then the normalized weight update equation becomes [32]:

$$w_k^i \propto w_{k-1}^i \frac{p(z_k|x_k^i)p(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{k-1}^i, z_k)} \tag{2.18}$$

and the posterior filtered density $p(x_k|Z_k)$ becomes [32]:

$$p(x_k|Z_k) \approx \sum_{i=1}^{N} w_k^i \delta(x_k - x_k^i) \tag{2.19}$$

As $N \rightarrow \infty$ this approximation gets close to the true posterior density $p(x_k|Z_k)$. In the SIS algorithm the normalized importance weights $\{w_k^i\}_{i=1}^N$ and the samples $\{x_k^i\}_{i=1}^N$ are propagated recursively as each measurement is received sequentially. The pseudo-code of SIS algorithm is shown in Algorithm 2.1. This algorithm is considered as the basis of particle filters. The choice of importance density function is important for the performance of particle filters [32]. This topic will be discussed in Section 2.1.4.

Algorithm 2.1 SIS particle filter[32]

procedure $[\{x_k^i, w_k^i\}_{i=1}^N]$ = SIS$(\{x_{k-1}^i, w_{k-1}^i\}_{i=1}^N, z_k)$

1: for $i = 1 : N$

$x_k^i \sim q(x_k|x_{k-1}^i, z_k)$: prediction

$\tilde{w}_k^i = w_{k-1}^i \frac{p(z_k|x_k^i)p(x_k^i|x_{k-1}^i)}{q(x_k^i|x_{k-1}^i, z_k)}$: weight update

end for

2: $s\_\tilde{w} = SUM[\{\tilde{w}_k^i\}_{i=1}^N]$

3: for $i = 1 : N$

$w_k^i = s\_\tilde{w}^{-1} * \tilde{w}_k^i$

end for

2.1.2.1 Degeneracy Problem

After many iterations, the increase in the relative variance of importance weights leads to a common problem known as the degeneracy problem in SIS particle filter. This means after a number of steps, the normalized weight of one particle becomes close to one and the normalized weights of others become close to zero. And the update of the particles whose normalized weights are close to zero leads to wasted computational effort. One of the measures of degeneracy is the effective sample size N_{eff} which is calculated as [32]:

$$N_{eff} = \frac{1}{\sum_{i=1}^N (w_k^i)^2} \tag{2.20}$$

where $1 \leq N_{eff} \leq N$. When the normalized weight of each particle is $1/N$ then $N_{eff} = N$. When only one particle has a normalized weight with value of one and

the others' normalized weights are zero, then $N_{eff} = 1$. This shows that when N_{eff} gets smaller, the degeneracy becomes more pronounced. The most common solution for the degeneracy problem is resampling [32].

2.1.3 Resampling

In the SIS filter, when there is a significant degeneracy, resampling is required. This step eliminates the particles that have small weights and replicates the particles that have large weights. At the end of resampling, the normalized weights of all particles become equal to $1/N$. There are resampling algorithms that run in $O(N)$ time complexity such as Systematic resampling, Stratified resampling and Residual resampling [32].

We have mentioned the importance density function, weight update and the resampling algorithm. A generic prototype for particle filter (PF) which consists of these components is shown in Algorithm 2.2. In this algorithm, N_{thr} is a threshold for the number of effective particles. If the effective sample size is smaller than this threshold, resampling operation is done [32].

Algorithm 2.2 A generic prototype for particle filters[32]

procedure $[\{x_k^i, w_k^i\}_{i=1}^N]$ = PF$(\{x_{k-1}^i, w_{k-1}^i\}_{i=1}^N, z_k)$

1: $[\{x_k^i, w_k^i\}_{i=1}^N]$ = SIS$(\{x_{k-1}^i, w_{k-1}^i\}_{i=1}^N, z_k)$

2: $N_{eff} = \frac{1}{\sum_{i=1}^N (w_k^i)^2}$

3: if $N_{eff} < N_{thr}$

$[\{x_k^i, w_k^i\}_{i=1}^N]$ = RESAMPLE $[\{x_k^i, w_k^i\}_{i=1}^N]$

end if

Although resampling operation solves the degeneracy problem, it causes other practical problems. One of the problems is limiting the opportunity of parallel implementation because all the particles must be combined. Another problem is sample impoverishment. This happens because the particles with large weights are selected many times, leading to a loss of diversity among the particles [32].

2.1.4 Selection of Importance Density

The importance density $q(x_k|x_{k-1}^i, z_k)$ is a critical part of designing the particle filter. The optimal choice of the importance density is [32]:

$$q(x_k|x_{k-1}^i, z_k) = p(x_k|x_{k-1}^i, z_k) \tag{2.21}$$

$$q(x_k|x_{k-1}^i, z_k) = \frac{p(z_k|x_k, x_{k-1}^i)p(x_k|x_{k-1}^i)}{p(z_k|x_{k-1}^i)} \tag{2.22}$$

And by substituting (2.22) into (2.18), the normalized weight update of the ith particle becomes [32]:

$$w_k^i \propto w_{k-1}^i p(z_k|x_{k-1}^i) \tag{2.23}$$

This equation shows that the weights of the particles at time k can be calculated at time $k-1$. In general, choosing the optimal importance density and evaluating the likelihood $p(z_k|x_{k-1}^i)$ are not possible. But there are some special cases where these operations are possible. The first case is that the model has a finite set of member x_k. The second case is that the model has importance density $p(x_k|x_{k-1}^i, z_k)$ that is Gaussian. For other cases that differ from these special cases use suboptimal importance density function [32].

The well-known sub-optimal density function is the transitional prior which is defined as [32]:

$$q(x_k|x_{k-1}^i, z_k) = p(x_k|x_{k-1}^i) \tag{2.24}$$

If the process noise of the system is Gaussian with zero mean and covariance Q, then the transitional prior becomes [32]:

$$p(x_k|x_{k-1}^i) = N(x_k; f_{k-1}(x_{k-1}^i), Q_{k-1}) \tag{2.25}$$

And by substituting (2.24) into (2.18), the normalized weight update of the ith particle becomes as [32]:

$$w_k^i \propto w_{k-1}^i p(z_k|x_k^i) \tag{2.26}$$

2.1.5 Sampling Importance Resampling (SIR) Particle Filter

SIR algorithm is derived from SIS filter by choosing the importance density function as transitional prior $p(x_k|x_{k-1}^i)$ and by using resample algorithm at each time step. The assumptions for using the SIR particle filter are: (1) the dynamics of the state and the measurement function are known; (2) to sample from the prior with the distribution of process noise is possible. Furthermore, the likelihood function $p(z_k|x_k^i)$ should be available. Since the resampling operation is done at each step, the normalized weight of each particle is equal to $\frac{1}{N}$ at the end of the step. Therefore, we do not need to pass the weights of the particles to the next time step. The normalized weight update operation of the ith particle becomes as follows [32]:

$$w_k^i \propto p(z_k|x_k^i) \tag{2.27}$$

The stages of the SIR particle filter algorithm are shown below [32]:

Algorithm 2.3 SIR Particle Filter ([32])

procedure $[\{x_k^i\}_{i=1}^N]$ = SIR$(\{x_{k-1}^i\}_{i=1}^N, z_k)$

1: for $i = 1 : N$

$x_k^i \sim p(x_k|x_{k-1}^i)$: prediction

$\tilde{w}_k^i = p(z_k|x_k^i)$: weight update

end for

2: $s\_\tilde{w} = SUM[\{\tilde{w}_k^i\}_{i=1}^N]$

3: for $i = 1 : N$

$w_k^i = s\_\tilde{w}^{-1} * \tilde{w}_k^i$

end for

4: $[\{x_k^i\}_{i=1}^N]$ = RESAMPLE $[\{x_k^i, w_k^i\}_{i=1}^N]$

In this pseudo-code, N is the number of particles; i is the index of the particle; k is the index of the time step; x_k^i represents the state of the ith particle; $\tilde{w}_k^i$ is the weight of the ith particle; w_k^i is the normalized weight of the ith particle; $s\_\tilde{w}$ is the sum of the weights of all particles; x_k represents the true state of the track; z_k is the measurement; $p(x_k|x_{k-1}^i)$ is the transitional prior and $p(z_k|x_k^i)$ is the likelihood function. We omit the time step variable k on the following equations and algorithms for simplicity.

In the first stage, predicted states of the particles are calculated and the weights of the particles are updated based on their predicted states and the measurement data. In stage 2, the sum of the weights which is to be used in stage 3 are calculated. In stage 3, weights are normalized, and, in stage 4, resampling is performed.

2.2 CUDA Programming Concepts and GPU Hardware

In this section, we provide an overview of some basic concepts of CUDA programming. In CUDA, an application consists of one or more kernels. Kernel is a function that runs on the GPU. A thread is an execution unit on the GPU with its own program counter, register and unique id in its block. A block consists of concurrently running threads. The threads in a block can cooperate with each other through barrier synchronization and shared memory. The threads across the blocks cannot cooperate with each other. A grid consists of thread blocks which execute the same kernel. Programmer's view of CUDA is given in Figure 2.1 [18].

There are mainly three types of memory region on the GPU. These are registers, shared memory and global memory. Registers and shared memory are on-chip memory and are the fastest memory regions on the hardware. Global memory is off-chip and slowest memory region on the hardware. Registers are private for each thread and can only be accessible by their own thread. Shared memory is private for each block and is accessible by all threads in a block. The threads in different blocks cannot access to the shared memory of the other blocks. The global memory is not private and can be accessible by all the blocks in the grid [8].

A GPU hardware consists of many streaming multiprocessors. An illustration of

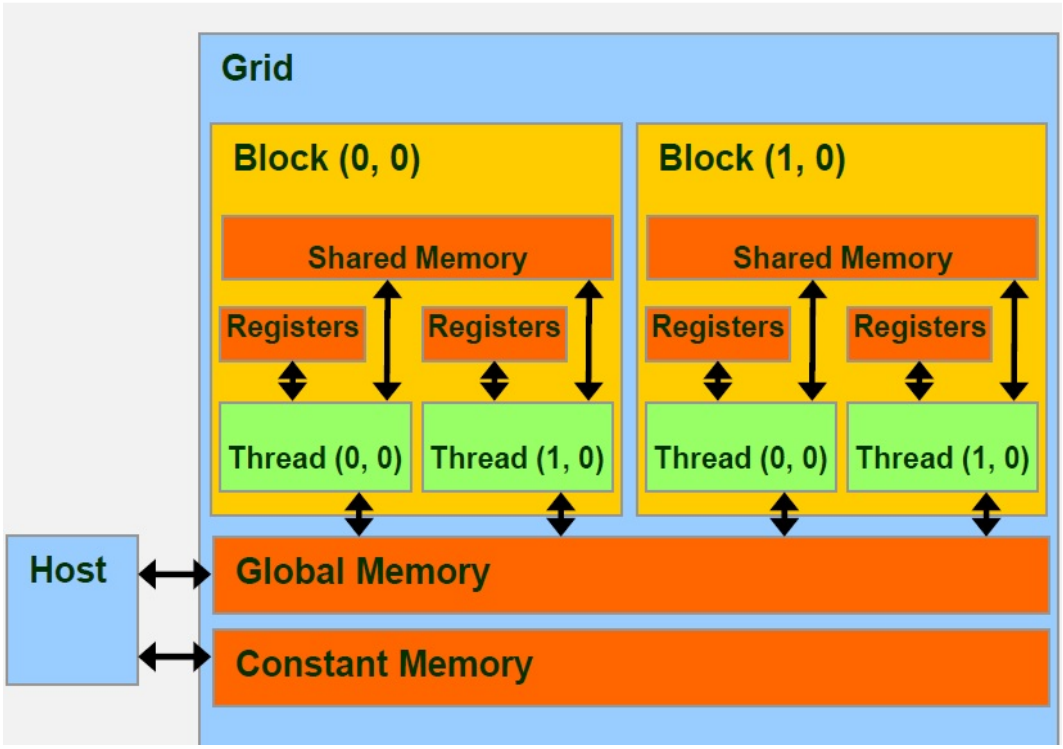

Figure 2.1: Programmer's view of CUDA [18].

streaming multiprocessor (SMX) in Kepler architecture is given in Figure 2.2 [24].

The blocks are scheduled to the different streaming multiprocessors. The threads execute on CUDA cores in a warp by warp basis. Warp is an execution unit which consists of 32 threads. The threads in a warp are physically related to each other. This means all the threads in a warp execute the same instruction at a time [8].

2.3 Target Tracking

Target tracking can be viewed as an application area at the intersection of control theory and signal processing. A target is any object that we want to track by using dynamic state estimation methods. Target tracking uses filter to perform recursive state estimation. The filter consists of kinematic equation of the target, measurement equation and the measurements obtained from the sensors. The most common tracking filters are alpha-beta filter, Kalman filter, extended Kalman filter and particle filter. The measurement consists of kinematic values such as position, velocity, accel-

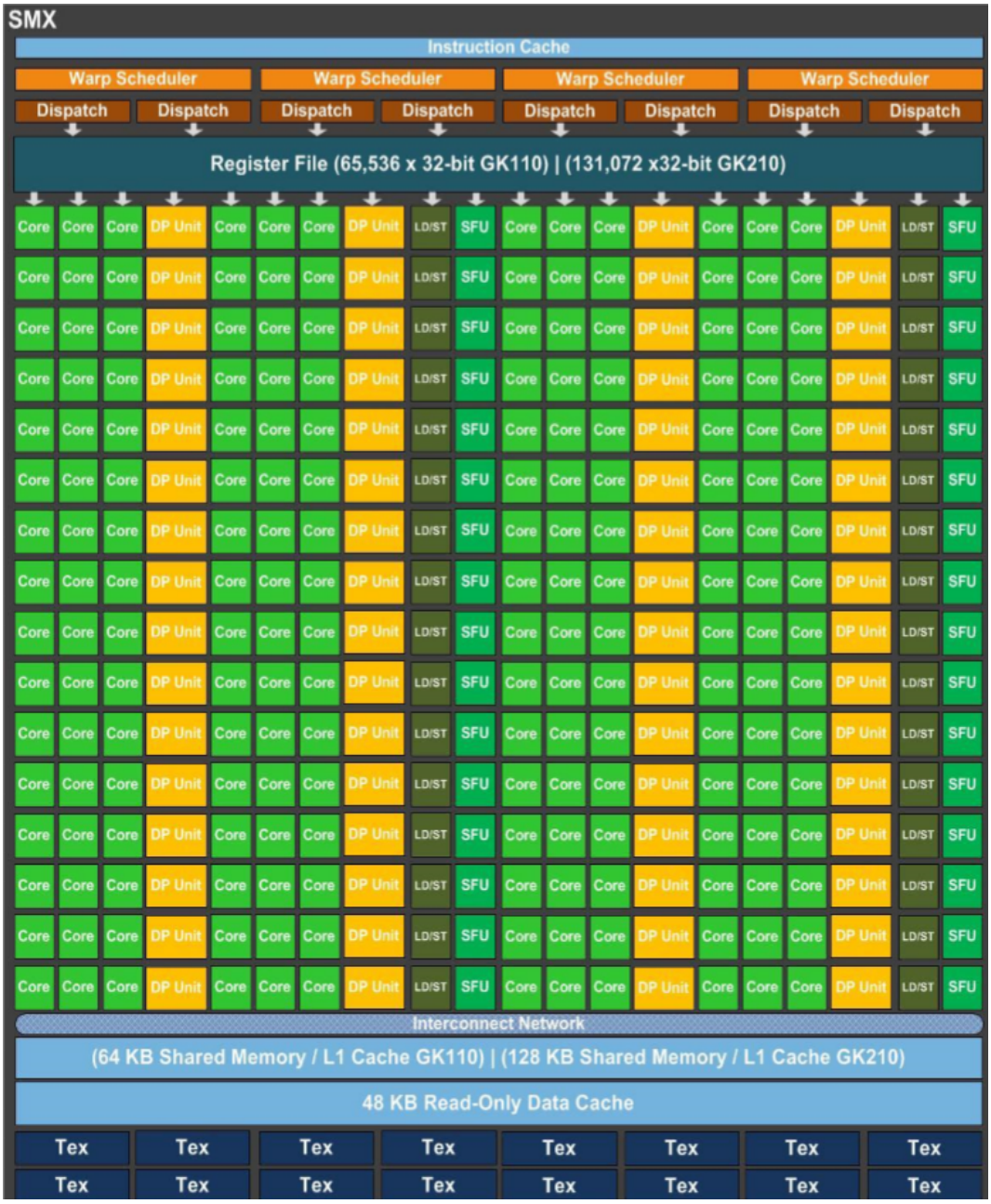

Figure 2.2: A streaming multiprocessor (SMX) in Kepler architecture [24].

eration, range, and bearing or attributes such as signal strength, intensity, aspect ratio. A measurement can belong to an existing target or it can belong to a new target or it can be false alarm. A track is sequence of measurements decided by the tracker to come from a single source. Instead of holding all data, it is enough to keep sufficient statistics. These are mean and covariance in Kalman filter, and particles and their weights in particle filter. A track can be in three different life stages [32]:

- Tentative: Track is in the initiation process. We are not sure whether it is a target or not.
- Confirmed: Track is a valid target inside the area.
- Deleted: Track is false alarm.

To identify each measurement and determine which measurements belong to which target, gating and association processes are performed. The measurements inside the gate of the target are candidate measurements to be assigned as a valid measurement for the target. If there are more than one measurement inside the gate or if a measurement is inside the gates of more than one target, we need a mechanism to associate the measurements. Nearest neighbor (NN), probabilistic data association (PDA) for single target tracking applications and global nearest neighbor (GNN), joint probabilistic data association (JPDA), multi-hypotheses tracker (MHT) for multi target tracking applications are the common algorithms for measurement association. M/N logic is one of the most common algorithm for switching the life stages of the track. For deletion of tentative tracks, if a track is not updated for the first M consecutive scans or is not updated at least M of N following scans, it is deleted. Otherwise, it is confirmed. For deletion of confirmed track, if a track is not updated for the first M consecutive scans and is not updated at least M of N following scans, it is deleted. A maneuver is a sudden change in the motion of the target. And the target assumes other model than the filter uses. There are different kinematic models. These are constant velocity model, constant acceleration model and coordinated turn model. The first model is used in non-maneuvering case, the second and the third models are generally used in maneuvering case [32]. In our experiments, we assume there is a single target and all measurements belong to this target.

Table 2.1: GPU configuration employed in experiments.

Property	Value	Property	Value	Property	Value
Architecture	Kepler	Global Memory	11520 MBytes	Warp Size	32
Clock Rate	745 MHz	Shared Memory	49152 bytes	Segment Size	128 bytes
CUDA Core	2880	Register Count	65536	SMX Count	15

2.4 Experimental Environment

All the experiments are performed on NVIDIA Tesla K40 GPU board. Single precision is used in all implementations. We compile the source codes on CUDA 7.5 compiler with the *-arch sm_35* compiler flag to target the architecture of Tesla K40. The specifications of the board are given in Table 2.1 [23, 24, 26].

We set the block size appropriately by trying to distribute the blocks to the SMXs evenly. Block size must be a power of 2 and the maximum we set is 512. Below is the block size for the different number of particles.

$$BlockSize = \begin{cases} 512, & N \in [2^{13}, 2^{22}] \\ 256, & N \in 2^{12} \\ 128, & N \in 2^{11} \\ 64, & N \in 2^{10} \\ 32, & N \in [2^{5}, 2^{9}] \\ 16, & N \in 2^{4} \end{cases} \tag{2.28}$$

We use XORWOW PRNG in the CURAND library to obtain random numbers. In the implementation of the pseudo-random number generator (PRNG), there is a trade-off between speed and bias. We choose the robust, but slower implementation. To overcome this expense, we initialize the PRNGs at the beginning and save their states to the global memory. We load their states from the global memory in a coalesced way when we want to generate a random number inside the kernel. We save the updated states to the global memory before exiting the kernel [25]. In speed measurements we include the load and save times but we exclude the initialization time of the PRNGs.

2.5 Statistical Measures and Distributions

To assess the quality of the resampling algorithms, we compare the sequences before and after resampling [19]. We use the information o^i in the new offspring sequence, which is the number of times the ith particle is replicated. To measure the difference between the two weight sequences we use mean squared error (MSE). The squared error (SE) of a sequence o_l is defined as follows [22]:

$$SE(o_l) = \sum_{i=1}^{N} \left(o_l^i - \frac{N\tilde{w}^i}{s\_\tilde{w}} \right)^2 \tag{2.29}$$

The MSE is the sample mean of the squared errors and is defined as follows:

$$MSE(o) = \frac{1}{K} \sum_{l=1}^{K} SE(o_l) \tag{2.30}$$

where K is the number of samples (or offspring sequences) for a particular weight sequence. MSE can be written as combination of two components, bias and variance [22]:

$$MSE(o) = Var(o) + \|Bias(o)\|^2 \tag{2.31}$$

where

$$Var(o) = \sum_{i=1}^{N} Var(o^i), \|Bias(o)\|^2 = \sum_{i=1}^{N} \left(\hat{o}^i - \frac{N\tilde{w}^i}{s\_\tilde{w}} \right)^2 \tag{2.32}$$

where $\hat{o}^i$ is the sample mean of the ith component of o across the K offspring sequences. $Var(o^i)$ is the sample variance of the ith component of o across the K offspring sequences. To assess the bias result of the resampling algorithms, we use the contribution of the squared bias to the mean squared error, $\|Bias(o)\|^2 / MSE(o)$. In the MSE results, MSE is normalized by the number of particles, $MSE(o)/N$ [22].

We obtain the weight sequences for our trials from gamma distribution. We choose gamma distribution because it is in fact a family of distributions with two parame-

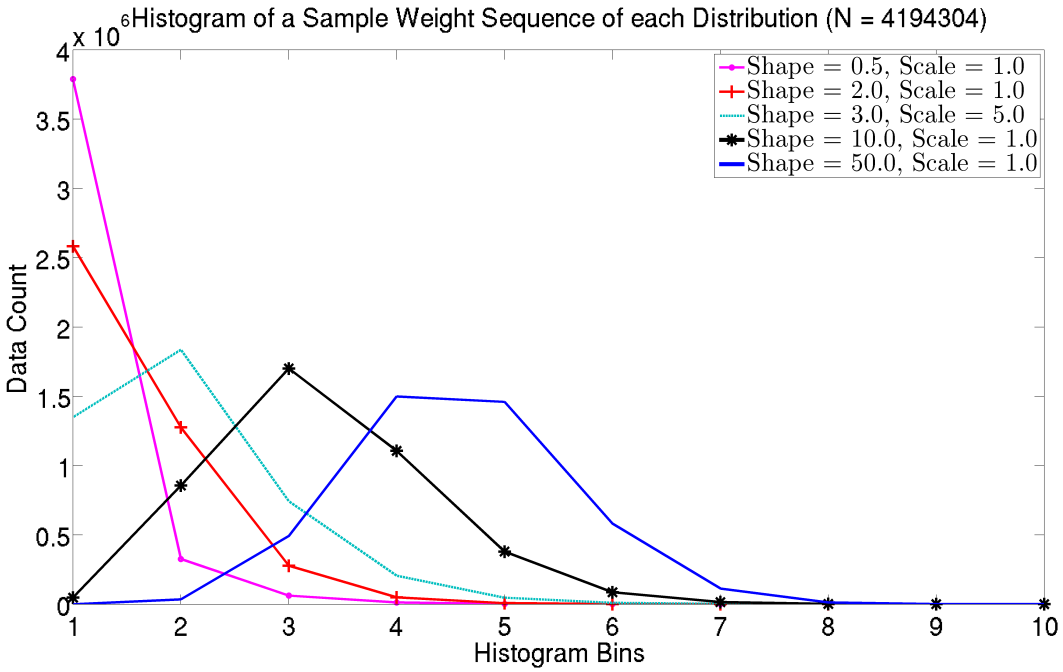

Figure 2.3: An illustration of each gamma distribution with $N = 4,194,304$. X-axis represents the bins of the histograms and y-axis represents the number of data in the bins. Shape and scale are the parameters of the gamma distribution.

ters, namely, shape and scale, which allow us to obtain a variety of distributions to generate random weight sequences [5]. We create five distributions by varying the shape and scale parameters. The histograms of a sample weight sequence of each gamma distribution with $4,194,304$ particles are given in Figure 2.3. In addition, we use other distributions to show the performance of resampling algorithms as the relative variance in the weight sequence increases. These distributions are generated as follows [22]:

$$\tilde{w}^i = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(\tilde{x}^i - y)^2\right) \tag{2.33}$$

where $\tilde{x} \sim N(0,1)$ and $y \sim N(\tilde{x},1)$ (drawn from normal distribution). The relative variance in weights increases as y increases. We create five different distributions by taking the values of y as integers from 0 to 4.

2.6 A Simple End-to-End Application

To show the relative significance of the resampling stage in the SIR particle filter in terms of time cost, we use a well-known highly non-linear example found in the literature [10, 2]. The equations for this system are as follows:

$$x_{k+1} = \frac{x_k}{2} + \frac{25x_k}{1+x_k^2} + 8cos(1.2k) + v_k \tag{2.34}$$

$$z_k = \frac{x_k^2}{20} + n_k \tag{2.35}$$

where v_k and n_k are zero mean Gaussian random variables with variance $\sigma_v^2 = 10$ and $\sigma_n^2 = 1$, respectively. We use the first equation in prediction and the second equation in weight update. The particles are initialized as $N\ (0, 2)$. The pseudo-code of this example is given in Algorithm 2.4.

Algorithm 2.4 Example SIR Particle Filter

procedure $[\{x_k^i\}_{i=1}^N]$ = SIR $(\{x_{k-1}^i\}_{i=1}^N, z_k)$

1: for $i = 1 : N$

$x_k^i \sim p(x_k|x_{k-1}^i)$: prediction

$\tilde{w}_k^i = p(z_k|x_k^i)$: weight update

end for

2: $s\_\tilde{w} = SUM[\{\tilde{w}_k^i\}_{i=1}^N]$

3: for $i = 1 : N$

$w_k^i = s\_\tilde{w}^{-1} * \tilde{w}_k^i$

end for

$\hat{x_k} = SUM[\{x_k^i * w_k^i\}_{i=1}^N]$

4: $[\{x_k^i\}_{i=1}^N]$ = RESAMPLE $[\{x_k^i, \tilde{w}_k^i \ or \ w_k^i\}_{i=1}^N]$

Algorithm 2.4 is a slight adaptation of Algorithm 2.3 for this application. Specifically, in stage 3, we calculate the estimation results of the filter after normalization process.

CHAPTER 3

RESAMPLING METHODS

In this chapter, we present the pseudo-code of the resampling methods we use in the experiments. And we mention their strength and weakness when implemented on the GPU.

3.1 Resampling

In resampling, the aim is to replicate the particles according to their weights. The expected number of replication of a particle i after resampling is related to its current weight as shown below [22]:

$$ER(i) = \frac{N\tilde{w}^i}{s\_\tilde{w}} \tag{3.1}$$

Satisfying (3.1) for each particle ensures unbiased resampling results as well as unbiased estimation of the marginal likelihood of the measurement data. In practice, however, it is not possible to obtain unbiased resampling results due to imperfect random number selections and numerical issues [22].

3.2 Systematic Resampling

The cumulative summation of the weights in the Multinomial, Stratified and Systematic resampling algorithms involves interactions among the weights. Global communication between the threads in the GPU makes the parallelization of the cumulative

summation of the weights difficult. There are some solutions with logarithmic time complexity in the number of particles. Furthermore, cumulative summation of the weights leads to the numerical instability problem for large number of particles or large weight variance when single precision floating point numbers are used [22].

The output of the resampling for the next time step, for each particle, can either be the number of replications of the particle or the ancestor of the particle. Stratified and Systematic adopt the former approach, whereas Multinomial, Metropolis and Rejection the latter [22]. We select the Systematic resampling in our experiments. It is better than Multinomial and Stratified in quality and speed [22]. The pseudo-code for the Systematic resampling algorithm is given in Algorithm 3.1.

Algorithm 3.1 Systematic Resampling [22]

1: **procedure** $[\{O^i\}_{i=1}^N]$ = SYSTEMATIC($\{\tilde{w}^i\}_{i=1}^N$)
2: $u \sim v[0\ 1)$
3: C = INCLUSIVE-PREFIX-SUM($\tilde{w}$)
4: foreach $i = 1 : N$
5: $\quad r^i = \frac{N * C^i}{C^N}$
6: $\quad O^i = \min(N, \lfloor r^i + u \rfloor)$
7: end foreach

The output of this algorithm is the cumulative sum of the number of replications of the particles which is called as cumulative offspring and denoted O. C is the cumulative sum of the weights; u is a real random number between 0 and 1, excluding 1, drawn from a uniform distribution; r is a real variable.

The 'foreach' loop can be executed in parallel among the threads where a dedicated thread runs for each particle. Each thread calculates the cumulative offspring of the particle it represents. We need to convert O to an ancestor vector in order to assign the states of the ancestor of the particle. An algorithm which is suitable for the GPU implementation is given in Algorithm 3.2.

In this algorithm, the 'foreach' loop can be executed in parallel among the threads where each thread runs for a particular particle. Each thread finds the children whose ancestors are the particle it represents.

Algorithm 3.2 Cumulative Offspring to Ancestors [22]

procedure $[\{x^i_{new}\}_{i=1}^N]$ = CO TO ANCESTORS $(\{x^i, O^i\}_{i=1}^N)$
foreach $i = 1 : N$
    if $i == 1$ then $start = 0$
    else then $start = O^{i-1}$
    end if
    $o^i = O^i - start$
    for $j = 1 : o^i$
        $x^{start+j}_{new} = x^i$
    end for
end foreach

3.3 Metropolis and Rejection Resampling

Murray and co-workers offer two alternative resampling methods, namely, Metropolis and Rejection. These methods do not suffer from the numerical instability problem since they do not need cumulative summation of the weights [22].

The Metropolis resampling algorithm uses only the ratio of the weights rather than any collective operations across the weights. It has the B parameter which represents a trade-off between bias and speed. Smaller B means faster results but with a larger bias. The pseudo-code for the Metropolis resampling is given in Algorithm 3.3.

Algorithm 3.3 Metropolis Resampling [22]

procedure $[\{x^i_{new}\}_{i=1}^N]$ = METROPOLIS$(\{x^i, \tilde{w}^i\}_{i=1}^N, B)$
foreach $i = 1 : N$
    $t = i$
    for $m = 1 : B$
        $u \sim v[0\ 1]$
        $j \sim v\{1, .., N\}$
        if $u \leq \tilde{w}^j / \tilde{w}^t$ then $t = j$ end if
    end for
    $x^i_{new} = x^t$
end foreach

In this pseudo-code, B is the trade-off parameter which serves as the number of iterations of the inner loop; u is a real random number between 0 and 1 drawn from a uniform distribution; j is a random integer between 1 and N drawn from a uniform distribution; t is the index of the particle selected as the ancestor of the ith particle for the next time step; x^i_{new} is the new state of ith particle for the next time step. The other parameters are as in Algorithm 2.3.

The 'foreach' loop can be executed in parallel among the threads where each thread represents a particle whereas the 'for' loop must execute sequentially. Each thread selects an ancestor for the particle it represents. The Metropolis resampling algorithm can run with a single CUDA kernel. The weights are stored on the global memory of the GPU. The read/write operations on the global memory are performed segment by segment. All the threads in the same warp are physically related to each other. This means the warp completes an instruction only when all the threads in the warp complete the same instruction. So it is important to read the weights from the same segment to avoid serialization in memory transaction [8]. But, due to the randomization in the Metropolis resampling, non-coalesced global memory access patterns occur causing the speed of the resampling deteriorates rapidly as the number of particles increases [22].

The Rejection resampling also does not suffer from numerical instability as it does not require collective operations across the weights. Furthermore, unlike the Metropolis resampling, it is unbiased. The pseudo-code for the Rejection algorithm is given in Algorithm 3.4.

In this algorithm, $\tilde{w}_{max}$ is the maximum weight and the other parameters are as in Algorithm 2.3 and Algorithm 3.3. Like the Metropolis resampling, the 'foreach' loop of the Rejection resampling can be executed in parallel among the threads where a dedicated thread runs for each particle. And each thread selects an ancestor for the particle it represents. The Rejection resampling also undergoes non-coalesced global memory access patterns. Furthermore, it suffers from warp divergence since the number of iterations of the 'while' loop is not equal in all threads of the warp.

Algorithm 3.4 Rejection Resampling [22]

procedure $[\{x^i_{new}\}_{i=1}^N]$ = REJECTION($\{x^i, \tilde{w}^i\}_{i=1}^N$)
foreach $i = 1 : N$
    $j = i$
    $u \sim v[0\ 1]$
    while $u > \tilde{w}^j / \tilde{w}_{max}$
        $j \sim v\{1, .., N\}$
        $u \sim v[0\ 1]$
    end while
    $x^i_{new} = x^j$
end foreach

CHAPTER 4

RELATED WORK

In this chapter, we discuss the related studies about resampling in the literature. There are studies about the parallel implementation of the Systematic resampling. But there are some practical challenges in the parallel implementation of the Systematic resampling. The cumulative summation of the weights are less-readily parallelized because of the interactions among the weights of the particles. In some of these studies, the authors focus to the efficient implementation of the cumulative summation of the weights. Another challenge is the dependency inside the loop of the selection of the particles which are propagated for the next time step. In other studies, the authors focus on the eliminating this dependency and achieving fully parallel implementation of the Systematic resampling. There are some other studies which aim to reduce the global operations among the weights in resampling by introducing distributed structures. Some researchers point out the importance of coalesced implementation of resampling on the GPU [1, 17]. Li and co-workers present classification, implementation and strategies of different resampling methods on a variety of architectures. They categorize resampling methods as sequential and parallel algorithms. Sequential implementation consists of three categories: single distribution sampling; compound-sampling; special strategies. Parallel implementation also consists of three categories: mapping to specific hardware platforms; distributed resampling; normalization-free resampling. They present the resampling algorithms according to these categories in detail [20].

4.1 Studies on Systematic Resampling

There are works that aim to propose efficient algorithms for the parallel implementation of the Systematic resampling.

Hendeby and his co-workers focus to the resampling stage of the particle filter. They implement the particle filter in parallel on the GPU. They note that there was no complete solution for the parallel implementation of particle filter on the GPU at the time they did this work. In their paper, they mention general purpose computing on GPU (GPGPU). As a programming language they use OpenGL Shading Language (GLSL). In the GPU based particle filter implementation, they focus to the resampling and weight normalization stages of the particle filter since all the particles interact with each other in these stages. Cumulative summation of the weights, selection and redistribution of particles are a challenge in the parallel implementation of them. They propose multi-pass scheme for the cumulative summation of the weights. In this scheme, an adder tree runs forward to obtain the summation of all weights and then runs backward to calculate cumulative summation of each particle by using the sum obtained in forward pass. The selection of particles for the next time step performs by using cumulative sum of the weights. They use a constant velocity tracking model to test the algorithms. They compare the results of the GPU implementation with the results of the CPU implementation of parallel particle filter. They state that the GPU implementation is faster than the CPU implementation with the same accuracy [12]. Hendeby and his co-workers use the same technique in another work. They claim that the GPU filter in their study is the first particle filter published in the literature that is completely parallelized on the GPU. They use a minimal sensor network with bearing only sensors to compare the performance of the algorithms. They report that they achieve faster results with the GPU implementation compared to the CPU implementation with the same accuracy [13].

Gong and his co-workers focus to the parallel implementation of the Systematic resampling. They propose the shared memory systematic resampling (SMSR) algorithm which is suitable for shared memory architectures. They eliminate the dependency inside the while loop of the Systematic resampling. They define left and right boundaries to write the selected particles between these boundaries to the global

memory. They implement all stages of PF in parallel on the GPU. Random numbers are generated in parallel. Parallel reduction technique is used for computing the sum of the weights. Parallel scan is used for the cumulative summation of the weights. They use a highly non-linear model to measure the performance of their algorithms. They compare serial implementation of PF, GPU implementation of PF except resampling stage and GPU implementation of PF with SMSR. They achieve 30+x speed up over the serial PF [10].

Hwang and Sung propose a solution to the load imbalance problem in the implementation of the Systematic resampling on the GPU. This problem occurs because of the variances of the weights. As the variance increases, imbalanced workload occurs in writing the selected particles to the global memory. They propose a solution called load balanced particle replication (LBPR) algorithm for the Systematic resampling. They modify the particle selection and replication index generation stages of resampling. They generate minimal replication index array to reduce the imbalanced workload among the threads instead of complete replication index. Then they recover the complete ones from minimal replication index array in particle selection index. They achieve almost constant execution time and outperform the conventional systematic resampling in worst case computation time [15].

Wu and his co-workers propose the iterated importance density function (IIDF) to solve the degeneracy problem. They employ this technique to the sampling stage of the particle filter and obtain iterated particle filter (IPF). They also propose parallel resampling (PR) and use it in the resampling stage of the IPF. The main idea of PR is same with the main idea of the Systematic resampling with some differences. In PR, first the particles are sorted according to their weights in descending order. Then the particles are divided into two subsets. The first one consists of the particles whose normalized weight is greater than $1/N$. The second one consists of the remaining particles. The particles in the first subset replicate themselves according to the ratio of their weights to the sum of all weights while some of the particles are eliminated from the second subset. When comparing the differences between PR and Systematic resampling, it is easier to implement PR in parallel; some particles are replicated one less than those in the Systematic resampling because of the floor operation; only the particles with the smallest weight are eliminated. But in the Systematic resam-

pling a better one may be eliminated. They implement IPF in parallel on the GPU by using PR in the resampling stage. They also introduce improved version of the parallel cumulative sum algorithm to calculate the cumulative sum of the weights more efficiently. They measure the performance of PR on an one-dimensional numerical simulation and a target tracking with passive radar application. They state that PR achieves comparable estimation results with the results of the Systematic resampling. Furthermore, they argue that PR is more convenient for parallel implementation of resampling [37].

The authors of the studies cited in this section focus on the efficient implementation of the cumulative summation of the weights in Systematic resampling. They offer parallel scan algorithms whose time complexity is $O(logN)$. They also focus to the bottleneck in the selection of particles. Although they eliminate the bottleneck, there still exists some problems. One of them is the numerical instability problem caused by the cumulative summation of the weights and the randomly generated numbers in the algorithm. Another problem is the load imbalance problem of the threads in a warp when they write the selected particles to the global memory of the GPU. Hwang and Sung propose a solution for the load imbalance problem and achieve almost constant execution time. In this thesis, we point out the numerical instability problem of the Systematic resampling and the efficient calculation of the cumulative summation of the weights. Furthermore, the resampling methods we devise do not suffer from the numerical instability problem as they do not require cumulative summation of the weights. In addition, the workloads among the threads in a warp are very close to each other.

4.2 Studies on Distributed Resampling

There are studies that aim to reduce the global operations over the weights by dividing the particles into subsets and performing local resampling.

Bolic and his co-workers propose an efficient algorithm for the distributed particle filters on FPGA [4]. They propose resampling with proportional allocation (RPA) particle filter and resampling with non-proportional allocation (RNA) particle filter

algorithms. The distributed architecture has four processing elements (PEs) and one central unit (CU) with interconnection network. Particle generation and calculation of weights are fully parallel in PEs, because there is no data dependency in these operations. But the resampling performs partially or fully in CU. In full resampling, CU gets the particles and their weights from each PE and performs sequential resampling. In partial resampling, resampling is distributed to each PE and CU is responsible only small portion of resampling. There are particle routings between PEs to balance particles on each PE. Their target is to reduce the communication in interconnection network and make resampling deterministic. They test the performance of the algorithms on a bearing only tracking application. They state that the performance of centralized resampling and RPA are same with the sequential PF. On the other hand, they achieve speed improvement with RNA and offer it as a good choice when speed is important.

Balasingam and his co-workers also propose similar distributed architecture as in [4]. They distribute the resampling to the PEs. The objective of distributed resampling is to reduce the communication among the PEs. They note that there is no study that optimizes the particle exchange among the PEs at that time. They propose an optimization algorithm for the particle exchange. They claim that they achieve improvement with the proposed algorithms [3].

Hong and his co-workers design a parallel system with four PEs and one CU. They use RPA and RNA algorithms described in [4] in the resampling stage. Each PE performs particle generation and weight update operations in parallel. They compute local sum of the weights and send it to the CU. Then CU computes the total sum of the weights. Weight normalization is done locally on each PE. Then each PE performs resampling locally. CU is also responsible for particle exchange after resampling. They describe 7 different configurations based on the particle exchange and weight normalization operations. The configurations are classified in terms of the architectures of PEs and resampling algorithms. They test their algorithms on a bearing only tracking problem. They state that they reduce the execution time of the resampling stage by a factor of the number of PEs [14].

Chao and his co-workers claim that there was no study in the literature which aim to

explore the utilization of CUDA in terms of data locality, execution model and memory hierarchy at the time they did their work [6]. They propose a technique, namely finite redraw importance maximizing (FRIM) prior editing, for the importance sampling stage of the particle filter. They discuss prior editing technique in the literature. In prior editing, each particle is subjected to an acceptance test. They are re-drawn unless they pass the acceptance test. The condition in the test is based on the likelihood $p(y_k|x_k)$. The authors state that there are two drawbacks in this technique. The first one is about execution time of the technique. It is unbounded and unsure. The second one is about warp divergence in the acceptance test. It causes some threads to be idle in the warp which leads to inefficient process of the sampling instructions. In their proposed technique, they define constant limit for the acceptance test. They draw particles N_{rdw} times to choose the best one as the results of importance sampling stage. They also introduce localized resampling. They distribute particles to N_{part} subsets evenly and each performs local resampling. Although the quality of resampling degrades, using localized resampling with FRIM lessens this degradation. This is occurred by the more particles with closer weights which are obtained in FRIM stage. They compare their approach with the standard particle filter on a bearing only tracking application. They state that they achieve similar quality with less number of particles when compared with the standard particle filter. They also state that setting N_{rdw} and N_{part} is a trade-off between quality and speed.

Shabany develops hardware efficient architectures for the sequential Monte Carlo (SMC) receiver algorithm. It consists of three blocks such that SMC core, weight calculator and resampler. The main contribution of his study is related to resampling stage. The execution time of his resampling algorithm does not depend on the distributions of the weights. In distributed implementation of resampling, they distribute the job evenly in two stages. In the first stage, replication factors are calculated. He claims this operation can perform in parallel efficiently. The second stage is the sample routing. He claims that it is hard to parallelize sample routing efficiently. He proposes a scheme where it has small number of PEs and a central control unit to handle the sample routing. This scheme is fixed and its execution time does not depend on the distribution of the weights in the PEs. Thus, resampling stage of the current time step and sampling stage of the next time step can be pipelined, which leads short

execution time [34].

Chitchian and his co-workers introduce local sub-filters. Each thread block represents a local sub-filter. Most of the stages of the particle filter are performed locally in these sub-filters including resampling with some limited communication with the other sub-filters. They propose a distributed resampling mechanism. Each sub-filter sends its t best particles to its neighbors and performs local resampling with $m+N_i*t$ particles where m is the number of particles in a sub-filter and N_i is the number of neighbors of a sub-filter. They define a graph topology to show the neighbor relationship of the sub-filters and they explore different topologies. They note that t and the topology are user-defined parameters. They achieve comparable accuracy with the sequential particle filter along with 10x to 100x speed up. They state that they surpass the existing distributed implementations in the literature [7].

Pan and his co-workers propose a hierarchical resampling (HR) method for the distributed particle filters. It decomposes the resampling stage into two hierarchies, namely, intermediate resampling (IR) and unitary resampling (UR). Assume N is the number of particles and K is the number of PEs. In the first hierarchy IR, each PE performs sampling and weight calculation for a single particle in parallel. Then the summation of the weights of K particles is calculated and resampling is performed for these K particles. Resampled particles are sent back to the PEs. The above operations repeat N/K times. Then second hierarchy UR resamples N/K particles on each PE. The summation of the weights which is calculated in the first hierarchy is used in resampling. In order to improve the UR stage, they use residual cumulative resampling (RCR) to pipeline and accelerate UR. HR achieves same accuracy with the standard distributed resampling algorithms in the literature. It also eliminates the particle redistribution stage along with some advantages such as fast execution time and efficient memory usage [31].

Tian and his co-workers propose a resampling tree scheme (RTS) for the resampling stage of their proposed distributed particle filter algorithm with resampling tree (DART). The resampling stage consists of two parts, branch resampling (BR) and root resampling (RR). In the tree structure, the weights are stored in the leaves of the tree initially. First, BR algorithm performs resampling by starting from the leaves to

the root. Then RR performs the remaining job. They state that their proposed method DART achieves speed up which surpasses linear boundary and outperforms state of art approaches [35].

Some of the authors of the studies cited in this section distribute the particles to the PEs and each PE performs local resampling. CU operates the particle routing between the PEs to balance the particles on each PE. Balasingam and his co-workers also optimize the particle routing. Chao and his co-workers offer a technique for the importance sampling stage of the particle filter to improve the quality of the local resampling. Shabany also offers a scheme to make the execution time of the particle routing not to depend on the distribution of the weights. There are also hierarchical or tree resampling schemes for the resampling stage of the distributed particle filter. In our proposed method Uphill resampling, each thread runs for each particle and performs local computations except the read/write operations of the weights in the global memory. There occurs non-coalesced global memory access pattern when reading the weights. We offer coalesced version of the Uphill resampling. It eliminates most of the access problem but still some non-coalesced access pattern occurs. We also devise another version of it. In this version, each warp selects some portion of the weight set randomly and performs resampling within this portion of data set. It behaves as local approach and eliminates the non-coalesced access problem. Differently from the studies above, we distribute the portion of the weights to the warps randomly. We can adjust the number of data in a portion of the weights which enables us to achieve a trade-off between speed and quality. Small size of weight portion means faster execution time but a limited portion of the weight set. Large size of weight portion means slower execution time caused by the non-coalesced access problem but better quality.

4.3 Studies on Metropolis Resampling

Liu and his coworkers devise parallel implementations of the Metropolis, Rejection and Systematic resampling methods on FPGAs. They note that this is the first work that implements these resampling methods on FPGA and they compare them with the GPU implementations of them. They offer memory access strategies for Metropolis and Rejection to improve their randomized access to the global memory. They modify

the Systematic resampling to save resources and achieve speed up. They compare the performance of them with the performance of their implementations on the GPU. They achieve significant speed up between 1.7x-49x over the GPU implementations of these methods. They point out that the Metropolis and Rejection resampling suffer from "warp divergence" on the GPU. They also note that with the number of particles around one million, the usage of off-chip FPGA memory will be necessary, which can limit the performance of FPGA in transferring time of the weights [21].

Liu and his co-workers also mention the drawbacks of the Systematic, Metropolis and Rejection resampling. These are numerical instability problem and parallel implementation of the cumulative summation of the weights for Systematic; randomized access to the global memory and warp divergence for Metropolis and Rejection; biased results for Metropolis. They state that FPGA based solutions overcome most of these problems. They distribute the particles into M parallel processing blocks. They define a simplified random permutation generator (RPG) which connects the memory to the parallel processing blocks so that each processing blocks can access the weights in the memory randomly. However, this randomized access to the memory causes non-coalesced global memory access patterns on the GPU. We overcome this problem by redirecting the threads in a warp to access the contiguous location of the global memory. We do not need to define any additional structures. We achieve faster results by sacrificing quality in a controllable manner.

CHAPTER 5

MEMORY COALESCING VARIANTS OF METROPOLIS RESAMPLING

In this chapter, we introduce memory coalescing variants of the Metropolis resampling named Metropolis-C1 and Metropolis-C2. We show how we solve the non-coalesced global memory access problem of Metropolis in detail. We compare M, C1 and C2 in terms of bias, MSE and execution time. We also compare C1 and C2 with M by reducing B parameter of M in two scenarios. In the first scenario, we match the MSE results of C1 and C2 with M by reducing B. In the second scenario, we match the execution time of C1 and C2 with M by reducing B. Furthermore, we also compare three resampling algorithms on a highly non-linear example. At the end, we discuss L1 cache usage of three resampling algorithms. A big part of the material presented in this chapter has appeared in [9].

5.1 Metropolis-C1 and Metropolis-C2 Resampling

We devise two solutions for the non-coalesced global memory access problem of the Metropolis resampling. The access pattern of Metropolis is given in Figure 5.1. The threads in the same warp access to the different segments of the global memory. We try to force them to access the same segment. We define an *s-segment* as a set of a fixed number of contiguous segments. Thus, the smallest s-segment is a single segment, and the largest s-segment is the set of all segments that spans the weight array. The set of all s-segments forms an equal-sized partitioning of the weight array. An illustration about s-segment is given in Figure 5.2.

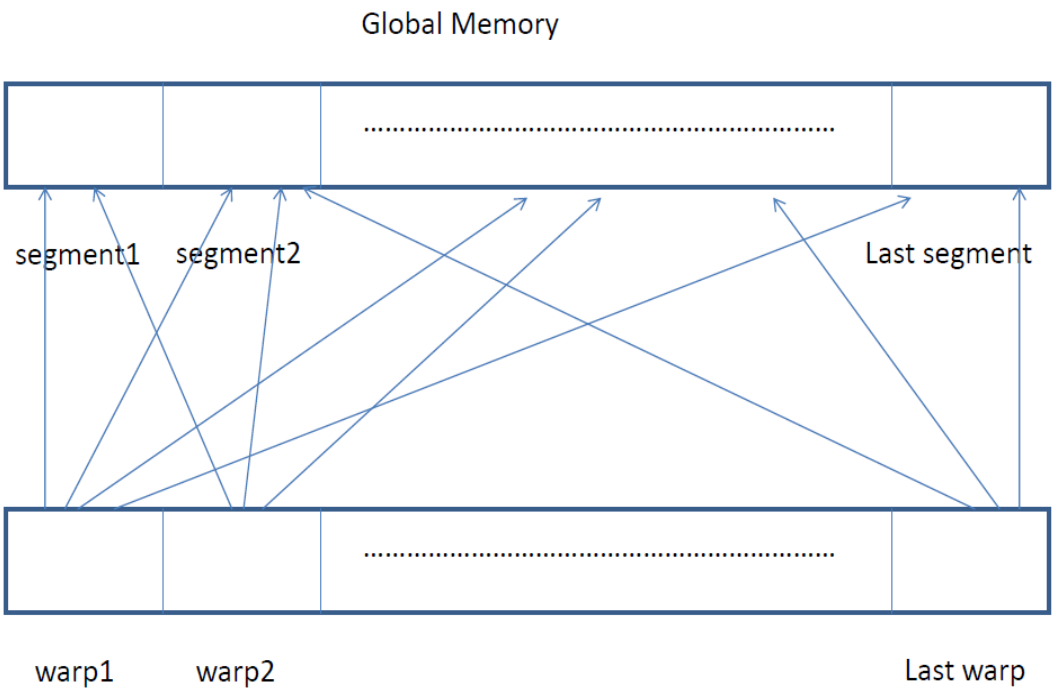

Figure 5.1: The global memory access pattern of Metropolis.

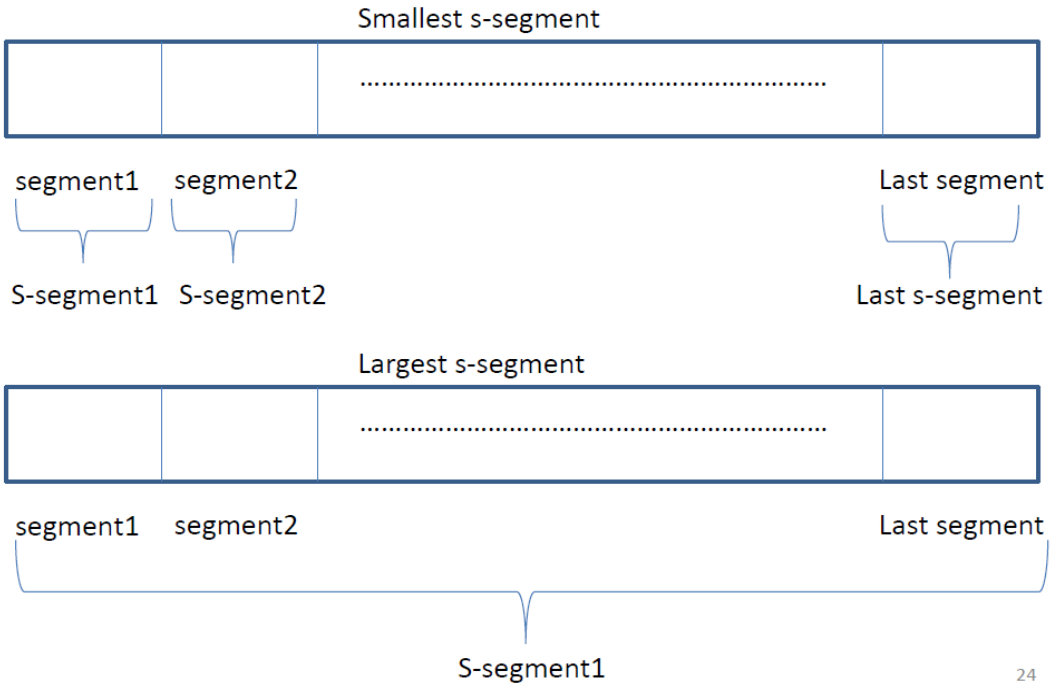

Figure 5.2: The smallest and largest s-segment.

The stages of Metropolis-C1 are given in Algorithm 5.1. In this technique, each warp selects a random s-segment. All the threads in the warp select random weights within this s-segment.

Algorithm 5.1 Metropolis-C1 Resampling

1: **procedure** $[\{x^i_{new}\}_{i=1}^N]$ = METROPOLIS-C1($\{x^i, \tilde{w}^i\}_{i=1}^N, B, SC, DC$)
2: foreach $i = 1 : N$
3: $t = i$
4: $s \sim v\{1, .., SC\}$
5: for $m = 1 : B$
6: $u \sim v[0\ 1]$
7: $j \sim v\{(s-1) * DC + 1, \ldots, s * DC\}$
8: if $u \leq \tilde{w}^j / \tilde{w}^t$ then $t = j$ end if
9: end for
10: $x^i_{new} = x^t$
11: end foreach

In Algorithm 5.1, SS is the size of an s-segment in bytes; SC is the number of s-segments; its value is $4N/SS$. The value of N must be a power of 2. The DC is the number of weights in an s-segment; its value is $SS/4$. The j is a random integer between the indexes of the first and the last element of the s-segment drawn from a uniform distribution; s is the index of the selected s-segment for a warp, drawn from a uniform distribution. All the threads in a warp must have the same value of s. This can be ensured by using the index of the warp as the sequence of the random number generator. The constant 4 is the size of the single-precision floating-point number in bytes.

The stages of Metropolis-C2 are given in Algorithm 5.2. The parameters of this algorithm are as in the Algorithm 5.1. The only difference from C1 is the selection of an s-segment. In C2, each warp selects a random s-segment at each iteration of the inner-loop. The selection of s for the threads in a warp can be done as in C1. Note that as the s-segment size, SS, is selected larger both algorithms become more similar to the original Metropolis algorithm, and for the limiting case of $SS = 4N$ both become behaviorally the same as Metropolis.

Algorithm 5.2 Metropolis-C2 Resampling

1: **procedure** $[\{x^i_{new}\}_{i=1}^N]$ = METROPOLIS-C2($\{x^i, \tilde{w}^i\}_{i=1}^N, B, SC, DC$)
2: foreach $i = 1 : N$
3: $t = i$
4: for $m = 1 : B$
5: $u \sim v[0\ 1]$
6: $s \sim v\{1, .., SC\}$
7: $j \sim v\{(s-1) * DC + 1, \ldots, s * DC\}$
8: if $u \leq \tilde{w}^j / \tilde{w}^t$ then $t = j$ end if
9: end for
10: $x^i_{new} = x^t$
11: end foreach

To assess the quality of the resampling algorithms, we use the metrics and distributions given in Section 2.5. We create 16 weight sequences from each distribution. We draw 256 offspring sequences from each weight sequence. The experimental results of the distribution are the average of the results of these 16 weight sequences. We execute each one of the three resampling algorithms in this framework. We consider the number of particles from 2^4 to 2^{22}. When the s-segment size is greater than $4N$, we set its value to $4N$. We choose the number of repetitions of each run, that is K, as 256. The output of the algorithms is the ancestor array (rather than the states of the particles). We estimate the confidence interval (CI) of each run with a level of 99% to see how close the sample mean is to the true mean [33]. The largest value of CI in our quality experiment is $\pm 15.717\%$ of the sample mean and in our speed experiment is $\pm 8.124\%$ of the sample mean. The smallest value of CI in our quality experiment is $\pm 0.011\%$ of the sample mean and in our speed experiment is $\pm 0.002\%$ of the sample mean. We observe that as the number of particles gets larger, CI tends to get smaller.

To assess the speed of the resampling algorithms, we calculate the execution times of the kernels of each resampling algorithm along with the execution times of the kernels to compute the value of B. The speed up is the ratio of the execution time of Metropolis over the execution times of the proposed algorithms.

We compute the value of B as prescribed in [22]. Therein we choose ϵ as $1/100$ and

β as $\bar{\tilde{w}}/\tilde{w}_{max}$ where $\bar{\tilde{w}}$ is the mean of the weights and $\tilde{w}_{max}$ is the maximum weight. We use the same B values for all three algorithms. We use the efficient reduction algorithm in calculations [11].

5.2 Bias, MSE and Execution Time Results

In this section, firstly, we compare bias, MSE and execution time results of C1, C2 and M where C1 and C2 achieve their highest speed. Secondly, we compare them on the same metrics where MSE of C1 and C2 are very close to MSE of M but in a faster manner. We set the s-segment size in two different scenarios. In the first scenario, we assign the value of SS as 128. The threads in the same warp read the randomly chosen weights in a single transaction in C1 and C2. The bias and MSE results of the M, C1 and C2 resampling on the gamma distributions are given in Figure 5.3. Their respective execution time and speed up results are given in Figure 5.4.

The results in Figure 5.3 and Figure 5.4 indicate that both of these techniques are better than M in speed, but worse in quality because they select the weights from a limited portion of the weight sequence. The contribution of the squared bias to the MSE in C1 is larger than that in M. Since C1 focuses on local selections, the expected number of repetitions of the particles becomes different than that in M. This causes the bias results of C1 become large. The variance in C1 is also large, because s-segment size and warp size cause variance in the results of MSE. The contribution of the squared bias to the MSE in C2 is similar to the one in M. This shows us that the expected numbers of repetitions of the particles in both algorithms are quite similar. The MSE results of C2 are worse than those of M, because s-segment size and warp size cause variance in the results.

C2 is slower than C1 because it generates an extra random number at each iteration of the inner loop. As C2 selects a new s-segment at each iteration of the inner loop, it is more prone to non-coalesced memory reads of the $\tilde{w}^t$. However, it is better than C1 in quality because it encounters more variety in the selection of weights. Regarding the speed up results, C1 and C2 achieve their highest speed up due to complete elimination of the non-coalesced global memory read for $\tilde{w}^j$. But the speed

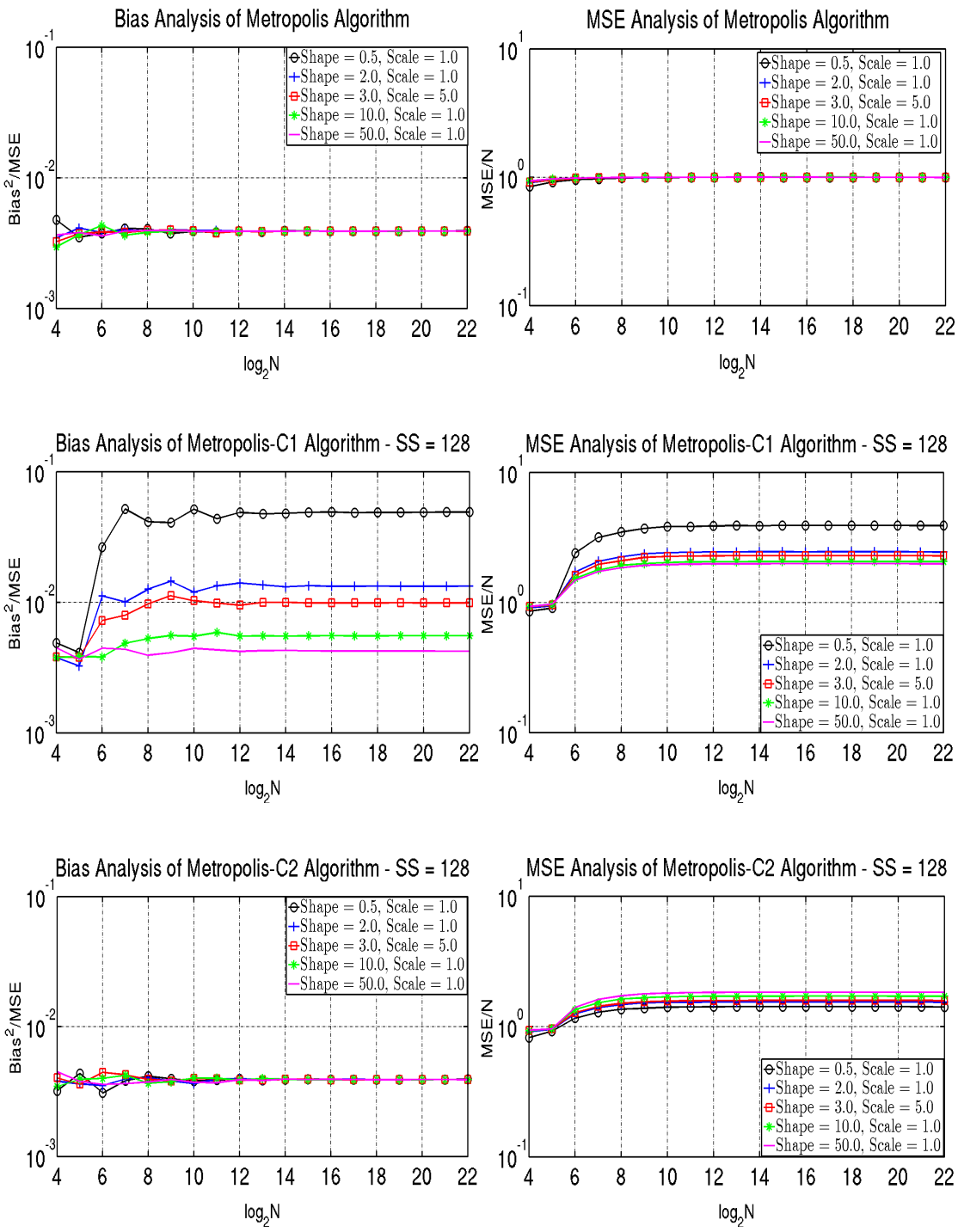

Figure 5.3: Bias and MSE results of the Metropolis, Metropolis-C1 and Metropolis-C2 resampling algorithms on the gamma distributions. S-segment size is 128 bytes.

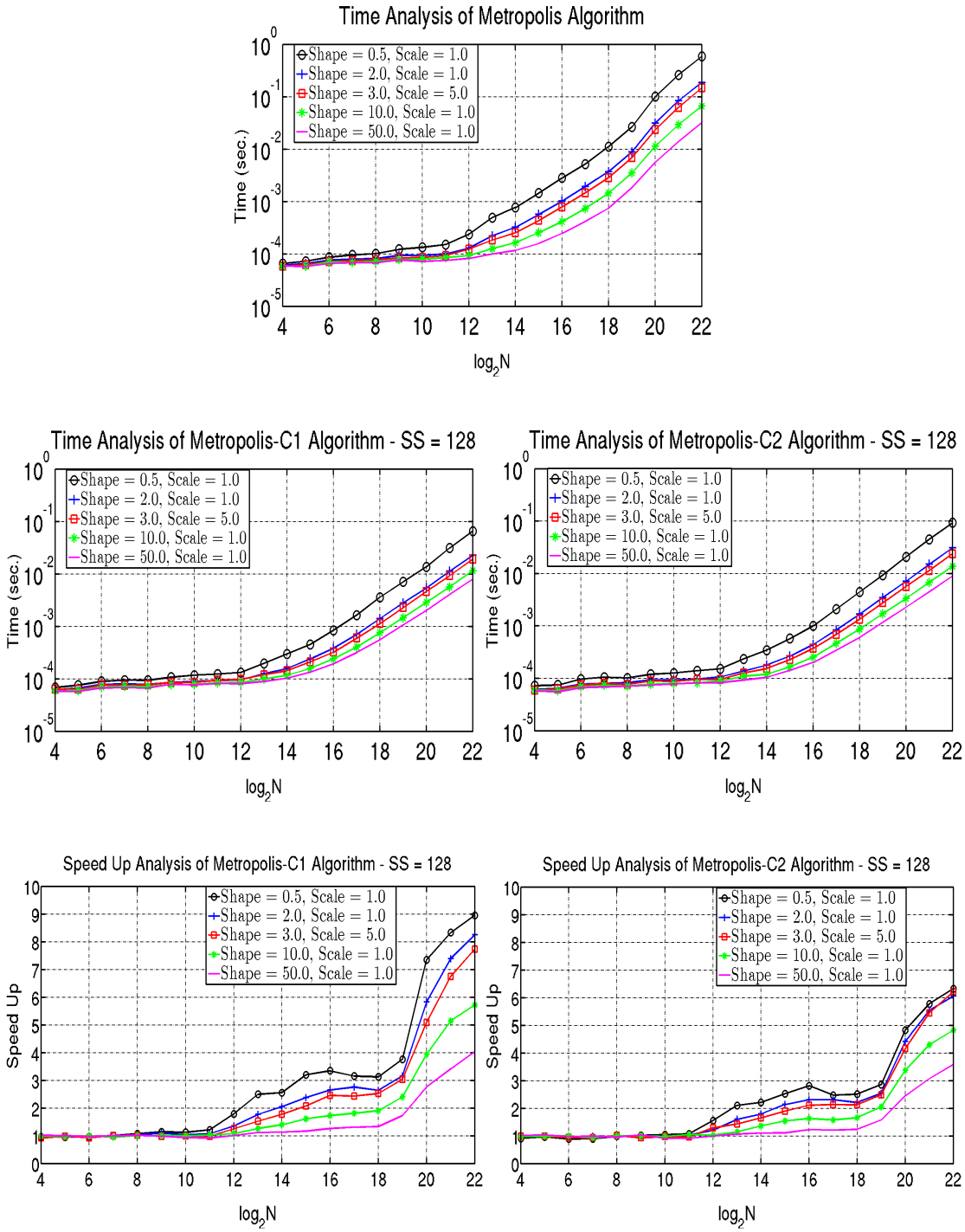

Figure 5.4: Execution time and speed up results of the Metropolis, Metropolis-C1 and Metropolis-C2 resampling algorithms on the gamma distributions. S-segment size is 128 bytes.

up of C2 is less than that of C1 because of the non-coalesced global memory read for $\tilde{w}^t$. When the number of particles exceeds 524288, the execution time of M becomes much worse where the improvements of C1 and C2 become more pronounced.

In the second scenario, we set SS to 2048. The threads in the same warp read the randomly chosen weights in at most 16 transactions. The bias and MSE results of the M, C1 and C2 resampling algorithms on the gamma distributions are given in Figure 5.5. Their respective execution time and speed up results are given in Figure 5.6.

The results in Figure 5.5 show that as the s-segment size is increased, both C1 and C2 pick the weights from a bigger portion of the weight sequence, which leads both techniques to approach M in quality. The contribution of the squared bias to the MSE in C1 becomes closer to the contribution in M as the s-segment size increases. Remember that C1 behaves same as M in quality when the s-segment size is $4N$. The variance caused by s-segment also reduces as well. Consequently, the MSE of C1 becomes similar to the MSE in M as the s-segment size increases. The contribution of the squared bias in C2 is similar to that in M since they have similar expectation on the number of repetitions of the particles. The variance caused by s-segment size reduces as the s-segment increases which leads to C2 give MSE results very close to M.

Regarding the results in Figure 5.6, the execution times of C1 and C2 are longer than their execution times in the first scenario due to lesser coalescing, but they are still faster than M. The speed up of C2 is less than that of C1 since C2 selects a new s-segment at each iteration which allows non-coalesced global memory read for $\tilde{w}^t$. When the number of particles exceeds 524288, the execution time of M becomes much worse where the improvements of C1 and C2 become more pronounced.

The bias and MSE results of M, C1 and C2 resampling algorithms on the distributions in (2.33) with the s-segment size 128 are given in Figure 5.7 and their respective execution time and speed up results are given in Figure 5.8. The bias and MSE results of resampling algorithms on the same distributions with the s-segment size 2048 are given in Figure 5.9 and their respective execution time and speed up results are given in Figure 5.10.

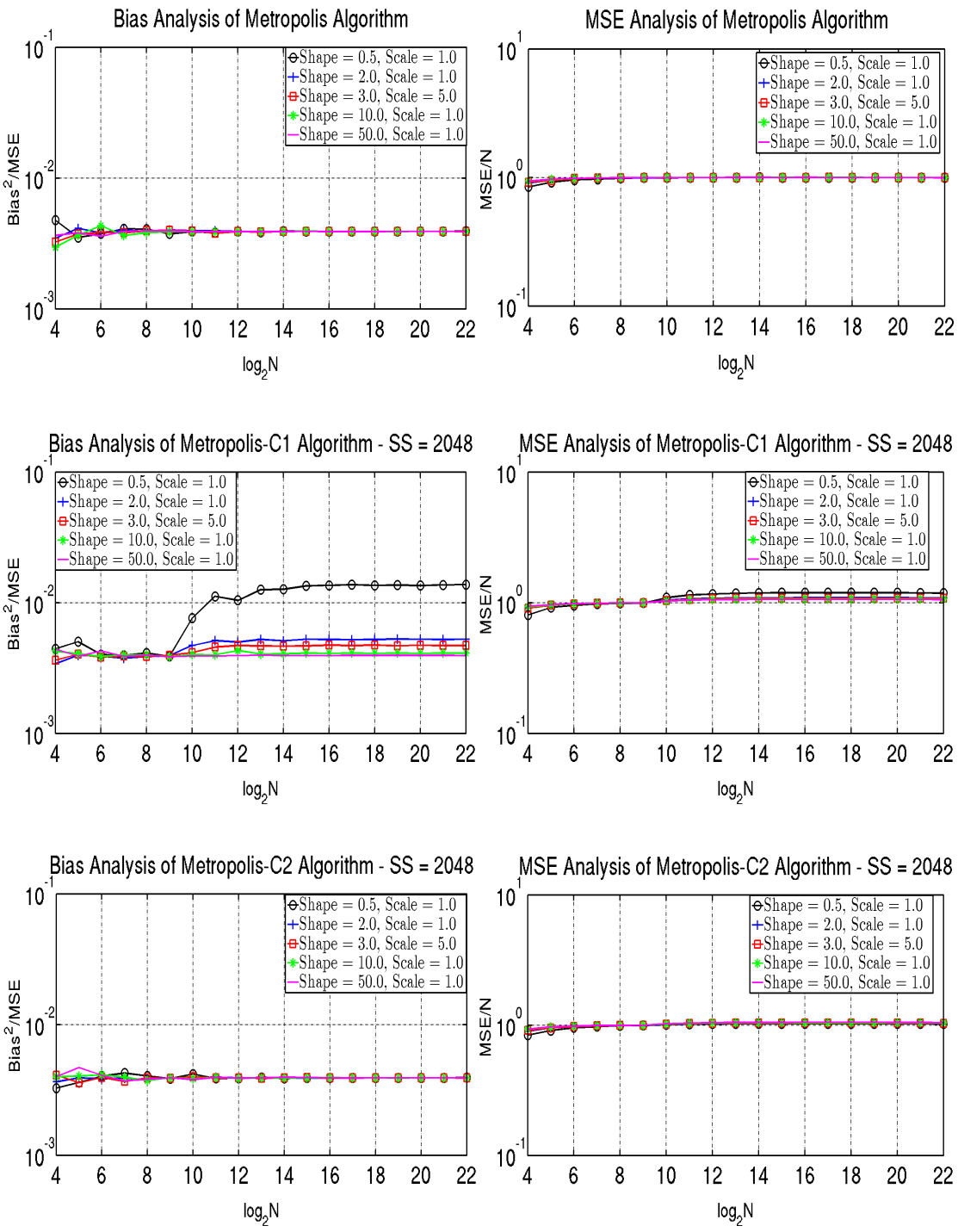

Figure 5.5: Bias and MSE results of the Metropolis, Metropolis-C1 and Metropolis-C2 resampling algorithms on the gamma distributions. S-segment size is 2048 bytes.

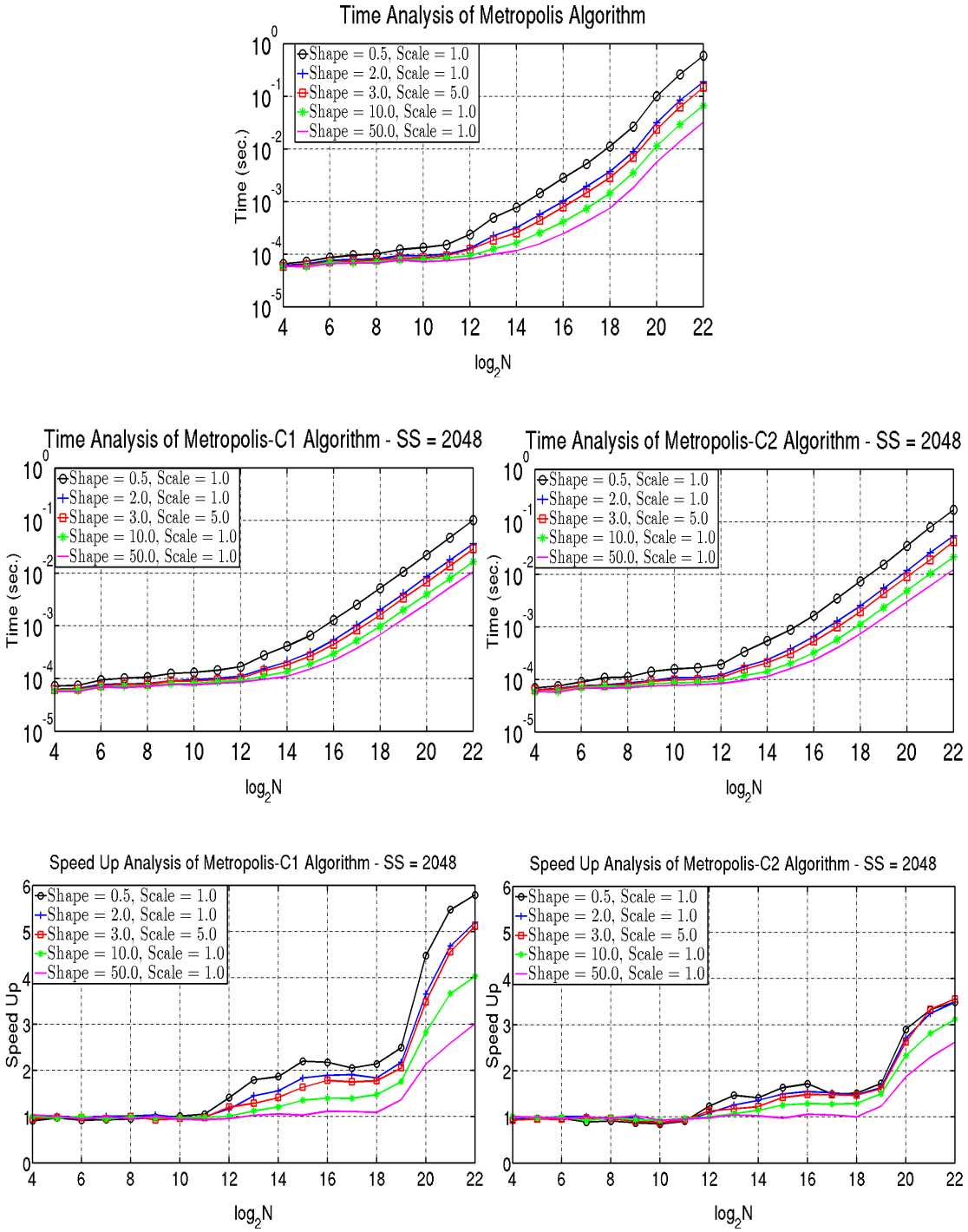

Figure 5.6: Execution time and speed up results of the Metropolis, Metropolis-C1 and Metropolis-C2 resampling algorithms on the gamma distributions. S-segment size is 2048 bytes.

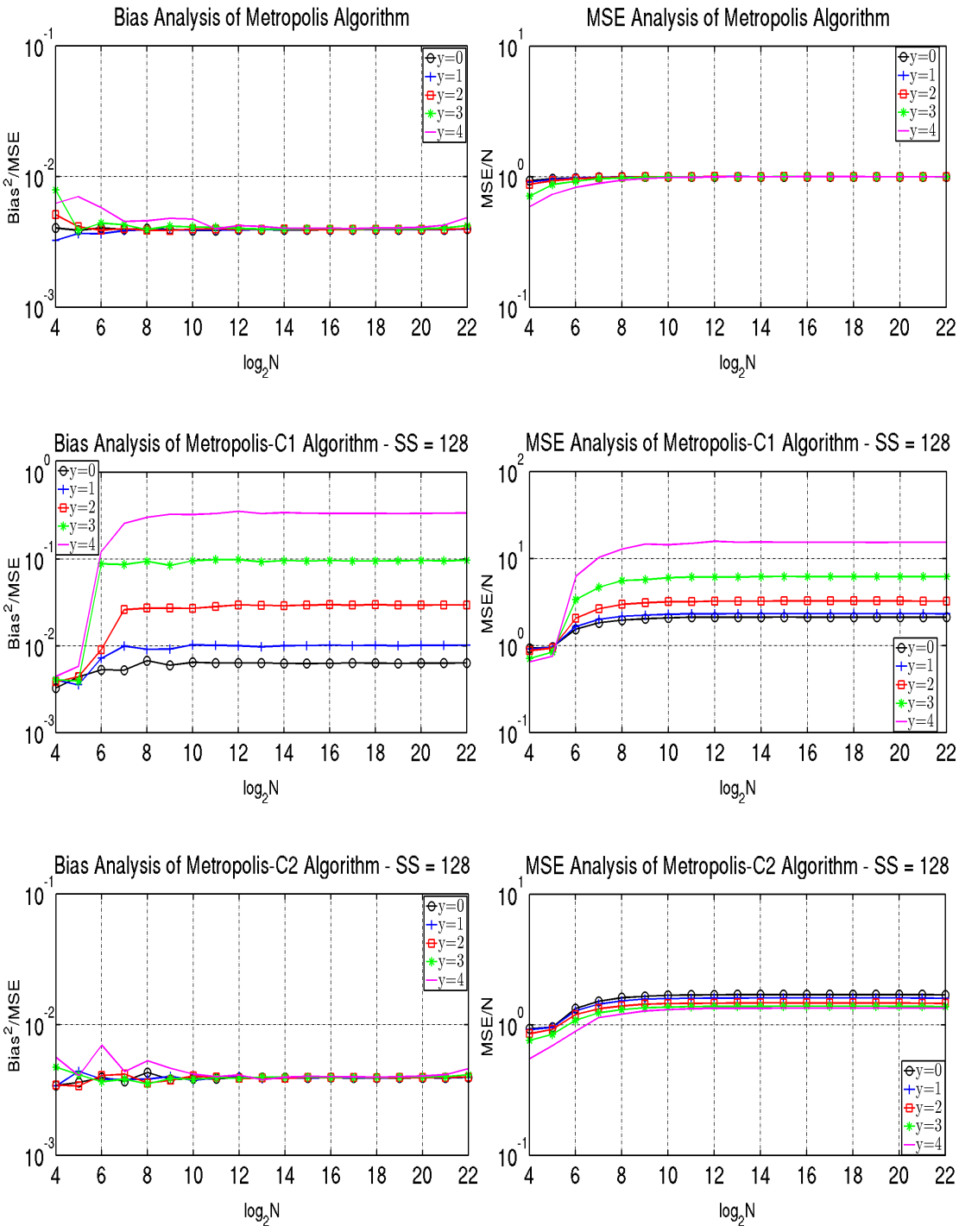

Figure 5.7: Bias and MSE results of the Metropolis, Metropolis-C1 and Metropolis-C2 resampling algorithms on the distributions in (2.33). S-segment size is 128 bytes.

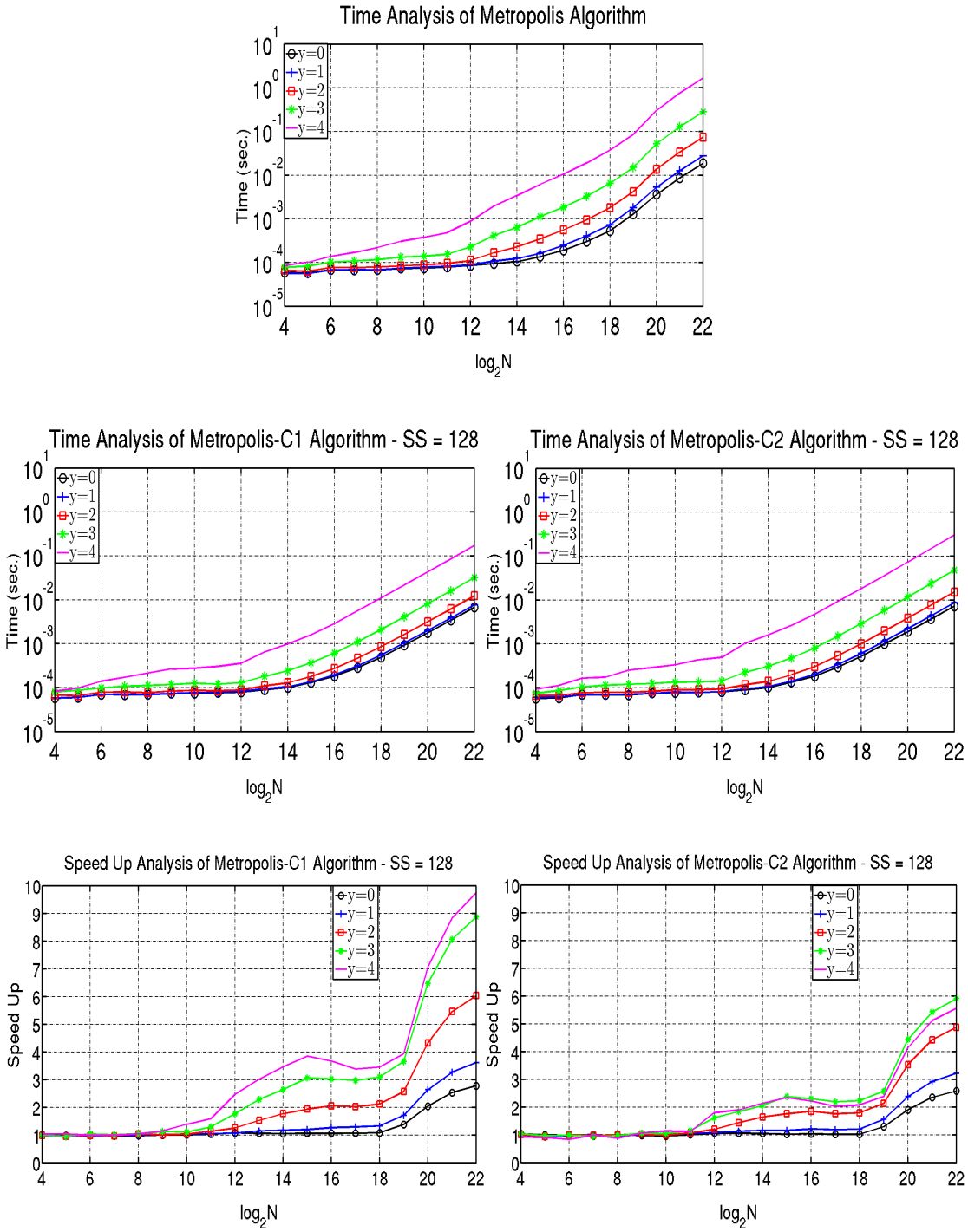

Figure 5.8: Execution time and speed up results of the Metropolis, Metropolis-C1 and Metropolis-C2 resampling algorithms on the distributions in (2.33). S-segment size is 128 bytes.

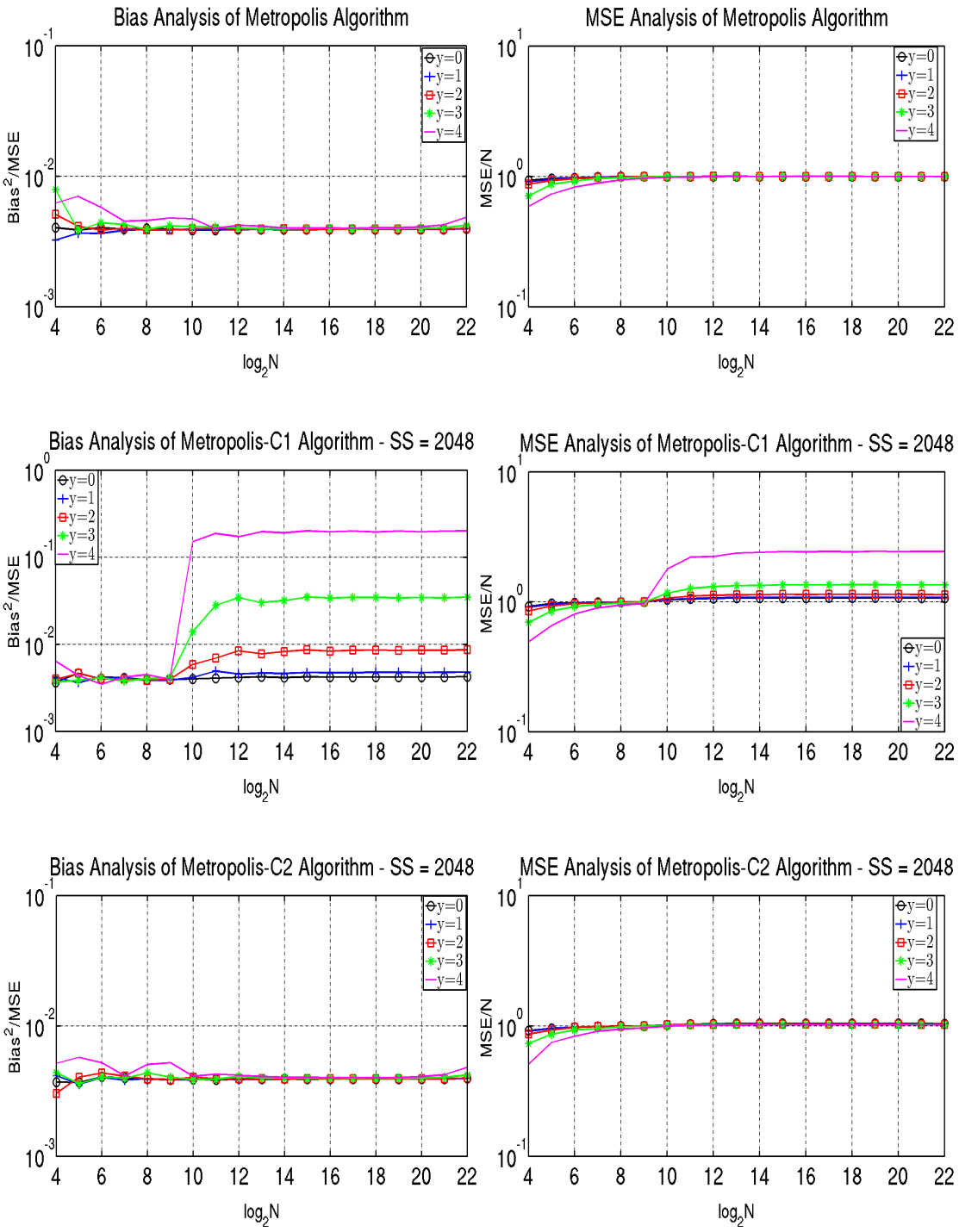

Figure 5.9: Bias and MSE results of the Metropolis, Metropolis-C1 and Metropolis-C2 resampling algorithms on the distributions in (2.33). S-segment size is 2048 bytes.

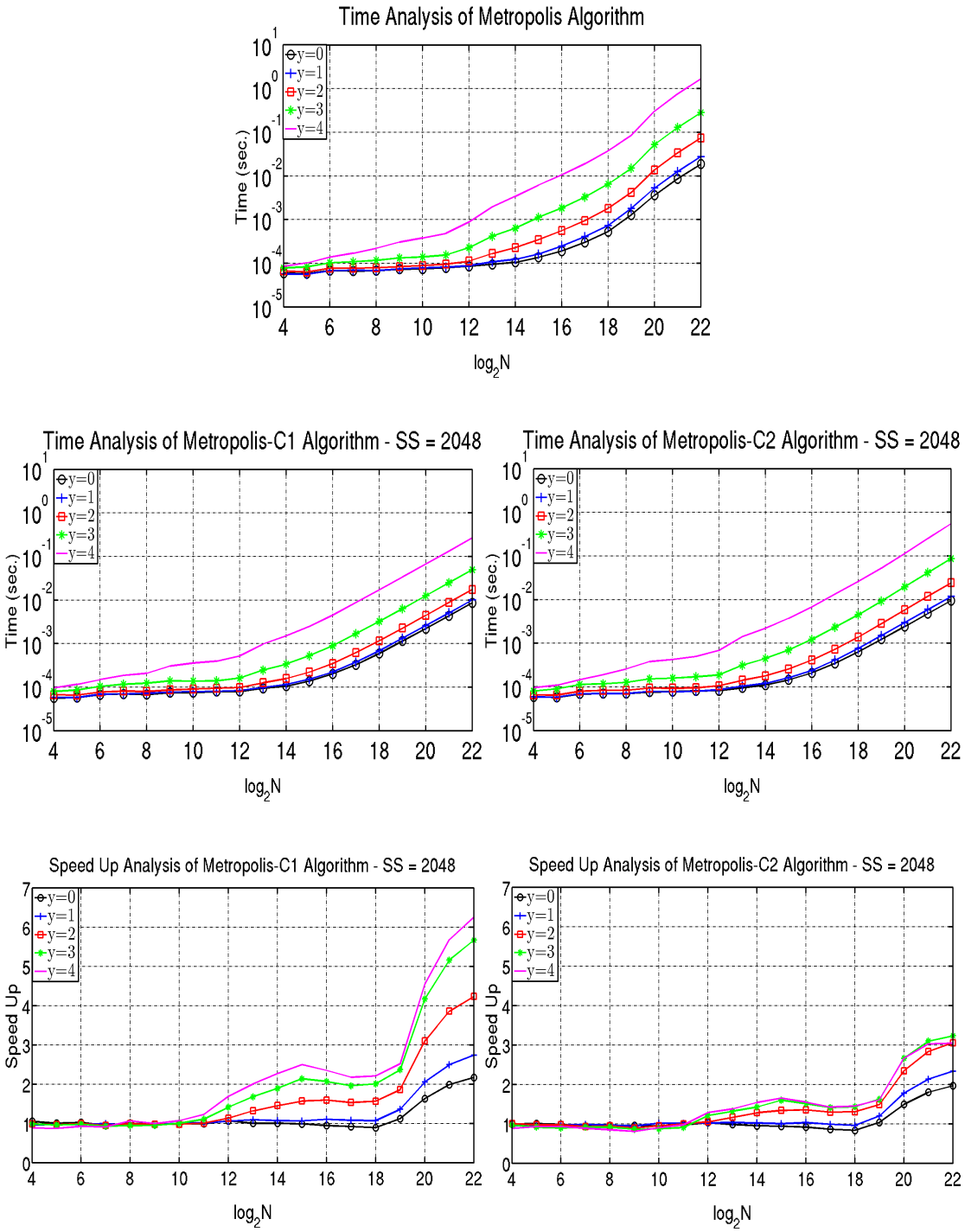

Figure 5.10: Execution time and speed up results of the Metropolis, Metropolis-C1 and Metropolis-C2 resampling algorithms on the distributions in (2.33). S-segment size is 2048 bytes.

The performance of C1 and C2 resampling are similar with the previous experiments on the gamma distributions. In addition to this, we can see how the relative variance in the weights affects the algorithms. The bias and MSE results of the C1 worsen as the relative variance in weights increases and show diminishing improvement as the s-segment size increases. However, the bias results of the C2 are not affected from the relative variance in weights. The MSE results of C2 are only affected from the s-segment size as in the previous experiments on the gamma distributions. For a targeted quality, the execution times of all resampling algorithms become longer as the relative variance in weights increases. The improvements of C1 and C2 become more pronounced with increased relative variance.

5.3 Discussion on the B Parameter

In the matter of speed vs. quality trade-off, an intriguing possibility is to reduce the value of B to run Metropolis faster at the expense of some decrease in quality. In this section we explore the feasibility of this approach in comparison with C1 and C2. For the first comparison we assume the quality is at a premium, and for the second comparison we assume the speed is at a premium.

In the first comparison, we match the MSE results of the three resampling algorithms by reducing B in M to see how far M can be sped up by iterating the inner loop less without sacrificing the quality significantly. Note that, we do not reduce B in C1 and C2. Since quality is important, the user should use C1 and C2 with relatively large s-segment sizes. Thus, in this set of trials we pick s-segment size as 2048. We look into a case where M, C1 and C2 have roughly the same MSE, and M is slowest. We use the distributions in (2.33). The bias, MSE and execution time results of M and C1 are given in Figure 5.11. The bias, MSE and execution time results of M and C2 are given in Figure 5.12. We do not consider the execution time of calculating B in this experiment since it is not calculated in M.

The results in Figure 5.11 and Figure 5.12 show that these resampling algorithms have different behaviors in quality even if their MSE results are very close. M exceeds the speed of C1 but its bias results become worse than C1. M exceeds the speed of C2

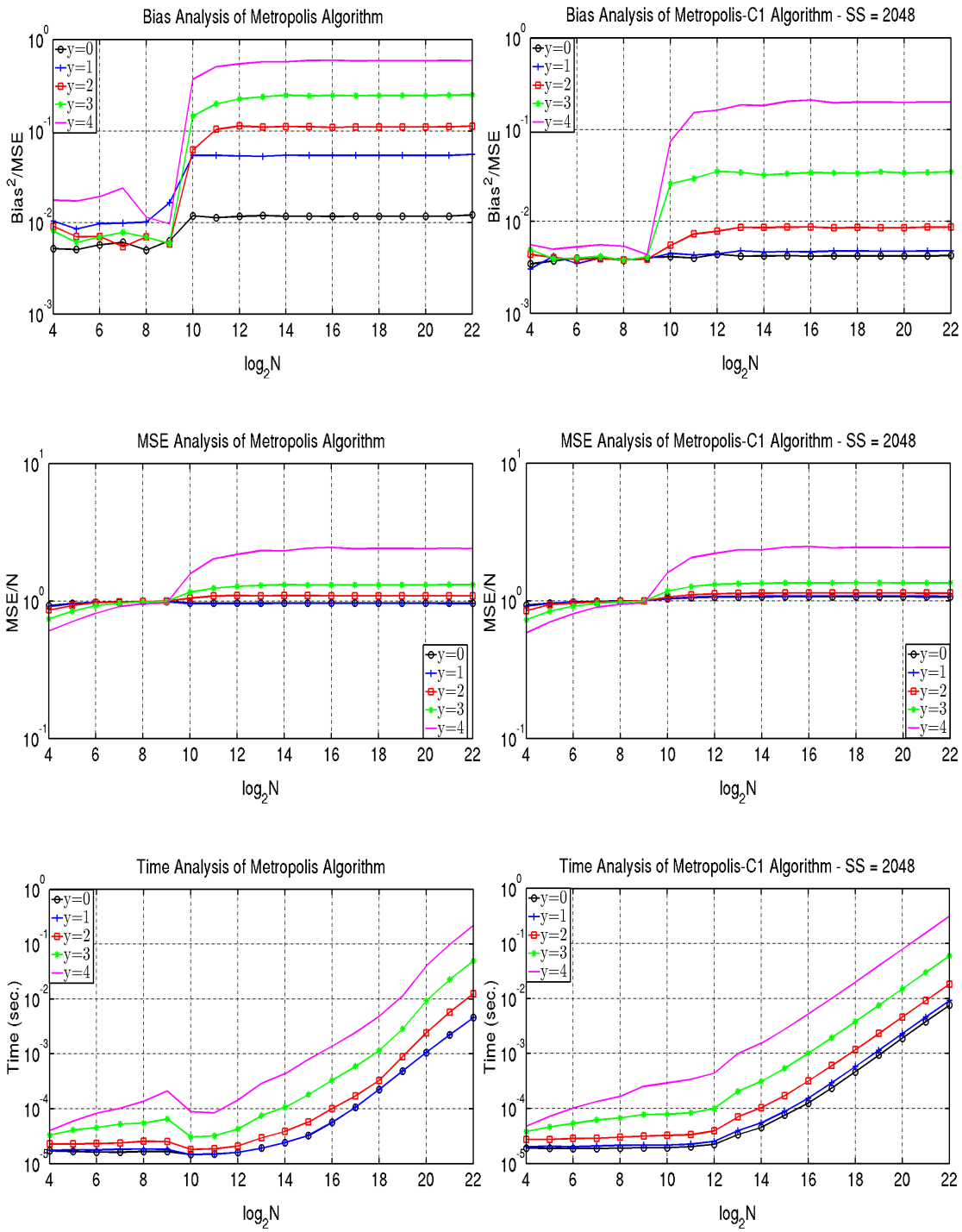

Figure 5.11: Bias, MSE and execution time results of the Metropolis and Metropolis-C1 resampling algorithms on the distributions in (2.33) where the MSE results of them are very close. S-segment size is 2048 bytes.

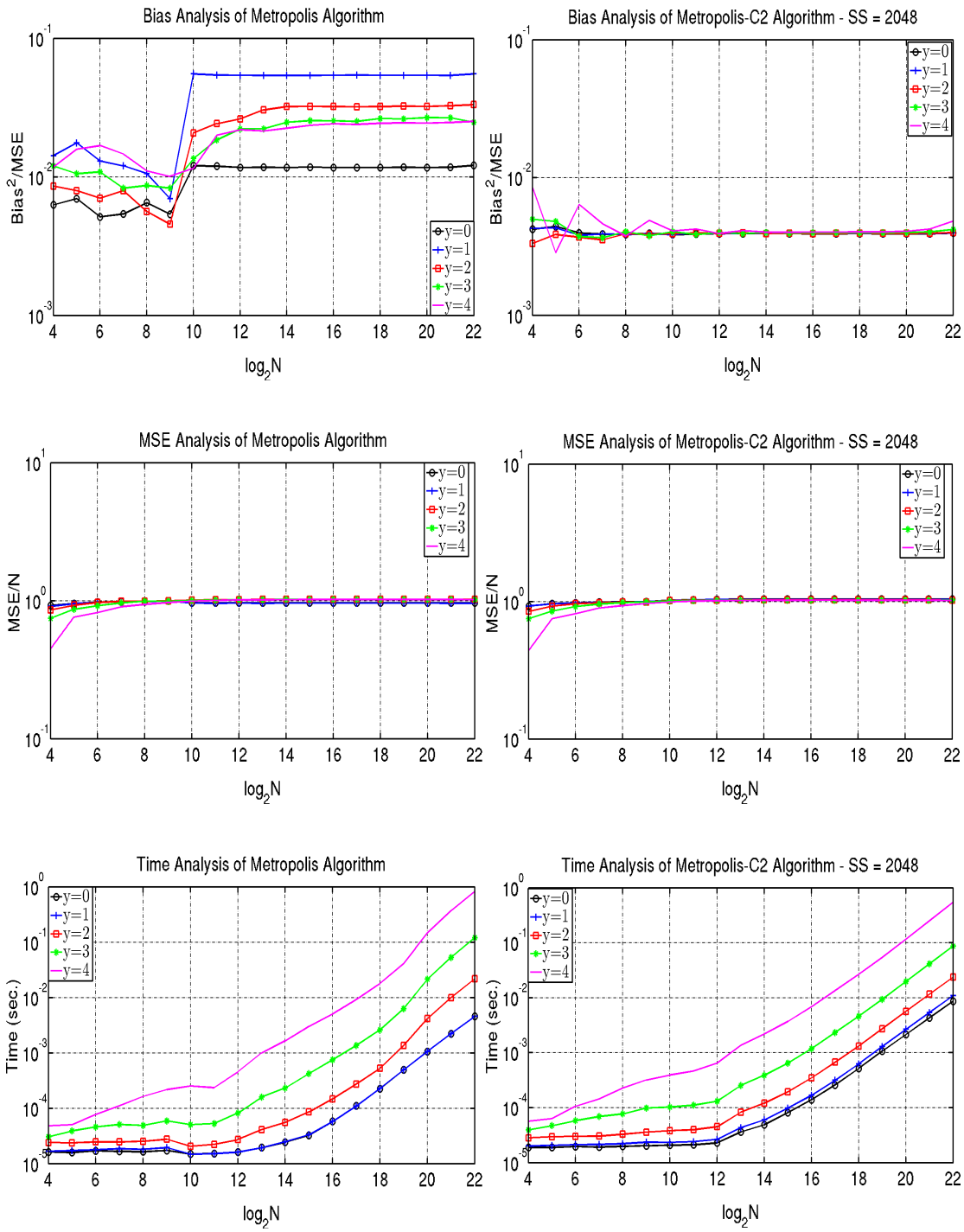

Figure 5.12: Bias, MSE and execution time results of the Metropolis and Metropolis-C2 resampling algorithms on the distributions in (2.33) where the MSE results of them are very close. S-segment size is 2048 bytes.

in the distributions where the number of particles or relative variance in the weight set is small. When the number of particles is very large, as the relative variance in the weight set increases, C2 becomes faster than M. The bias results of M become worse whereas the bias results of C2 are much better in all cases.

In the second comparison, we match the execution times of the three resampling algorithms by reducing B in M to see how much M sacrifices quality while it is sped up by matching the elapsed times of C1 and C2. Note that we do not reduce B in C1 and C2. Since speed is important, the user should use C1 and C2 with relatively small s-segment sizes. Thus, in this set of trials we pick s-segment size as 128. We look into a case where C1 and C2 have shorter elapsed time than M, whereas M yields higher quality results. We use the distributions in (2.33). The bias, MSE and execution time results of M and C1 are given in Figure 5.13. The bias, MSE and execution time results of M and C2 resampling are given in Figure 5.14. Again, we do not consider the execution time of computing B.

It is seen that M yields better MSE results than C2 in most of the cases. However, when the number of particles and the relative variance in the weight set are very large, the MSE results of C2 become better than those of M. On the other hand, the bias results of C2 are better than those of M as the number of particles increases. Although the contribution of the squared bias to the MSE in C1 is less than that in M when the number of particles is very large, the MSE results of M are less than the results of C1 in almost all cases.

This analysis suggests that playing with the B parameter of M does not achieve the same trade-off offered by C1 and C2. By reducing B, M sacrifices quality by allowing the bias to be large. On the other hand, by reducing s-segment size, C2 sacrifices quality by allowing the variance to be large. The situation is different for C1. Since C1 focuses on local selections, both bias and variance can be affected adversely by reducing the s-segment size. The performance of C2 is most promising since it yields smallest bias. The importance of bias in resampling is elaborated in [22].

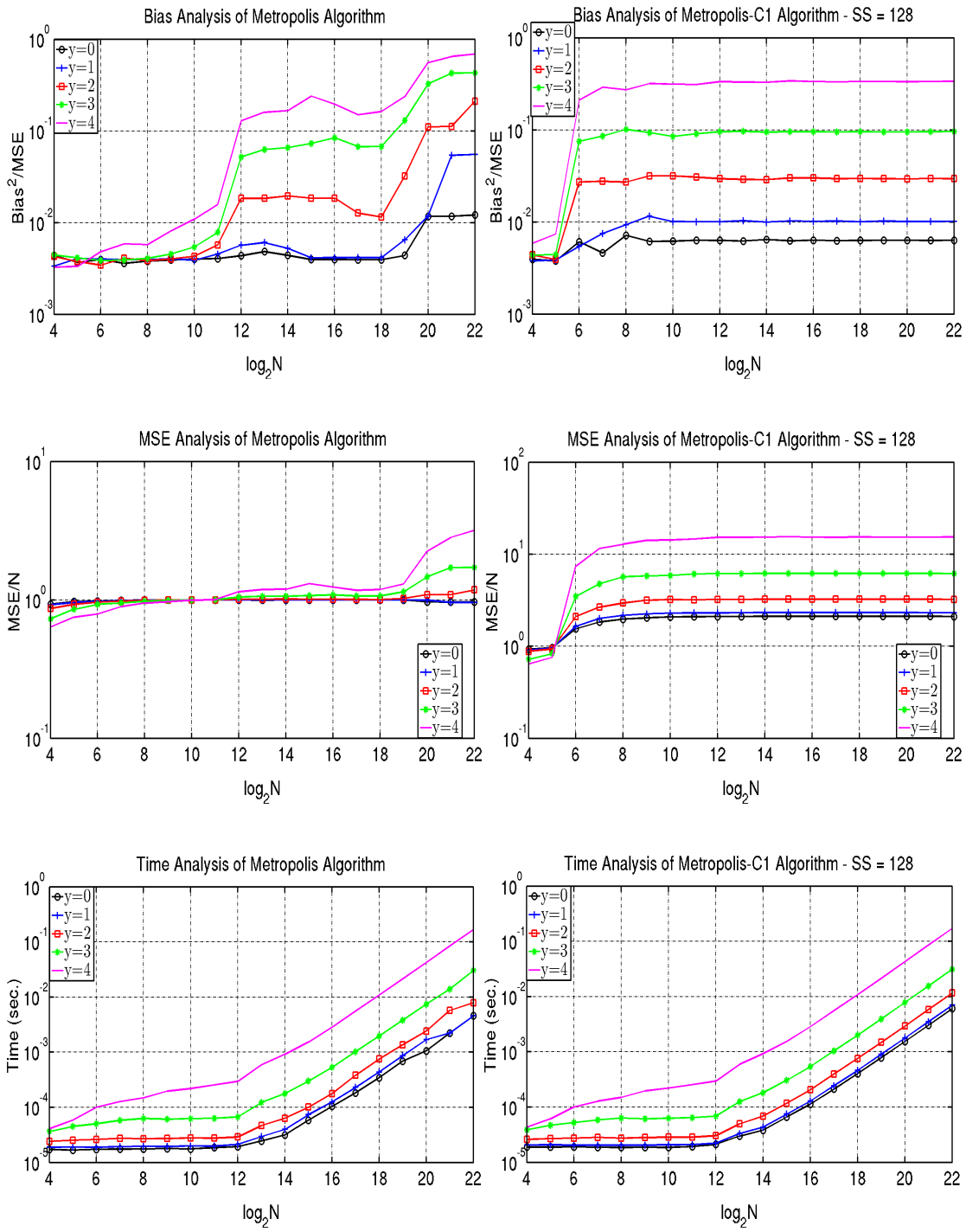

Figure 5.13: Bias, MSE and execution time results of the Metropolis and Metropolis-C1 resampling algorithms on the distributions in (2.33) where the execution time results of them are very close. S-segment size is 128 bytes.

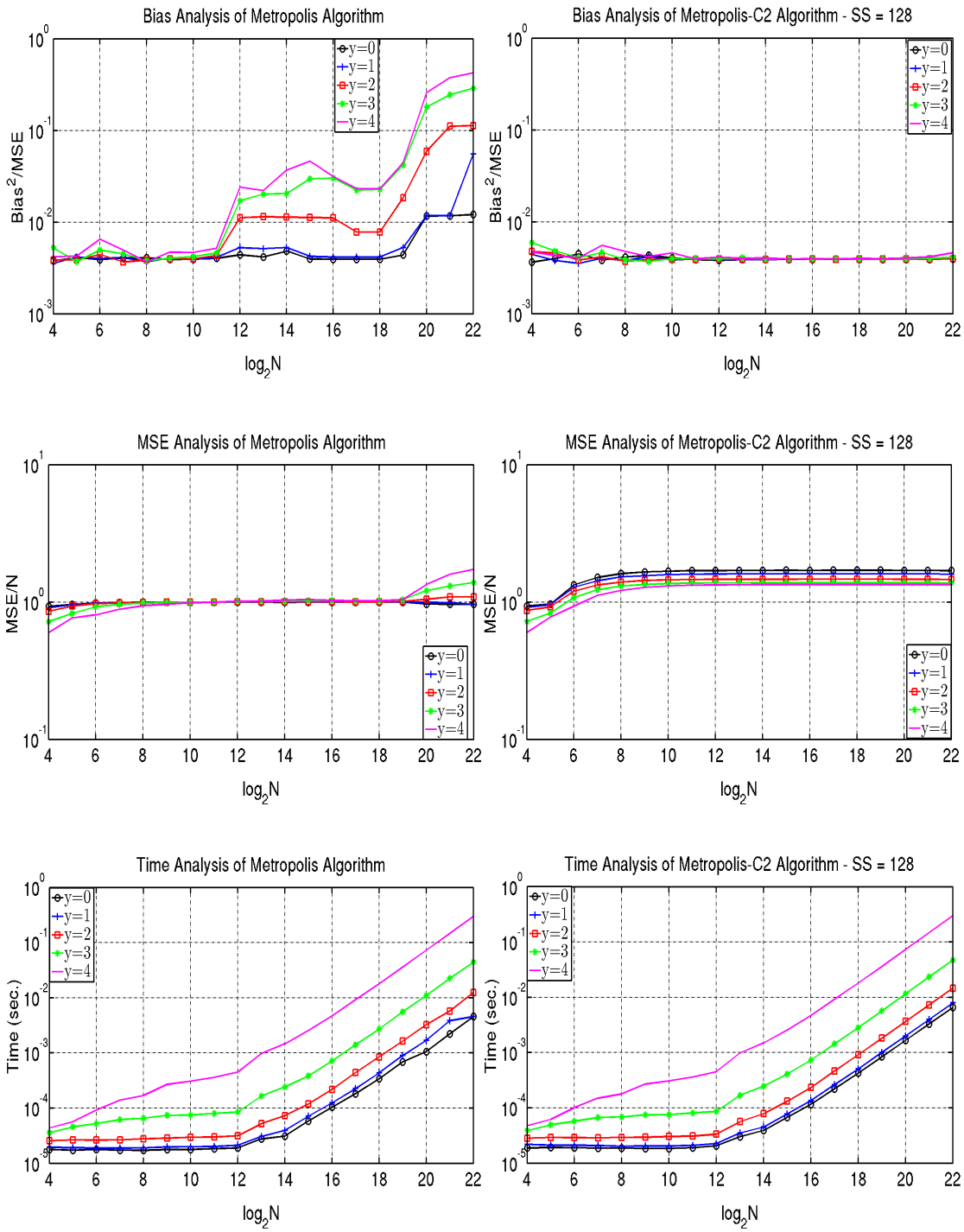

Figure 5.14: Bias, MSE and execution time results of the Metropolis and Metropolis-C2 resampling algorithms on the distributions in (2.33) where the execution time results of them are very close. S-segment size is 128 bytes.

5.4 A Simple End-to-End Application

To provide a perspective for the reader on the significance of the resampling stage in the overall cost of the SIR particle filter, we run the SIR particle filter in Algorithm 2.4 on a highly non-linear example system given in (2.34) and (2.35).

We create 16 different trajectories and run them 100 times with each one of the three resampling algorithms. We use these 16 trajectories in all the number of particles. The results are the average of the results of these 16 trajectories. We set the number of time steps to 100. The initial state x_0 is set to 0.1. We measure the quality using the root mean squared error (RMSE) metric [32]. We calculate the error at each time step by getting the difference between the true state and estimated state. We set ϵ to $1/10$ in calculation of B to accelerate the resampling algorithms without sacrificing quality much [22]. We set the *–use_fast_math* flag at compile time to enable efficient calculation of special functions such as cosines, square root, exponential [28]. We estimate the confidence interval (CI) of each run with a level of 99% to see how close the sample mean is to the true mean. The largest value of CI in our RMSE experiment is $\pm 0.191\%$ of the sample mean and in our execution time experiment is $\pm 0.751\%$ of the sample mean. In stage 4 of the SIR particle filter, we use the weights of particles without normalization since Metropolis does not need normalized weights. The ratios of each stage to the total execution times of the filter with varying numbers of particles are given in Table 5.1.

It is seen that the resampling stage takes a big portion of the total execution time. The behaviors of all resampling algorithms used in the previous experiments are manifest in these experiments. It seems that the improvements in execution times by C1 and C2 are important in PF applications. As the number of particles increases, the share of the resampling stage also increases, and the improvement in execution time brought about by C1 and C2 becomes more pronounced. The RMSE results of the experiments are given in Table 5.2. The results show us the RMSE results of M, C1 with $SS = 2048$, C2 with $SS = 128$ or 2048 are very close the each other. Using one of them is enough in this example. The RMSE of C1 with $SS = 128$ is slightly worse than that of M, but this may not be significant. Hence, one can prefer C1 with $SS = 128$, if speed is the primary concern.

Table 5.1: The ratio of times spent in different stages. The results are percentage of the total execution times. Total execution times (in seconds) are also given in the last row.

N=16384	Resampling Methods				
	M	C1-128	C1-2048	C2-128	C2-2048
Stage1	5.62	10.79	8.75	9.47	7.26
Stage2	5.30	9.96	8.07	8.74	6.70
Stage3	8.53	16.19	13.13	14.21	10.89
Stage4	80.55	63.05	70.04	67.58	75.15
Exec. Time	0.034	0.018	0.023	0.021	0.028

N=65536	Resampling Methods				
	M	C1-128	C1-2048	C2-128	C2-2048
Stage1	5.62	12.79	9.57	10.58	7.44
Stage2	2.50	5.69	4.26	4.71	3.31
Stage3	4.09	9.32	6.98	7.71	5.42
Stage4	87.79	72.20	79.19	77.01	83.83
Exec. Time	0.089	0.039	0.052	0.047	0.067

N=262144	Resampling Methods				
	M	C1-128	C1-2048	C2-128	C2-2048
Stage1	5.32	13.15	9.11	10.29	6.73
Stage2	1.20	2.98	2.06	2.33	1.52
Stage3	2.17	5.38	3.72	4.21	2.75
Stage4	91.31	78.50	85.11	83.17	88.99
Exec. Time	0.302	0.122	0.176	0.156	0.239

N=1048576	Resampling Methods				
	M	C1-128	C1-2048	C2-128	C2-2048
Stage1	2.74	13.31	8.98	10.21	6.01
Stage2	0.43	2.11	1.42	1.62	0.95
Stage3	0.85	4.12	2.78	3.16	1.86
Stage4	95.97	80.46	86.81	85.01	91.17
Exec. Time	2.215	0.456	0.676	0.595	1.010

N=4194304	Resampling Methods				
	M	C1-128	C1-2048	C2-128	C2-2048
Stage1	1.96	13.37	8.93	9.85	5.28
Stage2	0.25	1.73	1.16	1.28	0.68
Stage3	0.53	3.64	2.43	2.68	1.44
Stage4	97.25	81.26	87.48	86.19	92.60
Exec. Time	12.158	1.781	2.665	2.417	4.510

Table 5.2: RMSE Results.

Resampling Methods	The Number of Particles				
	16384	65536	262144	1048576	4194304
M	4.65801	4.65806	4.65911	4.65931	4.65894
C1-128	4.67725	4.67661	4.67649	4.67629	4.67659
C1-2048	4.66174	4.65886	4.65927	4.65919	4.65926
C2-128	4.66012	4.65909	4.65919	4.65912	4.65904
C2-2048	4.66008	4.65836	4.65901	4.65872	4.65909

5.5 Discussion on L1 Cache use

In Tesla K40, the L1 cache is used for local memory access such as register spills and stack data. Global operations leverage L2 cache, but it is not controllable by the programmer. However, one can use L1 cache for the global memory operations by setting the *-Xptxas -dlcm=ca* flag at compile time [27]. We use the *l1_cache_global_hit_rate* metric in CUDA profiler [29]. In our experiments we use the default setting, which is no L1 cache. Nevertheless, it is interesting to see how the proposed techniques could benefit from the use of L1 cache. As C1 and C2 memory access patterns tend to be more localized than those of M, we expect C1 and C2 benefit more from the use of L1 cache. The cache size of the L1 cache is 128 byte. When there occurs L1 cache miss, 128 byte from L2 cache or global memory is moved entirely. The cache hit ratio depends on some factors. Occupancy on SMX is one of the factor. In Tesla K40, at most 16 blocks or 64 warps or 2048 threads can be active at a time [36]. In our experiments, we consider these limitations to distribute the blocks among the SMXs evenly. The working set size is another factor on the cache hit ratio. This is the number of particles in our experiments. The other factor is the runtime scheduling of warps. The blocks are assigned to the SMXs and warps of these blocks are scheduled on the cores. When a block completes its job, the next block waiting in the queue becomes active. Simultaneous run of warps are dependent to the hardware resources. As long as enough resources exist, multiple warps can run at the same time. The last factor is the configuration between L1 cache and shared memory. They share 64 KB memory region. We can set the size of L1 cache as 16, 32 and 48 KB [36].

Table 5.3: L1 cache global loads hit ratio (percent) of the resampling algorithms.

Resampling	Cache	Number of Particles ($log_2 N$)								
Methods	Size	10	11	12	13	14	15	16	18	20
M	16 KB	99.72	99.80	99.85	51.01	26.29	13.61	7.07	1.80	0.45
M	32 KB	99.73	99.82	99.84	99.87	51.79	26.41	13.49	3.44	0.91
M	48 KB	99.73	99.81	99.85	99.87	77.52	39.65	20.20	5.15	1.36
C1-128	16 KB	95.64	96.36	96.61	96.62	96.75	96.82	96.80	96.81	96.80
C1-128	32 KB	94.98	96.13	96.44	96.55	96.77	96.82	96.82	96.85	96.84
C1-128	48 KB	95.54	96.32	96.63	96.60	96.75	96.79	96.84	96.86	96.85
C1-2048	16 KB	99.51	99.56	99.59	74.79	54.75	43.99	38.57	34.51	33.49
C1-2048	32 KB	99.48	99.57	99.61	99.63	92.66	85.11	80.25	76.18	75.09
C1-2048	48 KB	99.51	99.55	99.62	99.63	99.47	98.28	96.94	95.49	95.05
C2-128	16 KB	98.92	99.11	99.27	65.33	36.82	20.80	12.38	5.96	4.30
C2-128	32 KB	98.89	99.19	99.30	99.38	81.46	62.07	49.21	38.33	35.54
C2-128	48 KB	98.88	99.13	99.3	2 99.37	97.15	88.58	80.01	70.98	68.55
C2-2048	16 KB	99.54	99.64	99.71	53.34	27.42	13.85	7.04	1.83	0.51
C2-2048	32 KB	99.53	99.65	99.69	99.72	57.79	30.60	16.12	4.99	2.41
C2-2048	48 KB	99.49	99.64	99.71	99.74	85.90	51.70	30.05	11.33	7.20

We select a distribution that has the longest execution time among the ones we use. This is the distribution in (2.33) by setting the value of y to 4. For each cache configuration and each resampling algorithm, we create 16 different weight sequences for any N. The resampling algorithms draw 256 (the value of K) offspring sequences from each weight sequence. The results are the averages of these 16 weight sequences. We only measure the kernels of M, C1 and C2. The cache hit ratios of all resampling algorithms are given in Table 5.3. The values in the cells are the ratios of the cache hit in L1 cache for the global loads inside the CUDA kernel of the resampling algorithms. There are also two pieces of information given in the cells of the table. The numbers written normally mean that the algorithm runs faster compared to its no-cache setting. The underlined numbers mean that the algorithm runs slower compared to its no-cache setting.

In all algorithms, we gain speed by enabling L1 cache as long as the number of particles does not exceed 4096 when the cache size is 16 KB, 8192 when the cache

size is 32 KB, and 16384 when the cache size is 48 KB. In M, when the cache size is 16 KB, we can load at most 16KB/4Byte= 4096 single precision numbers. As long as the number of particles does not exceed 4096, M uses the L1 cache around 99.8%. As the number of particles is doubled, the probability of the finding the data in L1 cache at each iteration of the inner loop goes down to the half of the current ratio. This behavior is same when the cache size is 32 KB. As long as the number of particles does not exceed 8192, Metropolis uses the L1 cache around 99.8%. This is a bit different when the cache size is 48 KB. Because 48 is not power of 2 and we can load at most 12288 single precision numbers. When the number of particles is 16384, the cache hit ratio is around 75%. As the number of particles is doubled, the probability of the finding the data in L1 cache at each iteration of the inner loop goes down to the half of the current ratio.

In C1, when the s-segment size is 128, we gain speed in all experiments. This is because the L1 cache is large enough to store all segment data in all configurations. When the s-segment size increases to 2048, we still gain speed as the cache hit ratio is around 74% or higher for 16 KB cache configuration, and 85% or higher for 32 KB cache configuration. For 48 KB cache configuration, we gain speed at any number of particles. In C2, when the s-segment size is 128, we gain speed although the cache hit ratio is around 38%. When the s-segment size is 2048, we gain speed as the cache hit ratio is around 85% or higher. Main factor of slowdown of the algorithms is cache miss ratio. This suggests a strategy of bypassing the L1 cache as the cache miss ratio increases. The execution time results of the algorithms are given in Table 5.4 and the speed up results of the algorithms are given in Table 5.5. There are also two pieces of information given in the cells of the table. The numbers written normally mean that the algorithm runs faster compared to its no-cache setting. The underlined numbers mean that the algorithm runs slower compared to its no-cache setting.

When we examine the speed up results of C1 with $SS = 128$, it is seen that it achieves faster execution times along with higher speed-up in most experiments. But in some of the experiments the speed-up is lower than its non-cache version, even if C1 gains faster execution times. This shows us that M benefits more from the cache than C1 in some of the configurations, especially, when the number of particles is small. These are valid for the results of C1 with $SS = 2048$. In addition, the execution time of C1

Table 5.4: Execution times (in milliseconds) of the resampling algorithms.

Resampling	Cache	Number of Particles ($log_2 N$)								
Methods	Size	10	11	12	13	14	15	16	18	20
M	NO	0.36	0.48	0.88	1.99	3.39	6.17	10.5	37.8	305
M	16 KB	0.31	0.44	0.57	2.24	5.58	9.96	18.0	68.2	317
M	32 KB	0.30	0.41	0.56	0.94	3.86	8.33	16.6	67.4	308
M	48 KB	0.35	0.42	0.56	0.97	2.77	6.95	15.4	65.8	307
C1-128	NO	0.26	0.29	0.34	0.65	0.99	1.61	2.89	10.9	43.1
C1-128	16 KB	0.22	0.26	0.29	0.53	0.77	1.27	2.26	8.52	33.4
C1-128	32 KB	0.23	0.24	0.29	0.52	0.78	1.27	2.27	8.51	33.3
C1-128	48 KB	0.22	0.25	0.29	0.51	0.77	1.28	2.27	8.53	33.4
C1-2048	NO	0.34	0.39	0.50	1.00	1.48	2.47	4.43	17.0	67.0
C1-2048	16 KB	0.30	0.35	0.42	0.96	1.71	3.03	5.66	21.9	87.0
C1-2048	32 KB	0.29	0.34	0.42	0.78	1.31	2.42	4.56	17.7	70.0
C1-2048	48 KB	0.28	0.34	0.40	0.78	1.16	2.01	3.73	14.4	57.3
C2-128	NO	0.35	0.42	0.47	1.00	1.55	2.64	4.77	18.1	73.4
C2-128	16 KB	0.31	0.36	0.41	0.99	1.61	2.84	5.27	20.8	85.0
C2-128	32 KB	0.33	0.38	0.42	0.82	1.43	2.47	4.53	17.7	73.9
C2-128	48 KB	0.32	0.34	0.44	0.84	1.31	2.33	4.27	16.8	70.3
C2-2048	NO	0.45	0.58	0.70	1.42	2.20	3.70	6.74	26.0	114
C2-2048	16 KB	0.42	0.46	0.55	1.66	3.09	5.87	11.7	47.0	189
C2-2048	32 KB	0.38	0.47	0.54	1.14	2.30	4.86	10.6	46.3	187
C2-2048	48 KB	0.38	0.51	0.54	1.13	1.98	3.85	9.11	43.5	178

Table 5.5: Speed up of the resampling algorithms. The values are obtained by dividing the execution time of M to the corresponding execution time of C1 or C2.

Resampling	Cache	Number of Particles ($log_2 N$)								
Methods	Size	10	11	12	13	14	15	16	18	20
C1-128	NO	1.38	1.66	2.59	3.06	3.42	3.83	3.63	3.47	7.08
C1-128	16 KB	1.41	1.69	1.97	4.23	7.25	7.84	7.96	8.00	9.49
C1-128	32 KB	1.30	1.71	1.93	1.81	4.95	6.56	7.31	7.92	9.25
C1-128	48 KB	1.59	1.68	1.93	1.90	3.60	5.43	6.78	7.71	9.19
C1-2048	NO	1.06	1.23	1.76	1.99	2.29	2.50	2.37	2.22	4.55
C1-2048	16 KB	1.03	1.26	1.36	2.33	3.26	3.29	3.18	3.11	3.64
C1-2048	32 KB	1.03	1.21	1.33	1.21	2.95	3.44	3.64	3.81	4.40
C1-2048	48 KB	1.25	1.24	1.40	1.24	2.39	3.46	4.13	4.57	5.36
C2-128	NO	1.03	1.14	1.87	1.99	2.19	2.34	2.20	2.09	4.16
C2-128	16 KB	1.00	1.22	1.39	2.26	3.47	3.51	3.42	3.28	3.73
C2-128	32 KB	0.91	1.08	1.33	1.15	2.70	3.37	3.66	3.81	4.17
C2-128	48 KB	1.09	1.24	1.27	1.15	2.11	2.98	3.61	3.92	4.37
C2-2048	NO	0.80	0.83	1.26	1.40	1.54	1.67	1.56	1.45	2.68
C2-2048	16 KB	0.74	0.96	1.04	1.35	1.81	1.70	1.54	1.45	1.68
C2-2048	32 KB	0.79	0.87	1.04	0.82	1.68	1.71	1.57	1.46	1.65
C2-2048	48 KB	0.92	0.82	1.04	0.86	1.40	1.81	1.69	1.51	1.72

is larger than its non-cache version, even if the speed-up is higher than its non-cache version. In this situation, Metropolis is more affected adversely from the cache size than C1 with $SS = 2048$. These situations are also valid for C2.

CHAPTER 6

UPHILL RESAMPLING METHOD AND ITS VARIATIONS

In this chapter, we present our proposed resampling method named Uphill resampling. We show how we eliminate the numerical instability problem with the Uphill resampling. We compare bias, MSE and execution time results of Uphill with the results of Systematic, Metropolis and Rejection. We also present coalesced version of Uphill named Uphill-CA. We compare them in terms of bias, MSE and execution time. We discuss the importance of Uphill-CA both in theory and practice. We also present the generic version of Uphill named Uphill-C1. We compare them in terms of bias, MSE and execution time. Then we compare all resampling algorithms on a highly non-linear example. And we show how we benefit from the coalesced variants of the Uphill resampling. There is also an experiment about the speed up of the GPU implementation of SIR particle filter over the CPU implementation of it. Moreover, we discuss the global memory load transactions and SMX efficiency of the Uphill resampling algorithm and its variations on the profiler results. Finally, we discuss the factors on the execution times of Uphill and its variations.

6.1 Uphill Resampling

We propose a new resampling method, designated Uphill resampling. It only compares the weights of two particles. Similar to the Metropolis and Rejection resampling, it does not suffer from numerical instability as it does not need cumulative sum of the weights. It is suitable to implement on the GPU efficiently. The Uphill resampling can run with a single CUDA kernel. However, it suffers from the non-coalesced

global memory access problem like Metropolis and Rejection. The pseudo-code for the Uphill algorithm is given in Algorithm 6.1.

Algorithm 6.1 Uphill Resampling

1: **procedure** $[\{x^i_{new}\}_{i=1}^N]$ = UPHILL($\{x^i, \tilde{w}^i\}_{i=1}^N, B$)
2: foreach $i = 1 : N$
3: $t = i$
4: for $m = 1 : B$
5: $j \sim v\{1, .., N\}$
6: if $\tilde{w}^t < \tilde{w}^j$ then $t = j$ end if
7: end for
8: $x^i_{new} = x^t$
9: end foreach

In this algorithm, the inner loop bound B is indeed the number of candidate particles to be tried for being replicated for the next time step. The other parameters are as in Algorithm 2.3 and Algorithm 3.3. The 'foreach' loop can be executed in parallel among the threads where a dedicated thread runs for each particle. Each thread selects an ancestor for the particle it represents. The ancestor of the ith particle is selected from the set of $B + 1$ particles (B randomly selected particles plus the ith particle). The particle with the largest weight is chosen as the ancestor. If there is more than one particle with the largest weight, the one selected earliest is kept.

The ordering of the weights in the weight array is a determiner in selection process. The B parameter and the number of particles N have an effect on the selection process. Larger B increases the probability of the selection of the particles with larger weights. We can compute the expected number of replications of the particle which is in the ith position in the weight array, assuming that the weight array is ordered, with the following formula:

$$EU(i, B) = \frac{i^{B+1} - (i-1)^{B+1}}{N^B} \tag{6.1}$$

$EU(i, B)/N$ is the probability of selection of the ith particle. The details of the analysis are given in the Appendix A.1.

6.2 Uphill-CA Resampling

Uphill suffers from non-coalesced global memory access pattern like Metropolis and Rejection. We propose a memory coalesced variant of the Uphill resampling, designated Uphill-CA (abbreviated UCA), to ameliorate this problem.

We use s-segment concept which is introduced in [9] and Section 5.1. An s-segment is a collection of contiguous segments, so that the smallest s-segment is a single segment and the largest s-segment is the one that includes all segments. The sizes of s-segments used in the algorithms in the same experiment are always equal. In this technique, each warp selects an s-segment at each iteration of the 'for' loop. Each thread in the same warp selects a random particle within the selected s-segment. The pseudo-code for Uphill-CA is given in Algorithm 6.2.

Algorithm 6.2 Uphill-CA Resampling

1: **procedure** $[\{x^i_{new}\}_{i=1}^N]$ = UPHILL-CA($\{x^i, \tilde{w}^i\}_{i=1}^N, B, SC, DC$)
2: foreach $i = 1 : N$
3: $t = i$
4: for $m = 1 : B$
5: $s \sim v\{1, .., SC\}$
6: $j \sim v\{(s-1) * DC + 1, .., s * DC\}$
7: if $\tilde{w}^t < \tilde{w}^j$ then $t = j$ end if
8: end for
9: $x^i_{new} = x^t$
10: end foreach

SC is the number of s-segments; its value is $4N/SS$ where SS denotes the size of an s-segment in bytes. The value of N must be a power of 2. DC is the number of weights in an s-segment; its value is $SS/4$. Recall that a single-precision floating-point number occupies 4 bytes in CUDA. The j is a random integer between the indexes of the first and the last elements of the s-segment drawn from a uniform distribution. The s is the index of the selected s-segment for a warp, drawn from a uniform distribution. All the threads in a warp must have the same value of s; this can be ensured by using the index of the warp as the sequence of the random number generator.

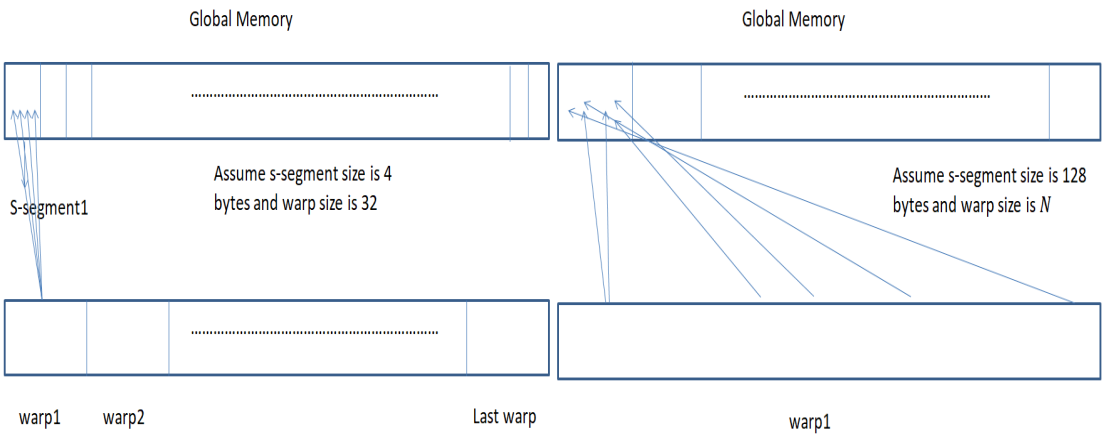

Figure 6.1: Examples of variances caused by s-segment size and warp size.

We analyze the Uphill-CA resampling algorithm. The expected number of replications of the particle which is in the ith position of the weight array, assumed ordered, is given below.

$$EUCA(i,B) = \frac{1}{(SCDC)^B}(S^B + (i-1)(S^B - (S-1)^B)) \tag{6.2}$$

where $S = \sum_{k=1}^{SC} p_{ik}$ and p_{ik} is the position of the largest element whose weight is less than or equal to $\tilde{w}^i$ in s-segment k. Since $\sum_{k=1}^{SC} p_{ik} = i$ and $SCDC = N$, the analysis is exactly same as the analysis of the Uphill resampling method. But there occurs variance in the results depending on the s-segment size and warp size. Two examples about this variance are given in Figure 6.1. It is seen that the probability of drawing a weight is equal to $\frac{1}{SCDC}$ where $SCDC = N$. But all the threads in the warp draw the weights from the same s-segment, that is why the variances are occurred in the experiments. The details of the analysis are given in the Appendix A.2 and A.3.

6.3 Uphill-C1 Resampling

We also propose a generic version of the Uphill resampling called Uphill-C1 (abbreviated UC1). We use the same s-segment concept as in the Uphill-CA resampling. The stages of the algorithm are given in Algorithm 6.3.

Algorithm 6.3 Uphill-C1 Resampling

1: **procedure** $[\{x^i_{new}\}_{i=1}^N]$ = UPHILL-C1($\{x^i, \tilde{w}^i\}_{i=1}^N, B, SC, DC$)
2: foreach $i = 1 : N$
3: $t = i$
4: $s \sim v\{1, .., SC\}$
5: for $m = 1 : B$
6: $j \sim v\{(s-1) * DC + 1, .., s * DC\}$
7: if $\tilde{w}^t < \tilde{w}^j$ then $t = j$ end if
8: end for
9: $x^i_{new} = x^t$
10: end foreach

The parameters are as in the Algorithm 6.2. The only difference from Uphill-CA is the selection of an s-segment. In Uphill-C1, each warp selects a random s-segment only once at the beginning. So this method approaches as a local way. The selection of s for the threads in a warp can be done as in Uphill-CA.

We analyze the Uphill-C1 resampling algorithm. The expected number of replications of the particle which is in the ith position of the weight array, assumed ordered, is given below. The details of the analysis are given in Appendix A.4 and A.5.

$$EUC1(i, B) = \frac{1}{SC}\frac{1}{DC^B}\left(\sum_{k=1}^{SC} p_{ik}^B + (i-1)(c_i^B - (c_i - 1)^B)\right) \tag{6.3}$$

where p_{ik} is the position of the largest element whose weight is less than or equal to $\tilde{w}^i$ in s-segment k. And c_i is the position of the ith particle in its own s-segment. Note that as the s-segment size, SS, is selected larger the Uphill-C1 algorithm becomes more similar to the original Uphill algorithm, and for the limiting case of $SS = 4N$ it becomes behaviorally same as the Uphill resampling. The expectation of the algorithm differs as the s-segment size differs which makes Uphill-C1 a new resampling algorithm in theory. The Uphill resampling is one of the new resampling algorithm of the Uphill-C1 resampling. To give another examples, when $SC = N$ and warp size is equal to 1, there is only one weight in each warp and each thread selects single weight. They choose the selected weight as their ancestor if it is greater than their

own weight. When $SC = N$ and warp size is equal to N, there is only one warp and this warp selects a single weight. The threads in the warp compare their weights with this selected weight. The particles whose weight is greater than this selected weight replicate only once and the remaining ones consist of the particle of this selected weight. Note that, in both scenarios setting B as 1 is enough.

6.4 Finding Optimum B

Ideally, we would choose the optimum B for the Uphill and Uphill-CA resampling by minimizing the target function below:

$$\min_{0 \leq B} \sum_{i=1}^{N} \left(EU(i, B) - \frac{N\tilde{w}^i}{s\_\tilde{w}} \right)^2 \tag{6.4}$$

where $EU(i, B)$ is the expected number of replications of the Uphill resampling for the particle in the ith position of the weight array and $\tilde{w}^i$ is the weight of that particle. This target function can also be considered as the minimum expected square bias with respect to B. Implementation of this target function requires sorting on the weight set and many reduction operations because of the pairwise operations on the target function. This causes high computational cost which slows down the Uphill methods. We forgo pairwise operations, and use the metric SSD below which is computed by Algorithm 6.4.

$$SSD(P) = \sum_{i=1}^{N} \left(\frac{NP^i}{s\_P} - 1 \right)^2 \tag{6.5}$$

It is seen that when the members of P are all equal, SSD is zero. Further, the value of SSD increases monotonically as the relative variance in the distribution increases. Note that P is a sequence.

In the first stage, the value of all members of P are summed up. In the second stage, to form a summand, we subtract 1, which is the mean of the total expected replications of each member of P, from the expected number of replications of the members of

Algorithm 6.4 SSD

procedure $[value]$ = SSD $(\{P^i\}_{i=1}^N)$

1: $s\_P = SUM[\{P^i\}_{i=1}^N]$

2: $value = SUM[\{(\frac{NP^i}{s\_P} - 1)^2\}_{i=1}^N]$

P and square the difference. Then we sum the partial results to get the overall SSD value of the P.

First, we apply Algorithm 6.4 to the $EU(:, B)$ for each B from 0 to 8191 (a sufficiently large number for B) only once and save the results in a file. In the experiments, we load them from the file and save them to the global memory of the GPU as an array. Then we apply Algorithm 6.4 to the current weight sequence to obtain its SSD. To compute SSD we do not need to sort the weight sequence and we only perform two reductions with logarithmic time complexity. With $4,194,304$ particles, this metric is about 75 times faster than (6.4). In some weight distributions, the values of B are slightly different than the values of B in (6.4). But this does not harm Uphill performance substantially.

We search SSD of the weight sequence on the array to find a location where adding it does not violate monotonous order of the array. The pseudo-code of this process is given in Algorithm 6.5. Note that a single 'if-else if' instruction is run by each warp.

Algorithm 6.5 Find B

1: **procedure** $[B]$ = FIND-B $(array, value)$

2: foreach $i = 1 : N$

3: if $i == 1$ && $value \leq array^1$ then

4: $B = 0$

5: else if $value > array^{i-1}$ && $value \leq array^i$ then

6: $B = i - 1$

7: end if

8: end foreach

6.5 Quality, Execution Time Measures and Some Implementation Issues

To assess the quality of the resampling algorithms, we use the metrics and distributions given in Section 2.5. In any experiment, we choose the number of particles N as a power of 2 between 2^4 and 2^{22}. We create 16 different weight sequences from each distribution for any N. Each resampling algorithm draws 256 (the value of K) offspring sequences from each weight sequence. The results are the averages of these 16 weight sequences. The output of the algorithms is the ancestor array (rather than the states of particles). S-segment size is at most $4N$.

To assess the execution times, in terms of seconds, we measure the main kernels of the resampling algorithms along with necessary kernels to calculate a) B in Metropolis and Uphill, b) maximum weight in Rejection and c) cumulative sum of the weights and cumulative offspring to ancestor in the Systematic resampling. The speed up of the Uphill-CA or Uphill-C1 resampling is the ratio of the execution time of Uphill over the execution time of Uphill-CA or Uphill-C1 on the same input.

We use the formula in [22] to calculate the value of B in the Metropolis resampling. Therein we choose ϵ as $1/100$ and β as $\bar{\tilde{w}}/\tilde{w}_{max}$ where $\bar{\tilde{w}}$ is the mean of the weights and $\tilde{w}_{max}$ is the maximum weight. $\tilde{w}_{max}$ is calculated in the Rejection resampling as well. We use the SSD metric, defined in (6.5) and computed by Algorithm 6.4, to calculate the B in the Uphill resampling. We use prefix sum to calculate the cumulative sum of the weights in the Systematic resampling. For the sum, mean and maximum operations on the weight array, we use reduction techniques. We use an $O(logN)$ parallel reduction algorithm in the implementation of reduction [11], and an $O(logN)$ parallel scan algorithm for prefix sum operation [16]. In the Systematic resampling, we call a simple kernel that consists of one thread to obtain uniform number on the GPU.

6.6 Bias, MSE and Execution Time Results of Resampling Methods

In this section, we compare bias, MSE and execution time results of the Systematic, Metropolis, Rejection and Uphill resampling methods. We reproduce the results of

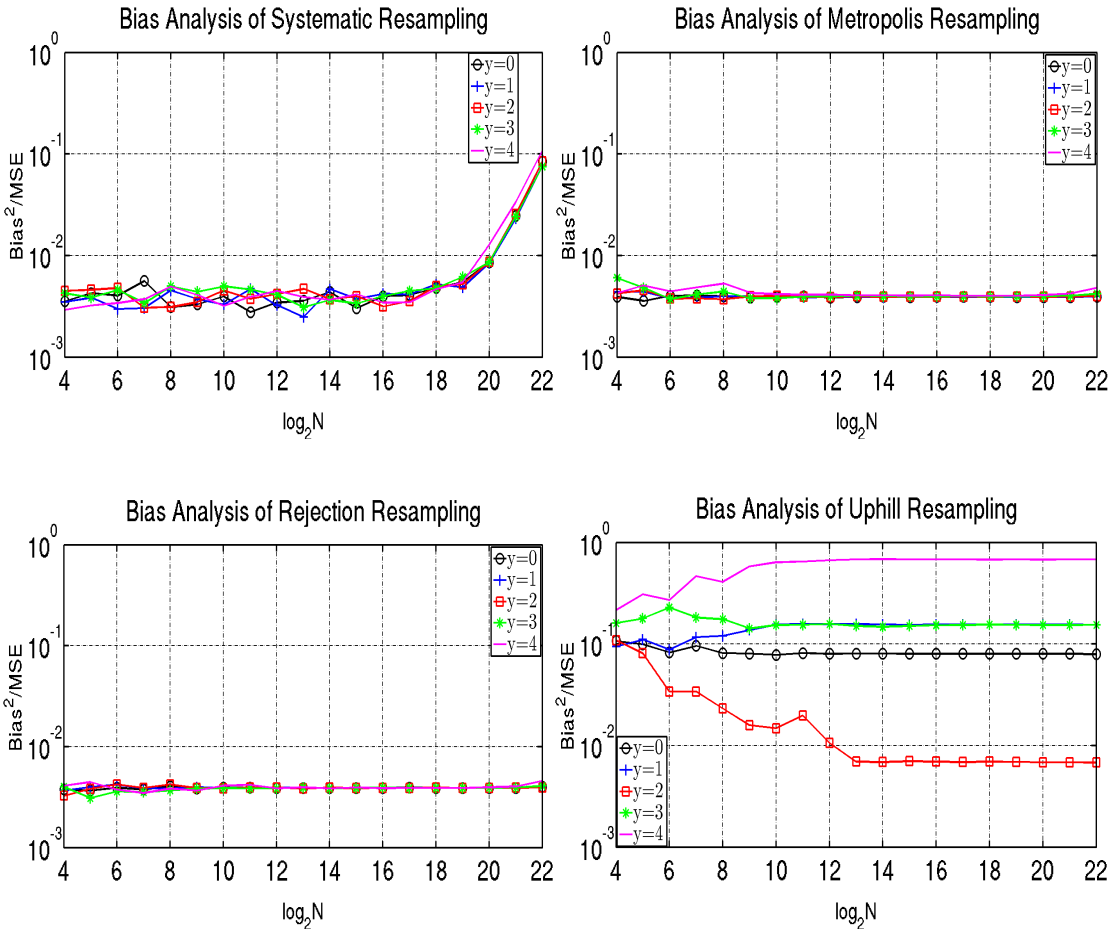

Figure 6.2: Bias results of the Systematic, Metropolis, Rejection and Uphill resampling methods on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

Systematic, Metropolis and Rejection in [22]. Furthermore, we compare the quality and execution time of the Metropolis and Uphill resampling by setting the value of B same in both algorithms. Figure 6.2 shows the bias, Figure 6.3 shows the MSE and Figure 6.4 shows the execution time results of the Systematic, Metropolis, Rejection and Uphill resampling algorithms on the distributions in (2.33).

It is seen that the bias results of the Systematic resampling are affected by numerical instability when the number of particles exceeds 2^{18}. Metropolis and Rejection do not suffer from the numerical instability problem since they use the ratio of weights. Uphill does not suffer from the numerical instability problem, too, since it performs pairwise comparison on the weights. However, the bias contribution, i.e. the ratio of the squared bias to the MSE, is higher than that in the other resampling methods

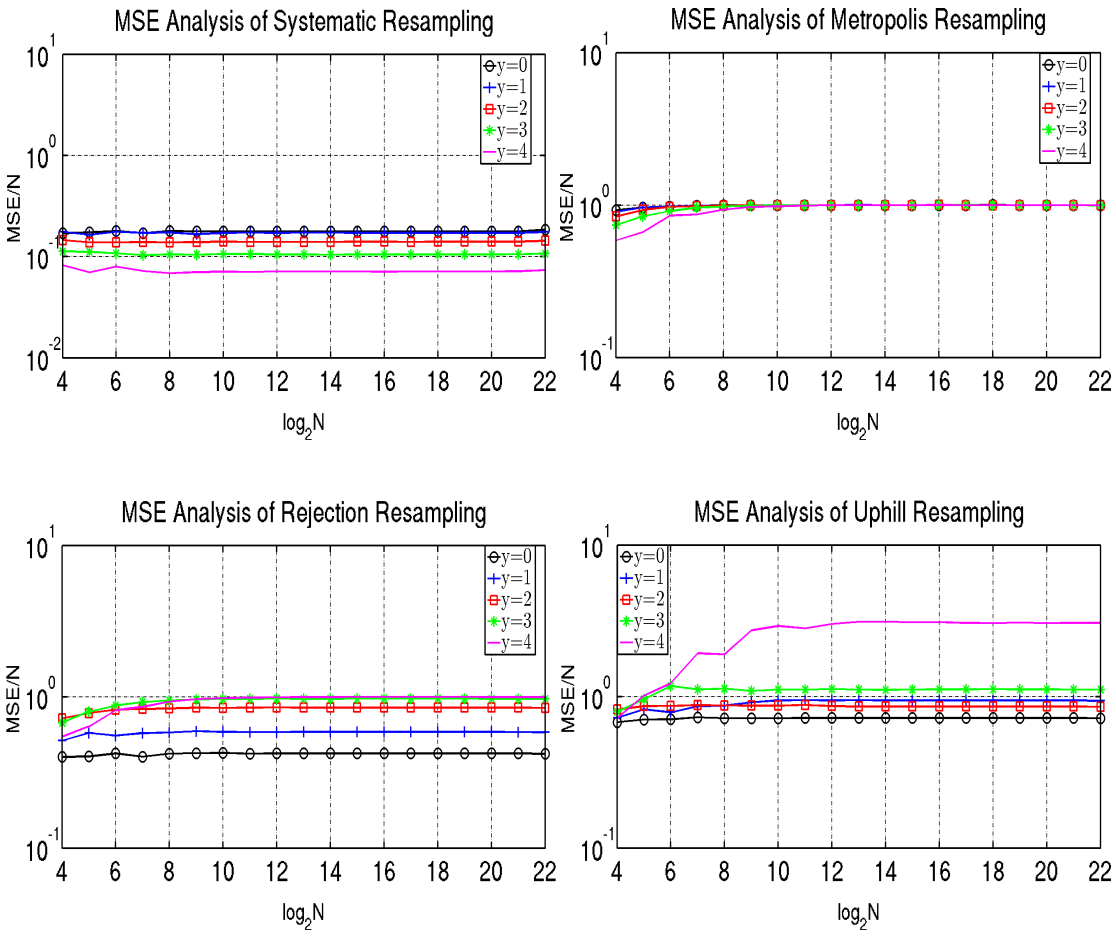

Figure 6.3: MSE results of the Systematic, Metropolis, Rejection and Uphill resampling methods on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

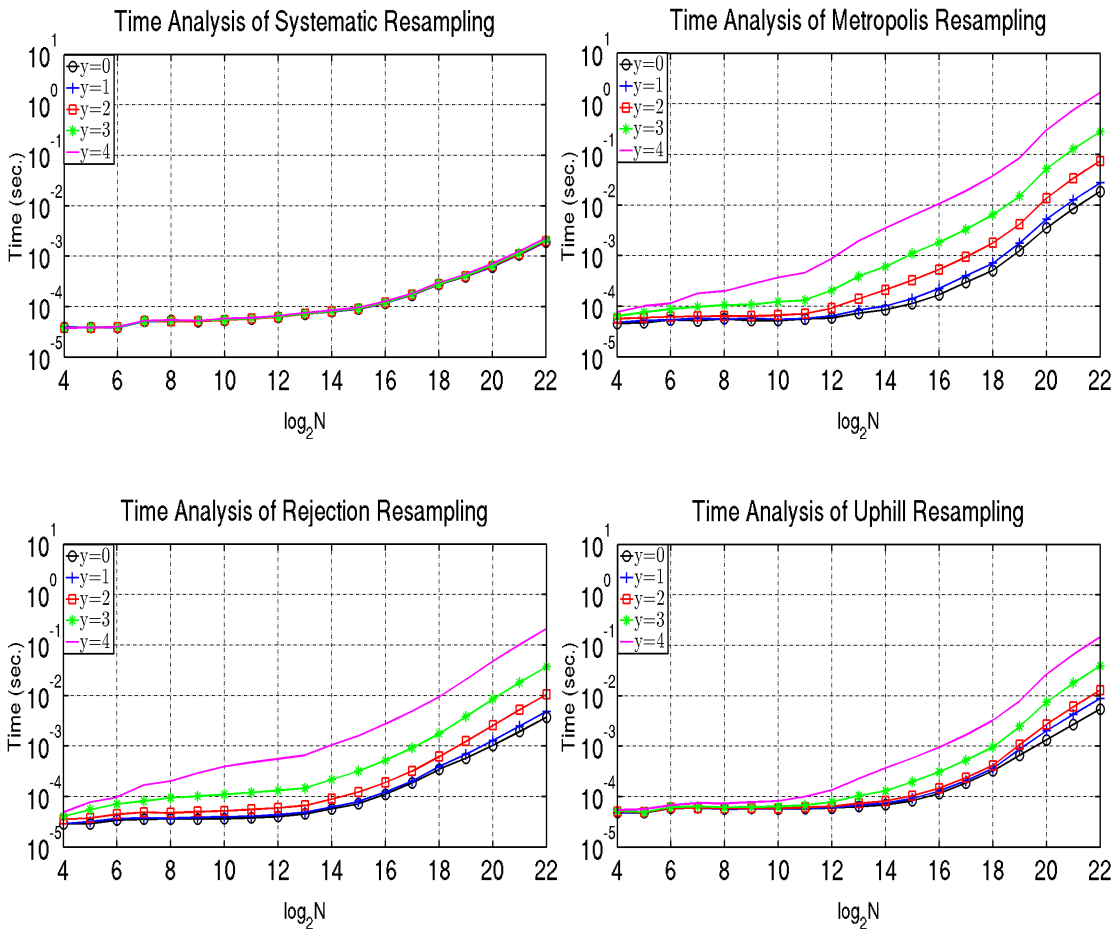

Figure 6.4: Execution time results of the Systematic, Metropolis, Rejection and Uphill resampling methods on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

because Uphill considers the quantity of the weights for order comparison. The Systematic resampling has the best MSE results, since it uses stratification over the cumulative sum of the weights. Rejection achieves better results in MSE and execution times than Metropolis does. The Uphill resampling achieves better results in MSE than Metropolis does when the relative weight variance within the weight sequence is small, and achieves better results in execution time than Metropolis does. Uphill achieves good MSE results even though its squared bias contributions are larger than the others. This is because the variance in the results of the Uphill resampling is smaller than those in the Metropolis and Rejection resampling. Random number generators also cause variance on the results. Uphill uses only one random number generator while Metropolis and Rejection use two.

The Systematic resampling has better execution time results since it is not affected from the relative variance within the weight sequence. The Uphill resampling is less affected from it compared to Metropolis and Rejection. Uphill has faster execution times than the Rejection resampling as the relative variance in the weights increases and has faster execution times than the Metropolis resampling as either the number of particles or relative variance in the weights is large.

We also investigate how Uphill is sensitive to the change in the value of y. We run the Uphill resampling on the distributions in (2.33) by choosing the value of y in the range $[3.5 - 4.5]$ with increments of 0.1. The bias, MSE and execution time results are given in Figure 6.5.

It is seen that the values of bias, MSE and execution time increases monotonically as the value of y increases when the number of particles is sufficiently large.

We compare the experimental results of the Metropolis resampling and Uphill resampling by setting the values of B of the Metropolis resampling as the values of B of the Uphill resampling. The bias, variance and MSE values of both resampling methods on the distributions in (2.33) are shown in Figure 6.6.

The bias of the Uphill resampling is smaller than that of Metropolis when the relative variance in the weights is large. This causes the MSE of Uphill to be better than those of Metropolis. When the relative variance in the weights is small, the bias of Metropo-

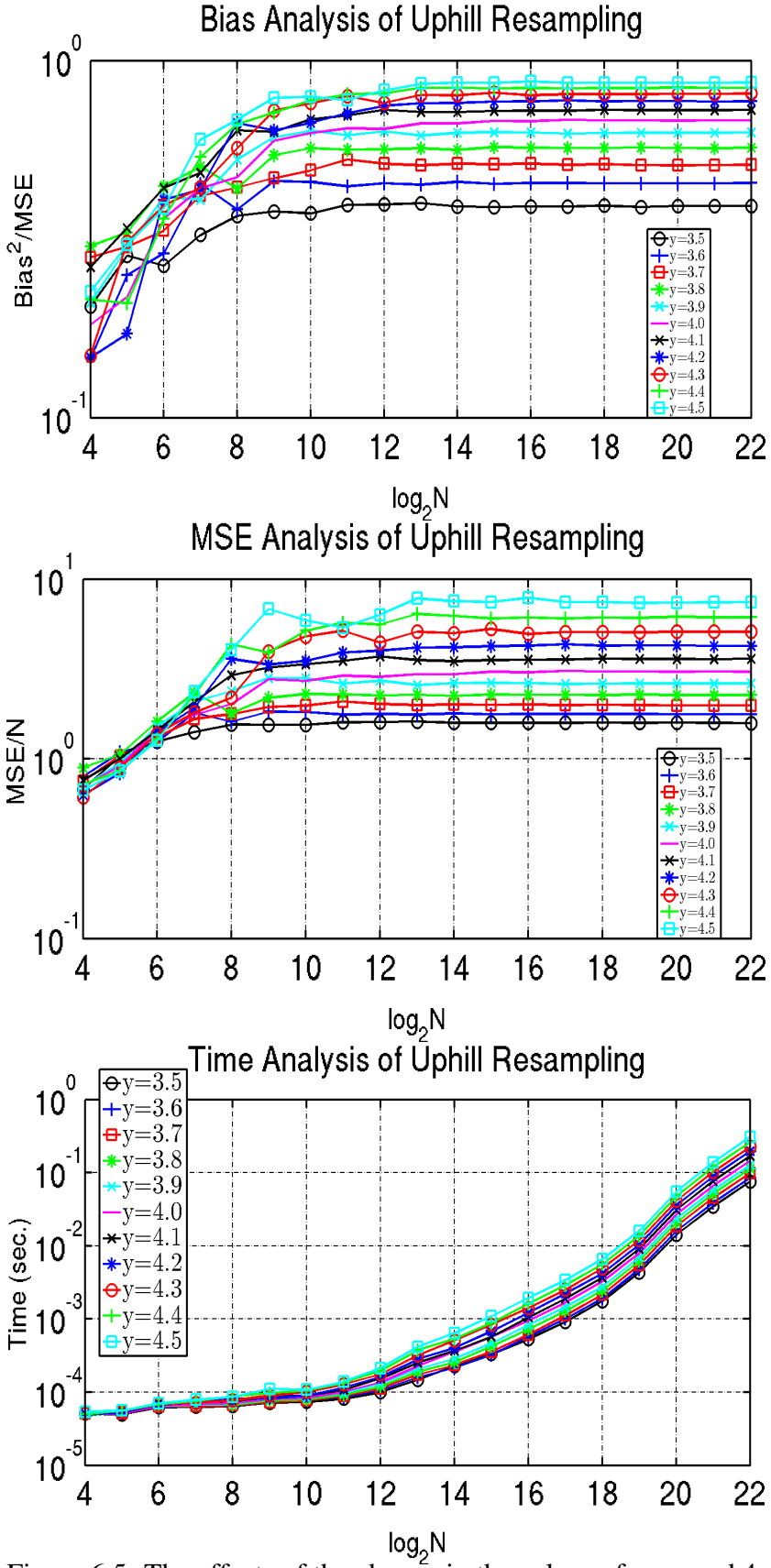

Figure 6.5: The effects of the change in the values of y around 4.

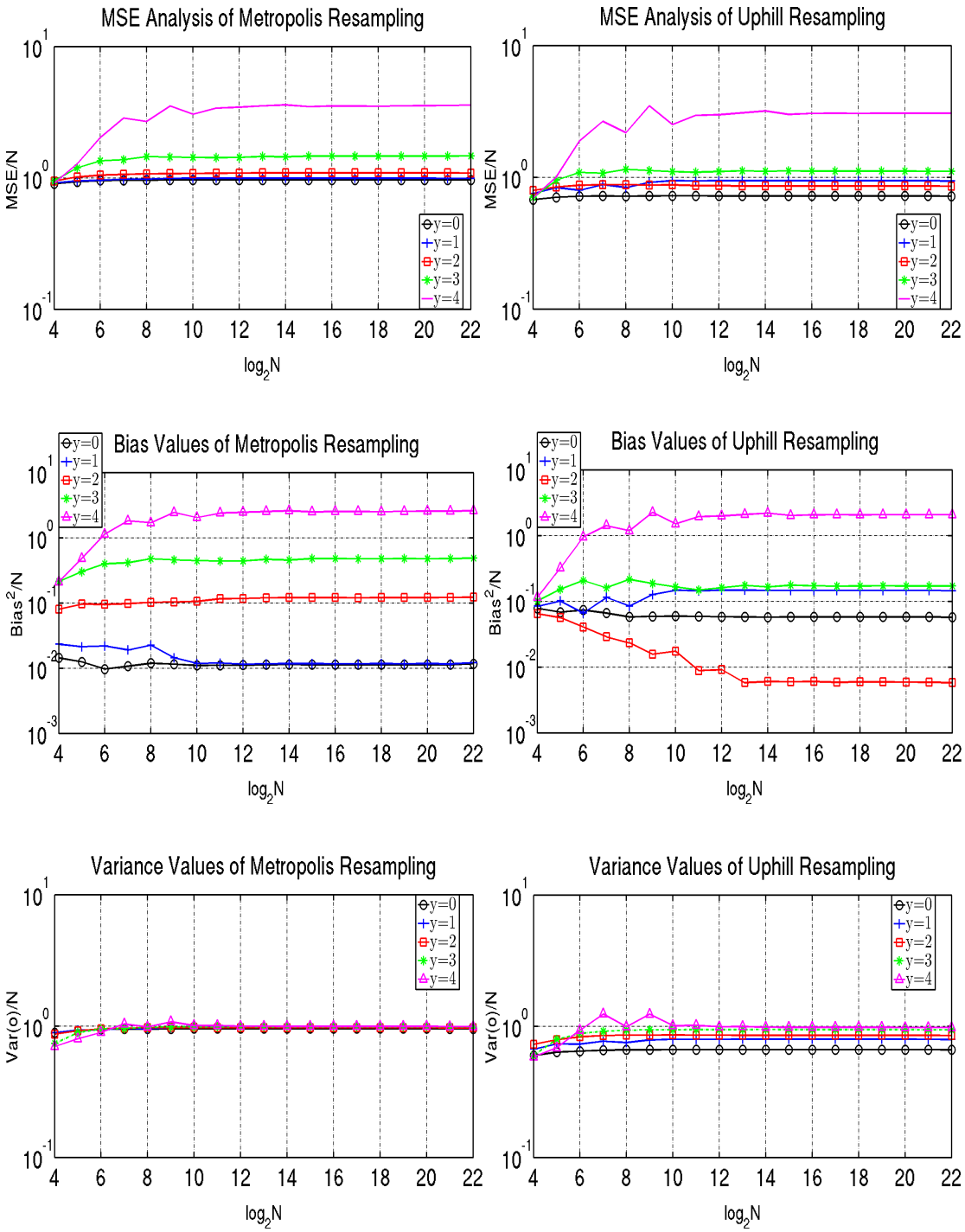

Figure 6.6: Bias, variance and MSE values of the Metropolis and Uphill resampling methods on the distributions in (2.33). The values of B in both algorithms are the same. The x-axis represents the number of particles (in logarithmic scale).

Table 6.1: The summary of the operations of Uphill and Metropolis.

Resampling Method	Random Number	Comparison	Division
Metropolis	Two	One	One
Uphill	One	One	-

lis is better than those of Uphill, but its MSE is worse. This is because of the variances in the results of Uphill to be smaller than those of Metropolis. The variance in the results of the Uphill resampling is smaller than that of Metropolis because Uphill generates a single random number in the body of the 'for' loop whereas Metropolis generates two. Furthermore, Uphill is slightly faster than Metropolis when the values of B are same in both because of the lesser number of basic operations performed by Uphill in the inner loop. The summary of the operations of Uphill and Metropolis are given Table 6.1.

The bias, MSE and execution time results of these four resampling algorithms on the generated gamma distributions are given in Figure 6.7, Figure 6.8 and Figure 6.9 respectively.

The four algorithms; Systematic, Metropolis, Rejection and Uphill, behave similarly to the previous experiment in terms of bias, except for Uphill with two distributions. The Systematic resampling is affected from the numerical instability problem in this experiment as well. In terms of MSE, Systematic has better results than those of the other resampling algorithms. The Uphill resampling is slightly better than Rejection in some cases and slightly better than Metropolis in most cases. The Systematic resampling has better execution times than those of the other methods. Uphill is faster than both Metropolis and Rejection in most cases.

We also compare the experimental results of Metropolis and Uphill on the gamma distributions by setting their B parameters as the value of B of Uphill. The bias, variance and MSE values of both resampling methods are given in Figure 6.10. Even though the bias of Metropolis is better than those of Uphill in most cases, Uphill is slightly better than Metropolis in MSE due to lower variance in the results.

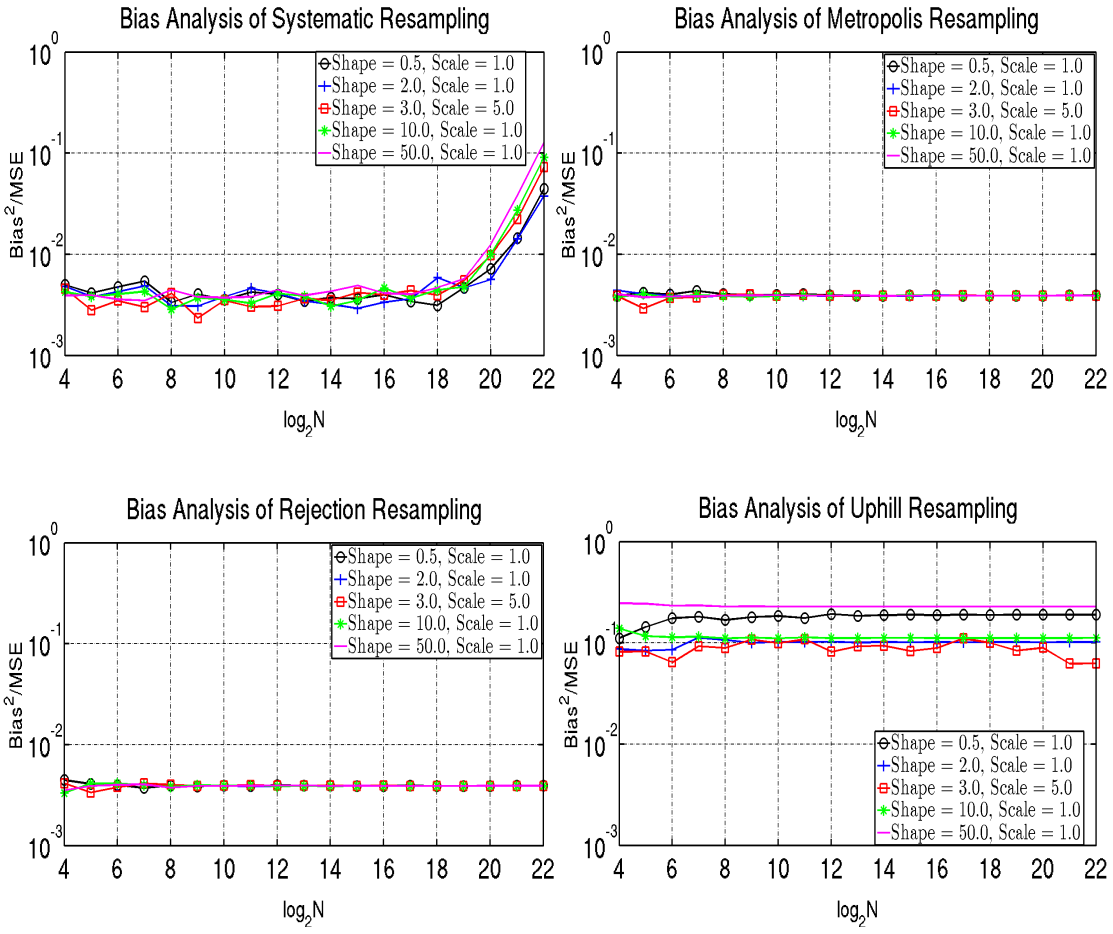

Figure 6.7: Bias results of the Systematic, Metropolis, Rejection and Uphill resampling methods on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

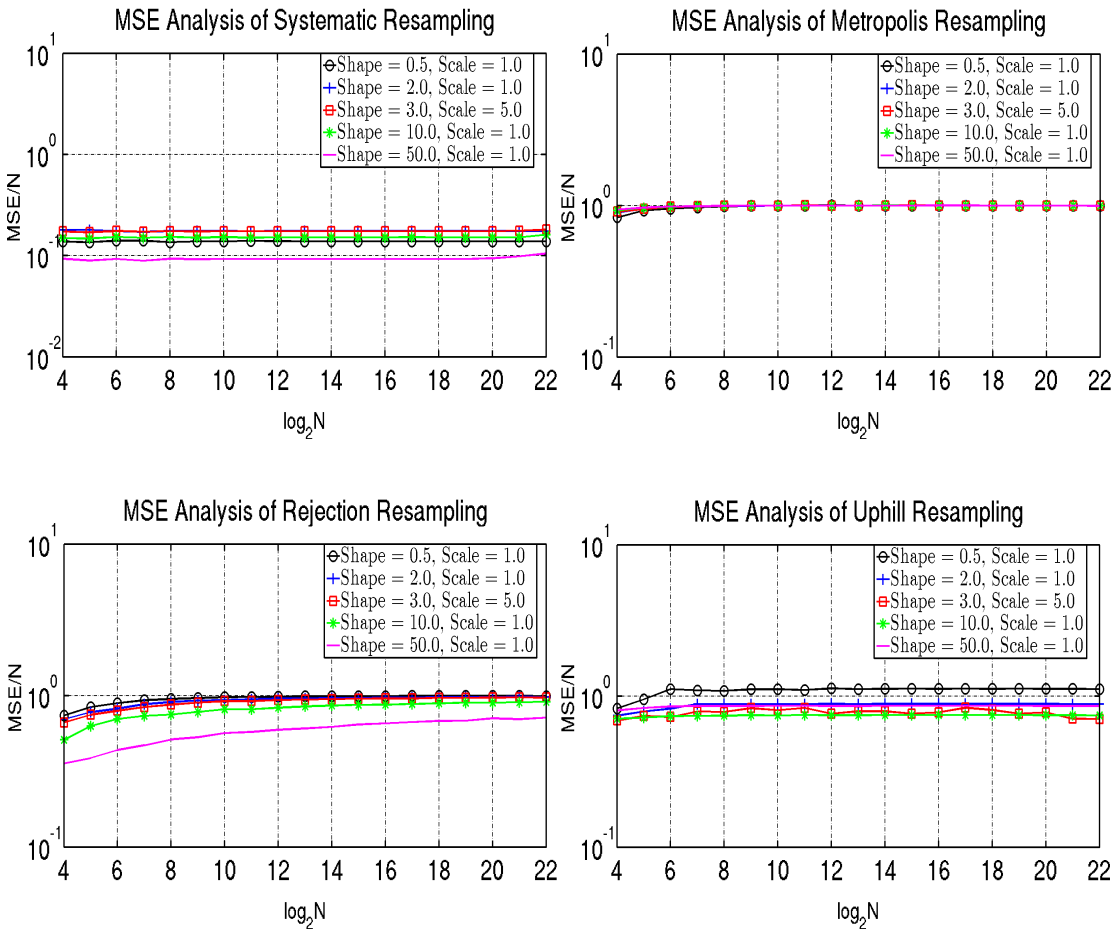

Figure 6.8: MSE results of the Systematic, Metropolis, Rejection and Uphill resampling methods on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

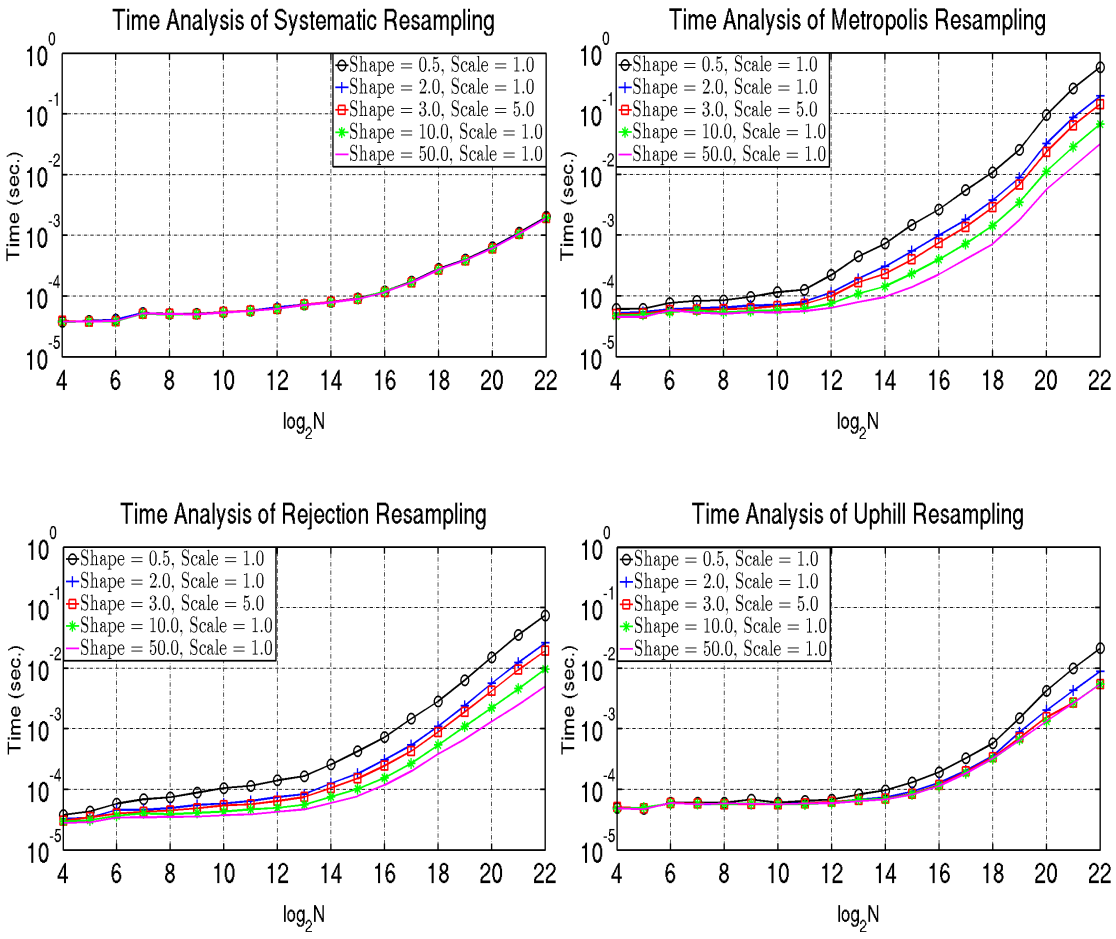

Figure 6.9: Execution time results of the Systematic, Metropolis, Rejection and Uphill resampling methods on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

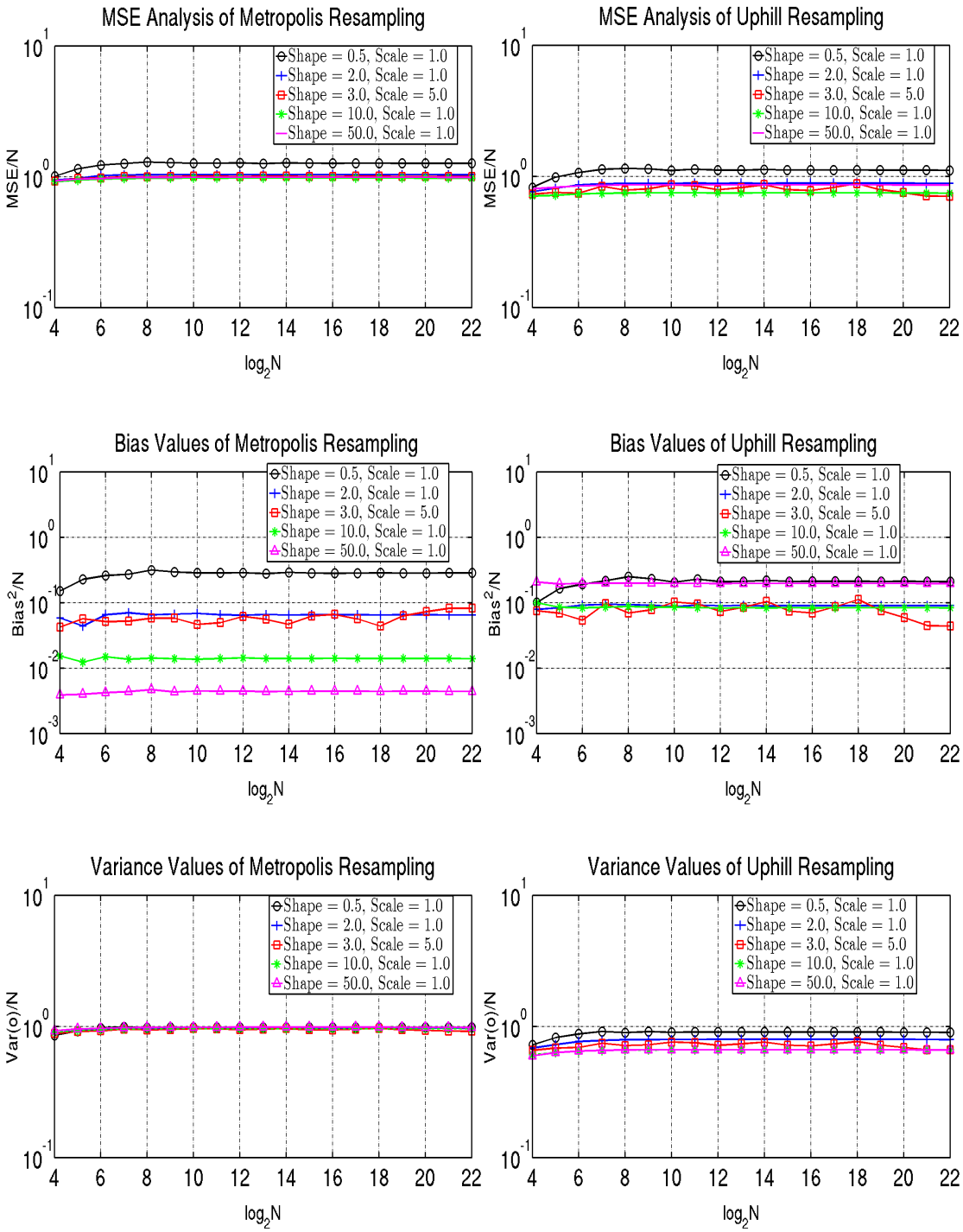

Figure 6.10: Bias, variance and MSE values of the Metropolis and Uphill resampling methods on the gamma distributions. The values of B in both algorithms are the same. The x-axis represents the number of particles (in logarithmic scale).

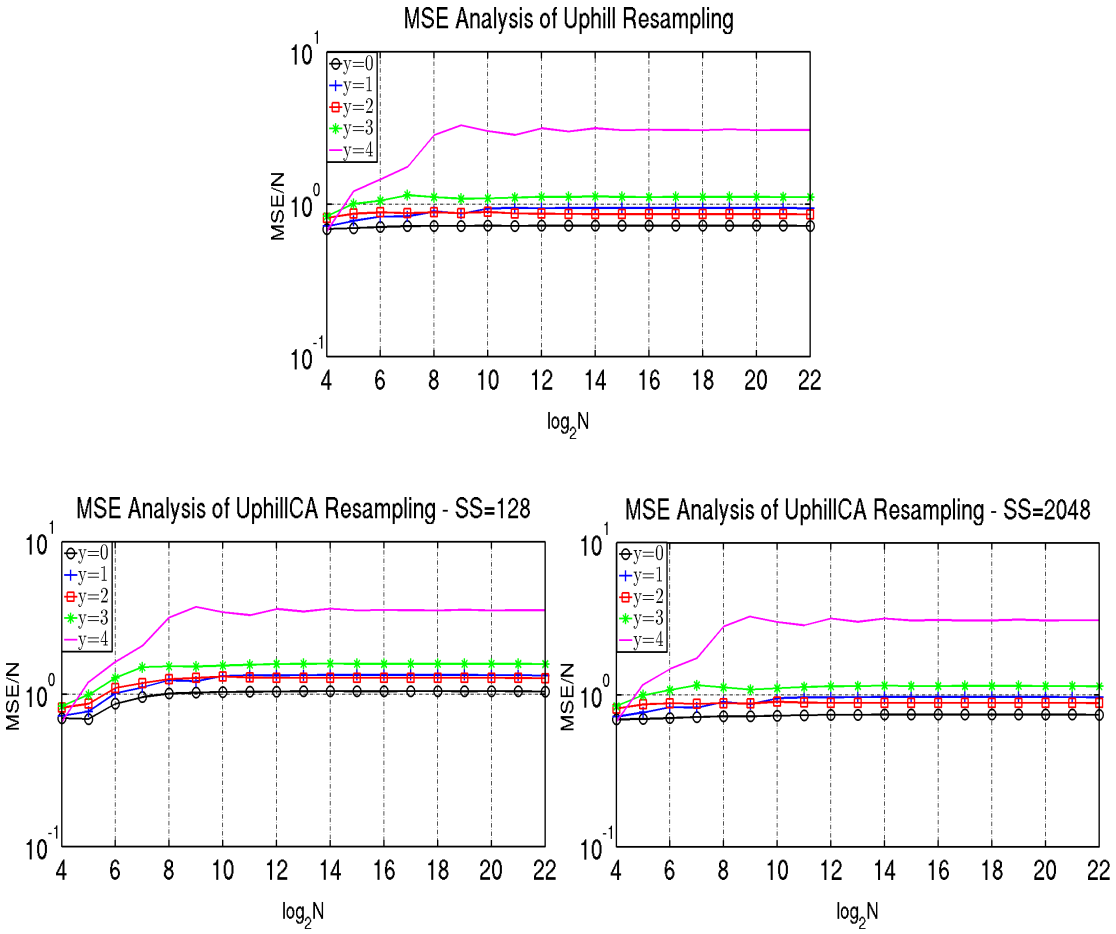

Figure 6.11: MSE results of Uphill, Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

6.7 Bias, Variance, MSE and Execution Time Results of Uphill-CA Resampling

Although the expected numbers of replications of each particle in the Uphill-CA resampling are same with Uphill, there occurs difference in variance depending on the s-segment size and warp size. In this section, we examine this variance in the MSE results. The MSE, bias and variance values of Uphill, Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ on the distributions in (2.33) are given in Figure 6.11 and Figure 6.12. The execution time and speed up results of Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ over the Uphill resampling on the distributions in (2.33) are given in Figure 6.13.

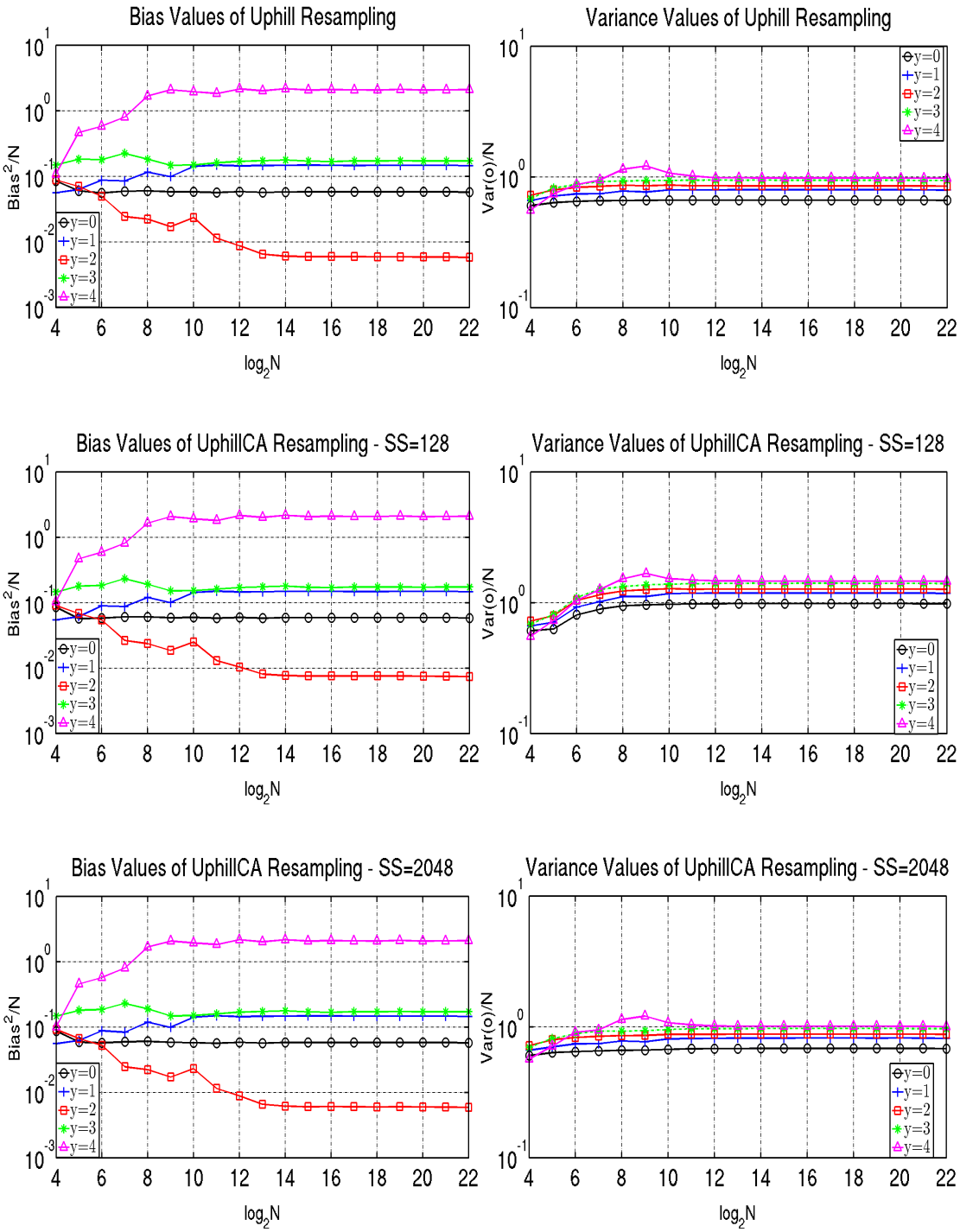

Figure 6.12: Bias and variance values of Uphill, Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

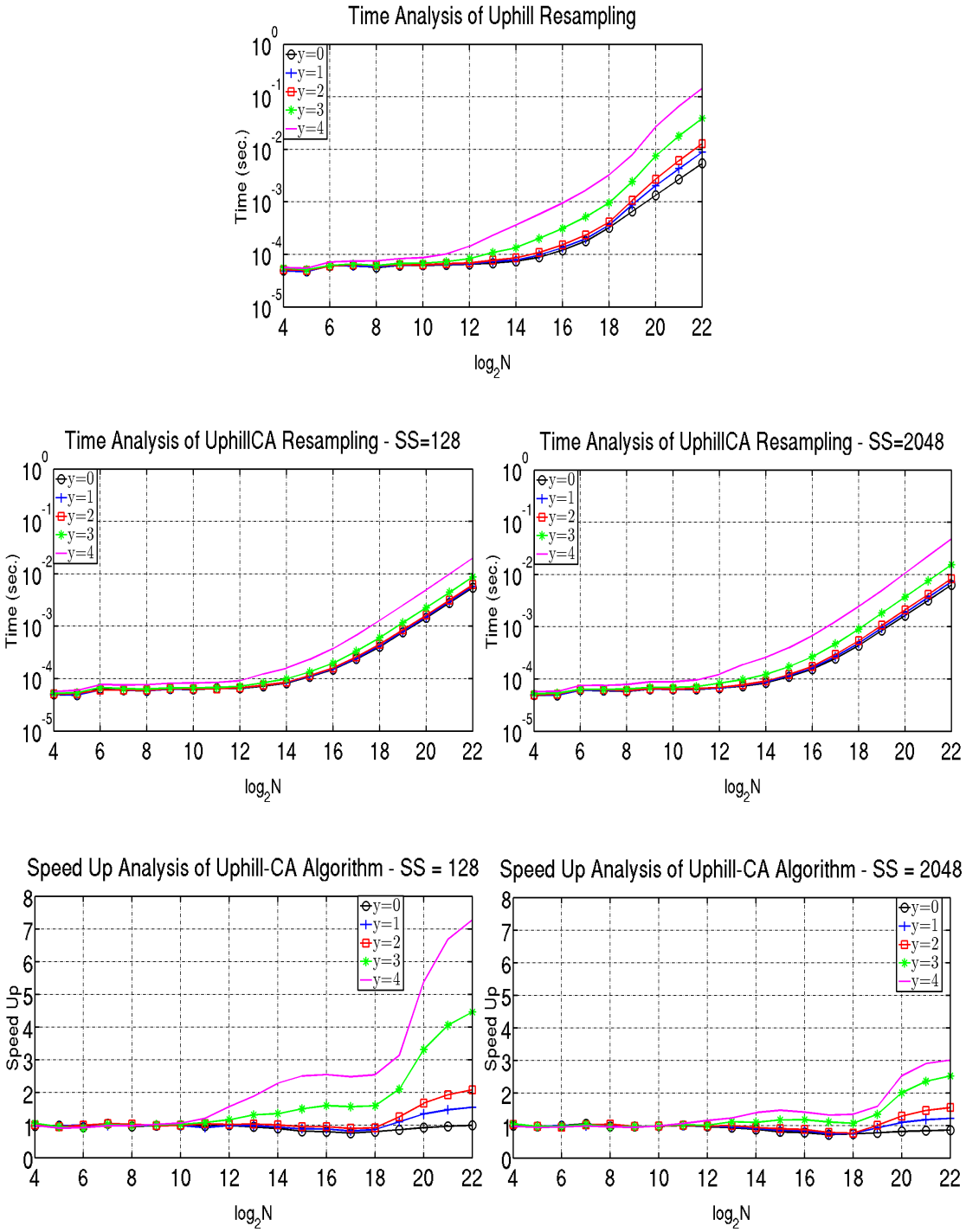

Figure 6.13: Execution time and speed up results of Uphill, Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

It is seen that the bias of three resampling algorithms are very close to each other. Uphill-CA with $SS = 128$ has the highest variance that causes its MSE to be worse than those of the others. The MSE of Uphill-CA with $SS = 2048$ improves as its variance reduces. Uphill-CA behaves similarly to the original Uphill resampling, thus, can be considered as a faster alternative to it. We can not analyze the variance caused by warp size since the warp size is a constant in the present GPU architectures. We achieve speed up over the Uphill resampling with Uphill-CA with $SS = 128$ up to seven times, and with Uphill-CA with $SS = 2048$ up to three times. The speed up becomes more pronounced as the relative variance in the weight sequence increases.

The MSE, bias and variance values of the Uphill resampling, Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ on the gamma distributions are given in Figure 6.14 and Figure 6.15. The execution time and speed up results of Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ over the Uphill resampling on the gamma distributions are given in Figure 6.16.

In terms of quality, the behaviors of three resampling algorithms are similar to their behaviors in the previous experiment. In terms of speed up over Uphill, we achieve up to three times with Uphill-CA with $SS = 128$, and up to two times with Uphill-CA with $SS = 2048$.

6.8 Bias, Variance, MSE and Execution Time Results of Uphill-C1 Resampling

Uphill-C1 is the generic version of the Uphill resampling. The behaviors of Uphill-C1 varies in theory as the s-segment differs. In this section, we discuss the behaviors of Uphill-C1 by comparing its quality and execution time with the results of Uphill. We show how bias and variance of Uphill-C1 are affected as the s-segment size differs. The MSE, bias and variance values of the Uphill resampling, Uphill-C1 with $SS = 128$ and Uphill-C1 with $SS = 2048$ on the distributions in (2.33) are shown in Figure 6.17 and Figure 6.18. The execution time and speed up results are given in Figure 6.19. The MSE, bias and variance values of them on the gamma distributions are shown in Figure 6.20 and Figure 6.21. The execution time and speed up results are given in Figure 6.22. We use the same way to calculate B as in Uphill and Uphill-CA.

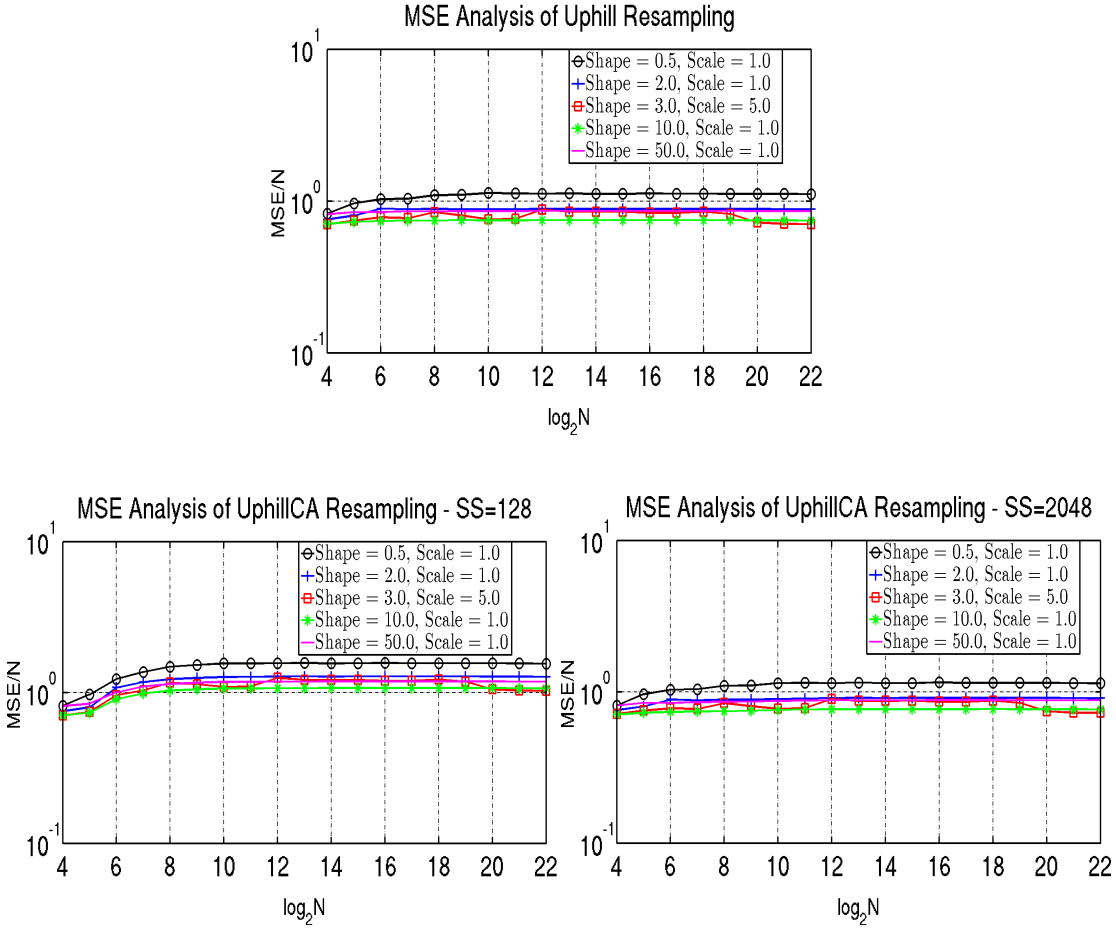

Figure 6.14: MSE results of Uphill, Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

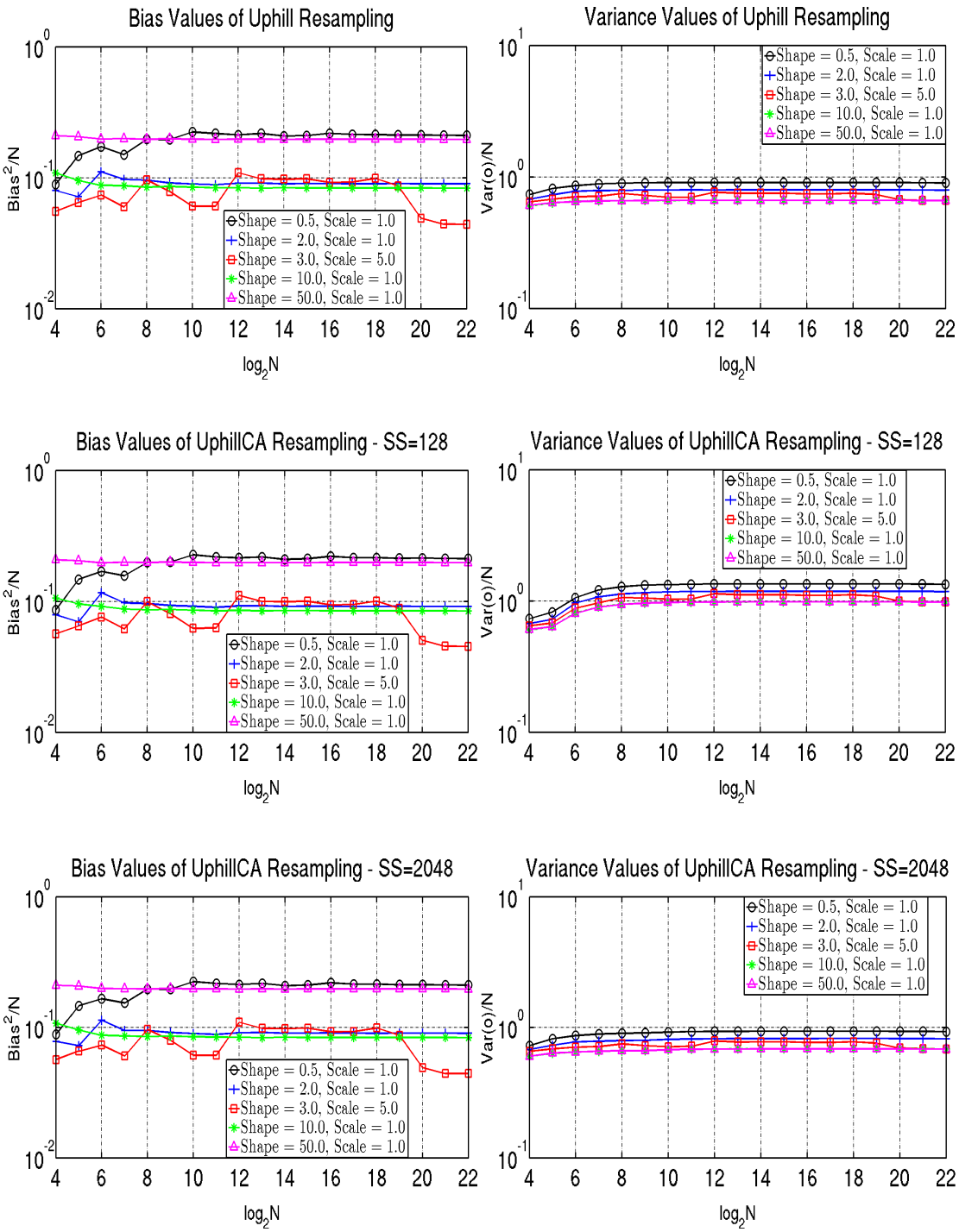

Figure 6.15: Bias and variance values of Uphill, Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

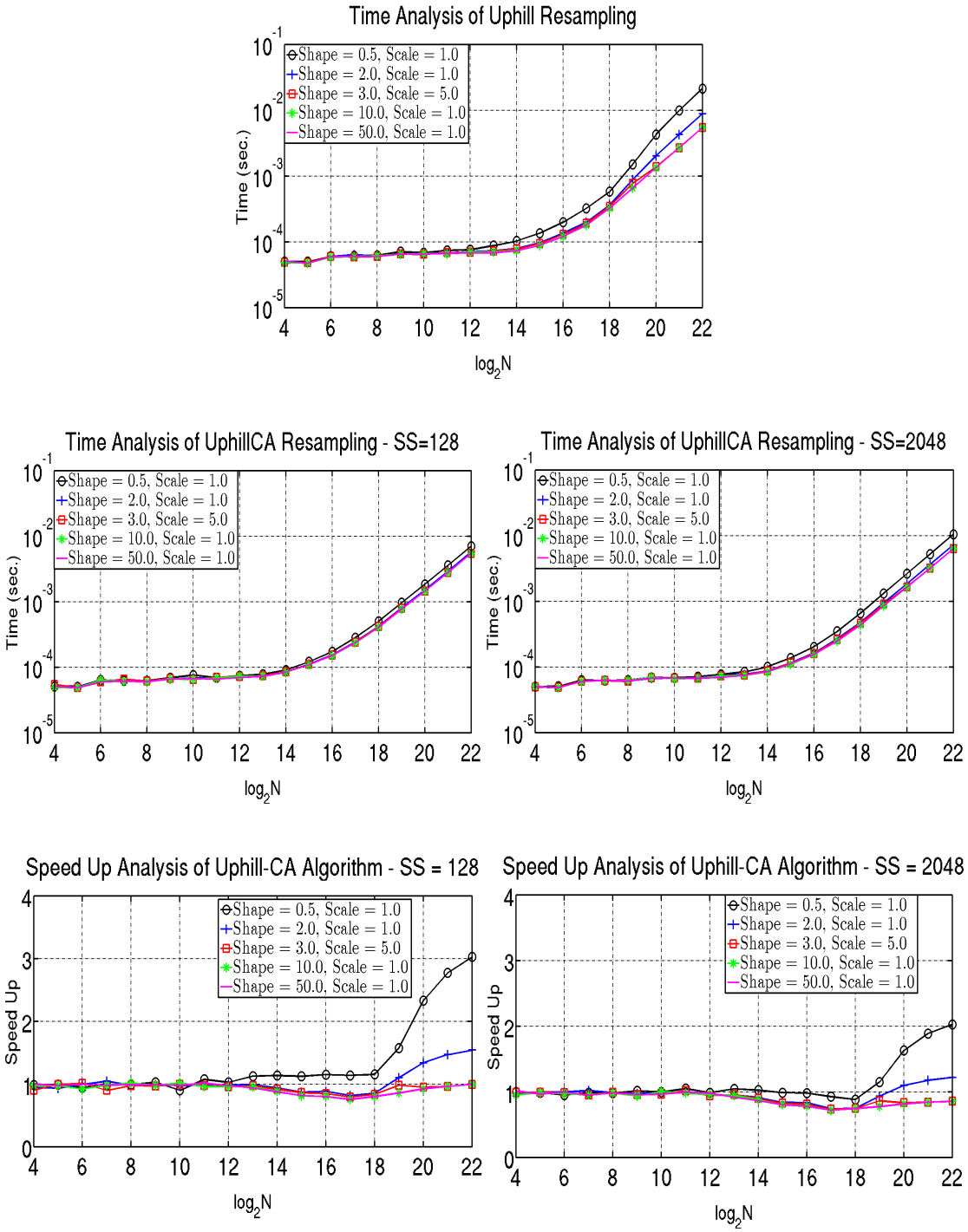

Figure 6.16: Execution time and speed up results of Uphill, Uphill-CA with $SS = 128$ and Uphill-CA with $SS = 2048$ on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

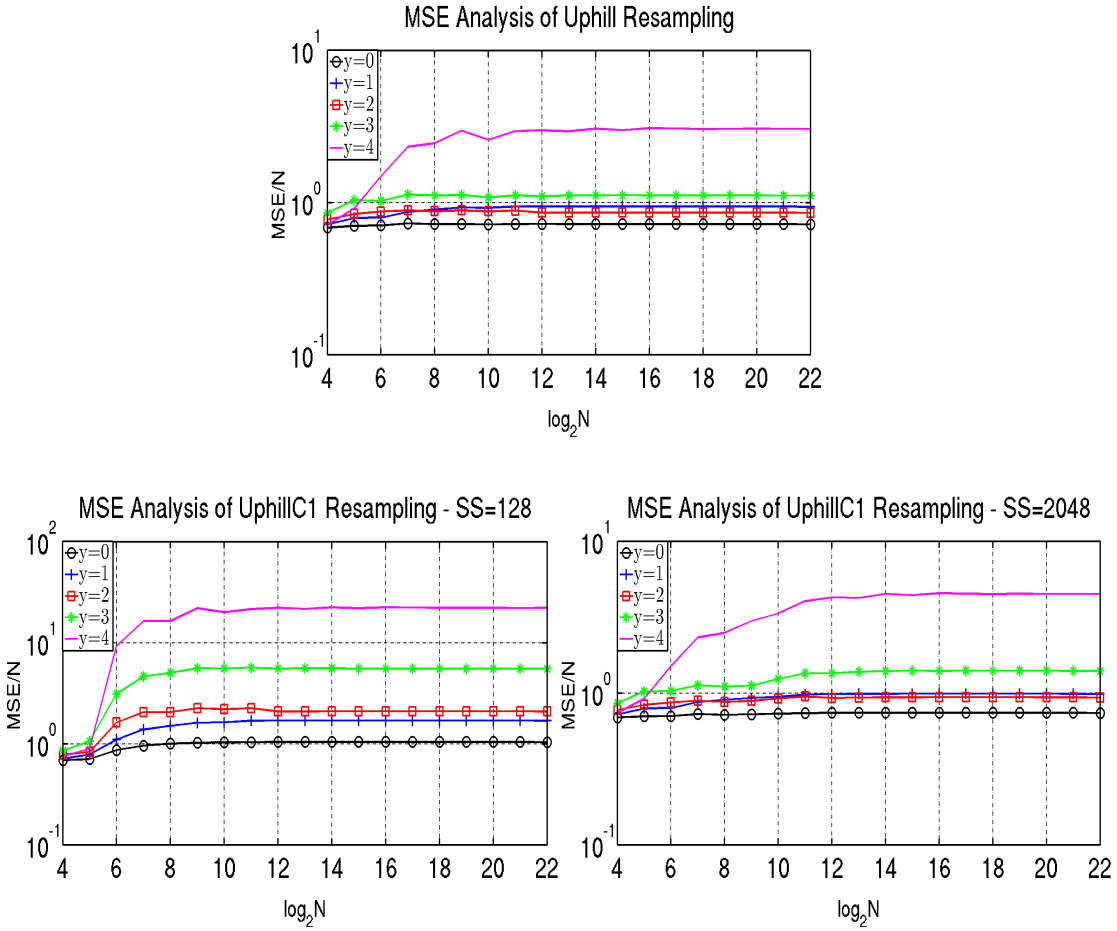

Figure 6.17: MSE results of Uphill, Uphill-C1 with $SS = 128$ and Uphill-C1 with $SS = 2048$ on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

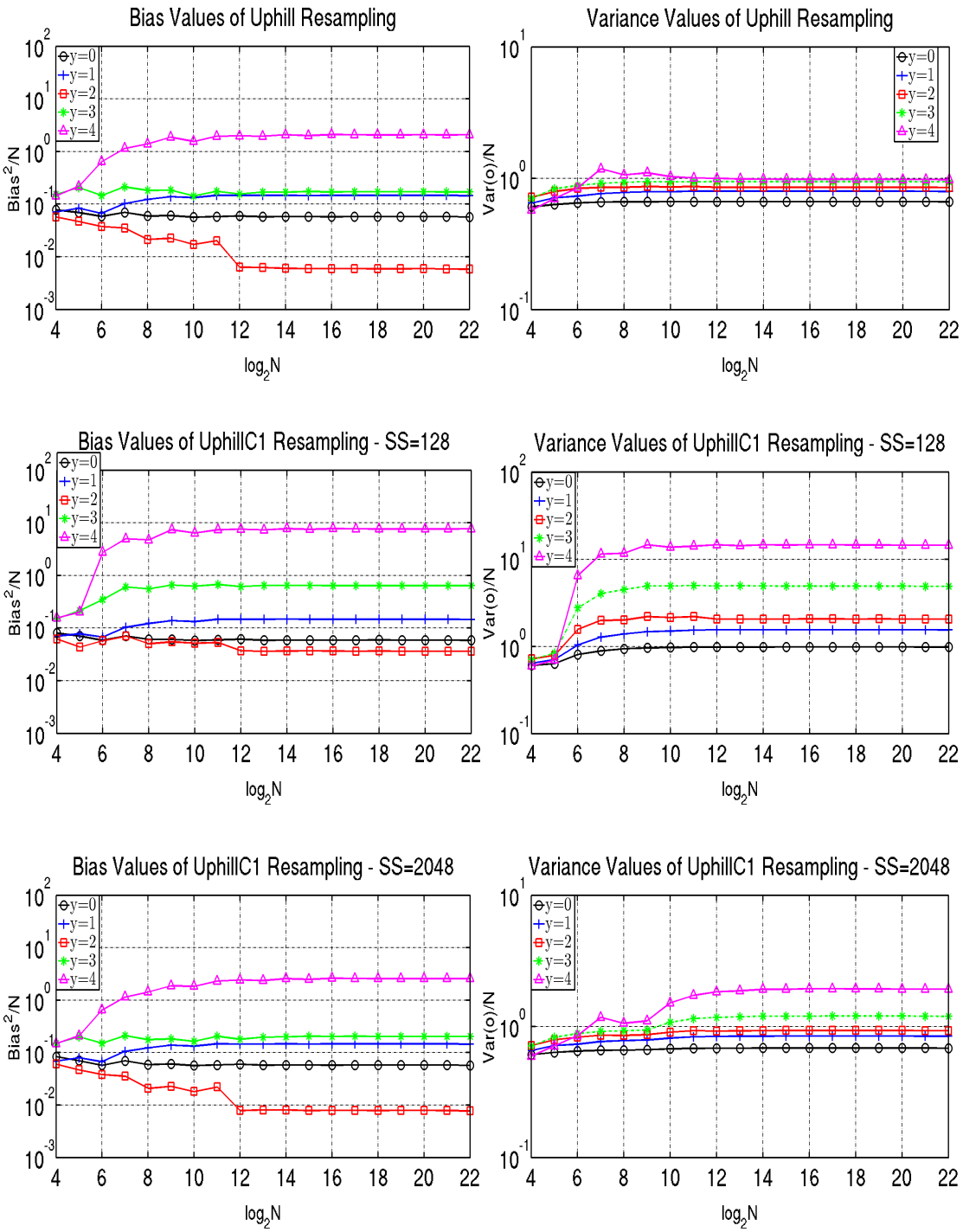

Figure 6.18: Bias and variance values of Uphill, Uphill-C1 with $SS = 128$ and Uphill-C1 with $SS = 2048$ on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

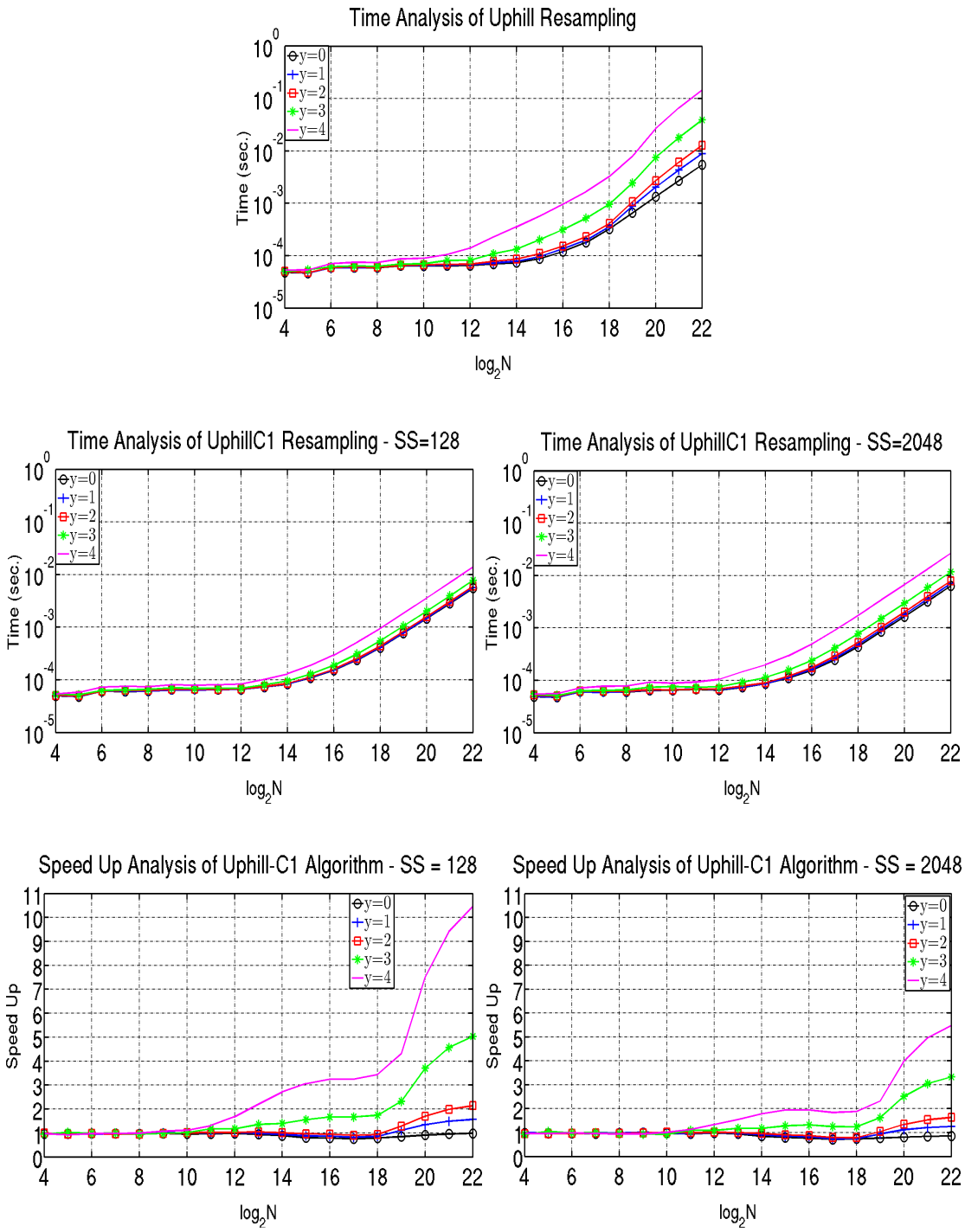

Figure 6.19: Execution time and speed up results of Uphill-C1 with $SS = 128$ and Uphill-C1 with $SS = 2048$ over Uphill on the distributions in (2.33). The x-axis represents the number of particles (in logarithmic scale).

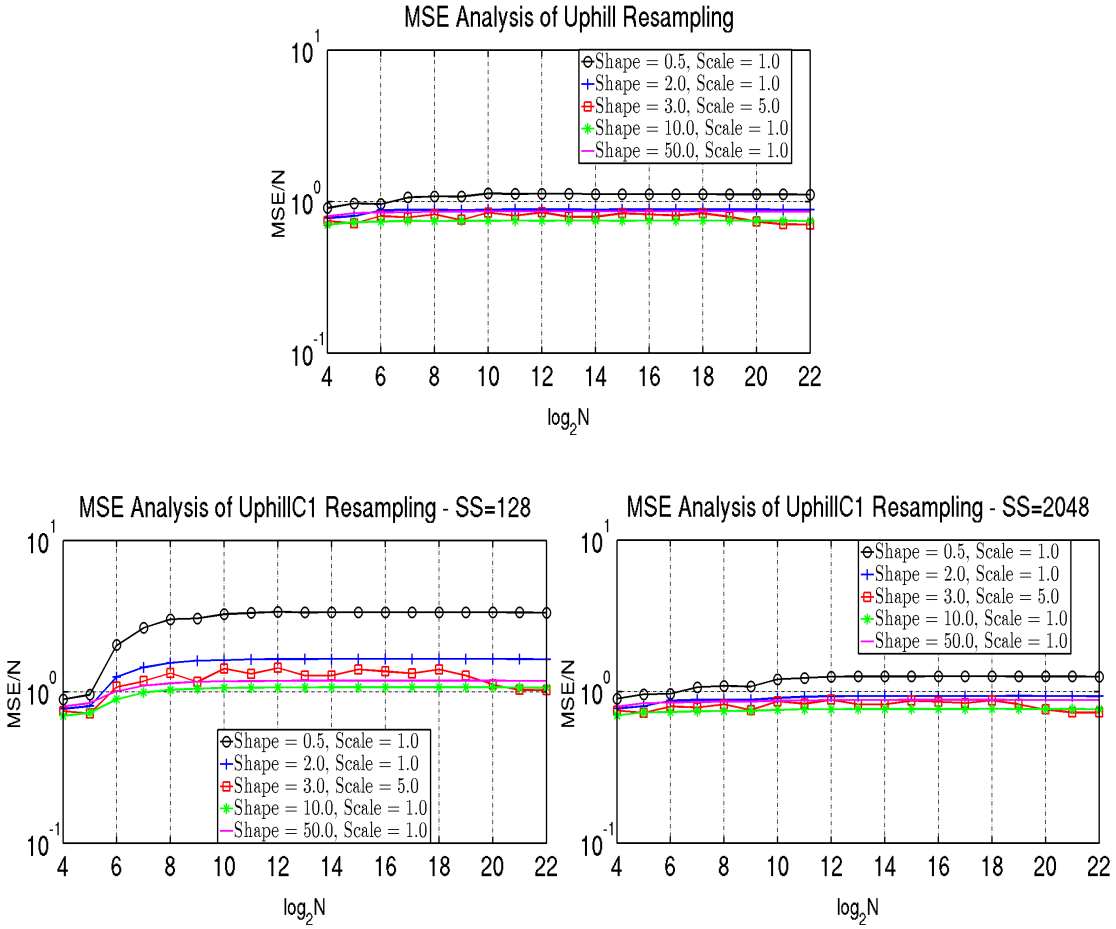

Figure 6.20: MSE results of Uphill, Uphill-C1 with $SS = 128$ and Uphill-C1 with $SS = 2048$ on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

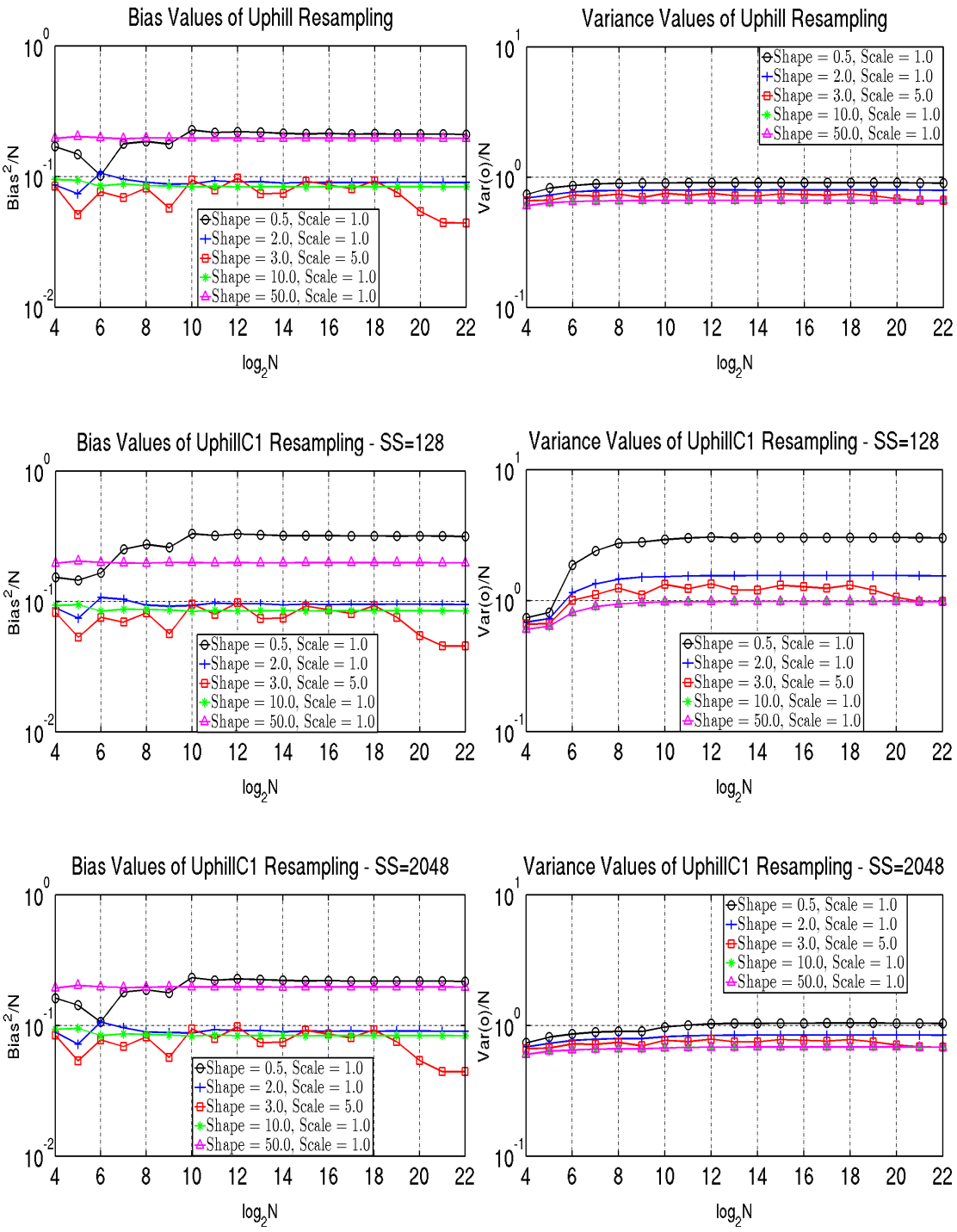

Figure 6.21: Bias and variance values of Uphill, Uphill-C1 with $SS = 128$ and Uphill-C1 with $SS = 2048$ on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

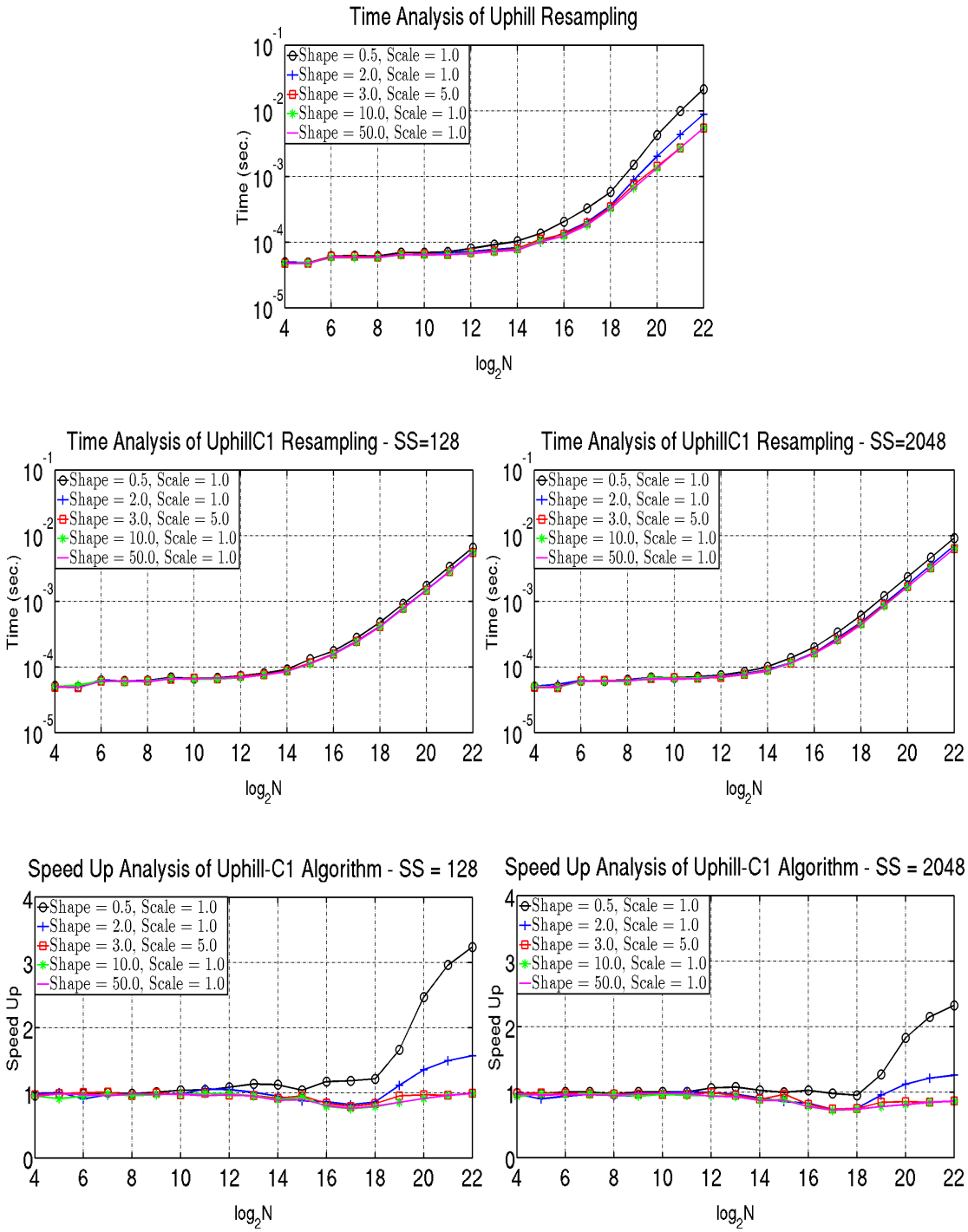

Figure 6.22: Execution time and speed up results of Uphill-C1 with $SS = 128$ and Uphill-C1 with $SS = 2048$ over the Uphill resampling on the gamma distributions. The x-axis represents the number of particles (in logarithmic scale).

It is seen that the bias of Uphill and Uphill-C1 with $SS = 2048$ are very close to each other, but the bias of Uphill-C1 with $SS = 128$ are worse than the Uphill resampling. S-segment size causes additional bias for Uphill-C1 with $SS = 128$. On the other hand, the MSE of Uphill-C1 with $SS = 128$ are much worse than the Uphill resampling whereas the MSE of Uphill and Uphill-C1 with $SS = 2048$ are very close to each other. Variances caused by s-segment size lead to much worse MSE for Uphill-C1 with $SS = 128$. This shows us, s-segment size causes both bias and variance on the MSE of Uphill-C1. And the results of Uphill-C1 with $SS = 2048$ show us the behavior of Uphill-C1 becomes same with the behavior of Uphill as the s-segment size approaches to $4N$. Using Uphill-C1 in tracking applications would not be much problem since the bias and variance caused by s-segment size are not significant in filtering applications. We can not analyze the variance caused by warp size since the warp size is a constant in the GPU. However, we can surmise that the variance increases as the warp size increases.

We achieve speed up with Uphill-C1 with $SS = 128$ up to ten times of the Uphill resampling and achieve speed up with Uphill-C1 with $SS = 2048$ up to five times of the Uphill resampling for the distributions in (2.33). We achieve speed up with Uphill-C1 with $SS = 128$ up to three times of the Uphill resampling and achieve speed up with Uphill-C1 with $SS = 2048$ up to two times of the Uphill resampling for the gamma distributions. The speed up becomes more pronounced as the relative variance on the weight sequence increases.

6.9 A Simple End-to-End Application

To provide a perspective for the reader on the significance of the resampling stage in the overall cost of the SIR particle filter, we run the particle filter in Algorithm 2.4 on a highly non-linear system given in (2.34) and (2.35).

We create 16 different trajectories and run them 100 times with each one of the five resampling algorithms. We use these 16 trajectories in all the number of particles. The results are the average of the results of these 16 trajectories. We set the number of time steps to 100. The initial state x_0 is set to 0.1. We measure the quality using the

root mean squared error (RMSE) metric [32]. We calculate the error at each time step by getting the difference between the true state and estimated state. We set ϵ to $1/10$ in calculation of B of the Metropolis resampling to accelerate it without sacrificing quality much [22]. We set the *–use_fast_math* flag at compile time to enable efficient calculation of special functions such as cosines, square root, and exponential [28]. In stage 4, we use the weights of particles without normalization for the Metropolis, Rejection and Uphill resampling methods, and the normalized weights for the Systematic resampling method. The ratios of each stage to the total execution times of the filter with varying numbers of particles are given in Table 6.2.

It is seen that the resampling stage takes a big portion of the total execution time. The ratio of time spent of the Uphill resampling increases as the number of particle increases. This is not valid for the Systematic resampling, in fact the ratio of time spent of it decreases as the number of particles increases. In the Metropolis and Rejection resampling, the ratios of time spent increase as the number of particles increases. This is also same for Uphill-CA with SS = 128 and Uphill-C1 with SS = 128. The improvement in execution time brought about by Uphill-CA and Uphill-C1 becomes more pronounced in PF applications. The RMSE results of the experiments are given in Table 6.3.

The results show us the RMSE results of each resampling algorithms are very close the each other. Therefore, the bias and variance in the results of the Uphill and Uphill-CA resampling would not be much problem in this kind of applications. Uphill-CA with $SS = 128$ seems preferable to the Systematic resampling in terms of execution time, with the additional benefit of numerical stability. The RMSE of Uphill-C1 with $SS = 128$ is slightly worse than that of others, but this may not be significant. Hence, one can prefer Uphill-C1 with $SS = 128$, if speed is the primary concern.

Table 6.2: The ratio of times spent in different stages. The results are percentage of the total execution times. Total execution times (in seconds) are also given in the last row.

$log_2N = 14$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	13.87	5.45	9.71	7.72	11.51	12.91
Stage2	13.10	5.14	9.17	7.28	10.65	11.94
Stage3	21.03	8.26	14.70	11.71	17.20	19.30
Stage4	52.01	81.15	66.42	73.29	60.63	55.85
Exec. Time	0.014	0.035	0.020	0.025	0.017	0.016
$log_2N = 16$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	23.83	5.45	11.72	8.46	13.77	16.48
Stage2	10.76	2.43	5.24	3.78	6.15	7.36
Stage3	17.52	3.97	8.52	6.14	10.00	11.98
Stage4	47.89	88.15	74.52	81.62	70.08	64.18
Exec. Time	0.021	0.092	0.043	0.059	0.036	0.030
$log_2N = 18$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	34.88	5.15	11.70	8.16	14.03	17.88
Stage2	7.88	1.17	2.64	1.85	3.18	4.06
Stage3	14.29	2.11	4.79	3.34	5.75	7.32
Stage4	42.95	91.58	80.87	86.65	77.04	70.74
Exec. Time	0.046	0.312	0.137	0.197	0.115	0.090
$log_2N = 20$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	40.62	2.67	7.81	4.31	14.16	18.50
Stage2	6.46	0.42	1.24	0.68	2.25	2.94
Stage3	12.60	0.83	2.42	1.34	4.39	5.74
Stage4	40.31	96.08	88.53	93.66	79.21	72.82
Exec. Time	0.149	2.277	0.777	1.407	0.429	0.328
$log_2N = 22$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	42.69	1.88	6.74	3.11	13.56	18.68
Stage2	5.54	0.24	0.87	0.40	1.76	2.43
Stage3	11.66	0.51	1.84	0.85	3.70	5.10
Stage4	40.11	97.36	90.55	95.63	80.97	73.79
Exec. Time	0.556	12.602	3.524	7.631	1.751	1.271

Table 6.3: RMSE Results.

Particle No.	16384	65536	262144	1048576	4194304
Systematic	4.47061	4.47037	4.46955	4.46965	4.46975
Metropolis	4.47114	4.47121	4.47062	4.47026	4.47021
Rejection	4.47184	4.46996	4.47007	4.46976	4.46970
Uphill	4.46984	4.46885	4.46855	4.46832	4.46942
Uphill-CA-128	4.46903	4.46840	4.46806	4.46839	4.46927
Uphill-C1-128	4.50601	4.50465	4.50318	4.50253	4.50237

6.10 Speed up Analysis of SIR Particle Filter over CPU Implementation

In this section, we investigate the execution times of the SIR particle filter on CPU and GPU. We experiment the speed up of its GPU implementation over its CPU implementation. We run the particle filter in Algorithm 2.4 on a highly non-linear system given in (2.34) and (2.35). The CPU processor is Intel Core i7-4790K along with 16 GB RAM.

We create 16 different trajectories and run them 10 times with the Uphill resampling algorithm. We use these 16 trajectories in all the number of particles. The results are the average of the results of these 16 trajectories. We set the number of time steps to 100. The initial state x_0 is set to 0.1. We measure the quality using the root mean squared error (RMSE) metric [32]. We calculate the error at each time step by getting the difference between the true state and estimated state. We set the *–use_fast_math* flag at compile time to enable efficient calculation of special functions such as cosines, square root, and exponential [28]. In stage 4, we use the weights of particles without normalization. The speed up results are given in Figure 6.23.

We achieve up to 98x speed up over the CPU implementation of the SIR particle filter. Since the resampling stage takes a big portion of the total execution time, improving resampling execution time is important in the GPU implementation. We achieve up to 79x speed up in resampling execution time. When the number of particles exceeds 2^{18}, the ratio of speed up decreases. This is because of the effect of L2 cache size. The size of L2 cache is not enough to load all data when the number of particles exceeds 2^{18}. Note that, the size of L2 cache is 1.5 MB in Tesla K40 board.

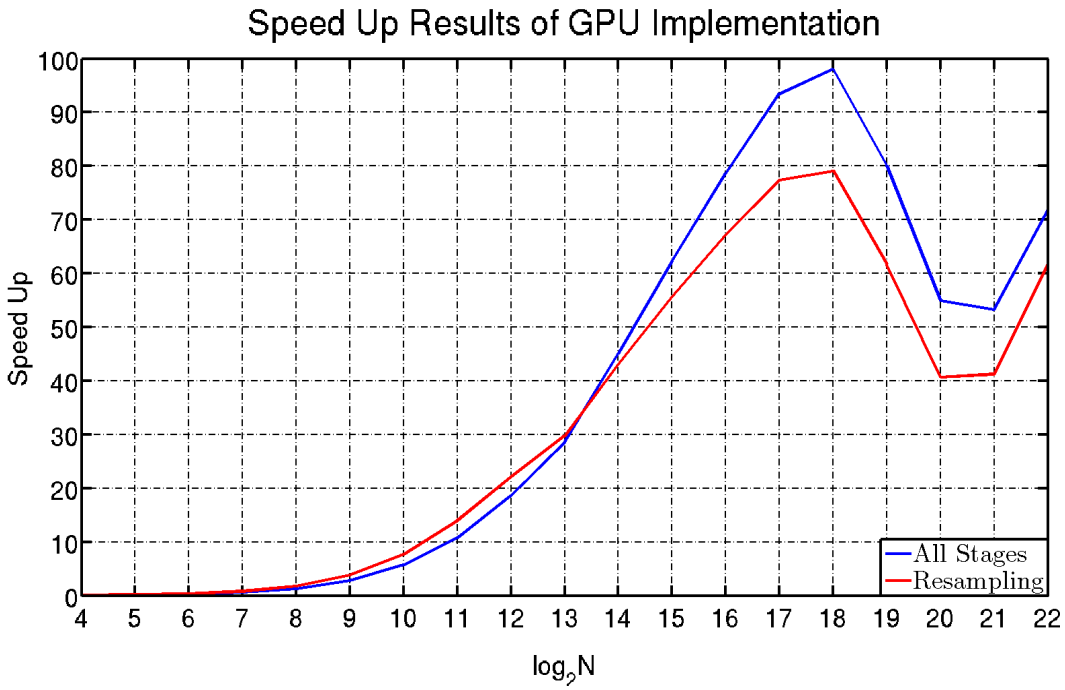

Figure 6.23: The speed up results of the GPU implementation of the SIR Particle Filter over the CPU implementation of it.

6.11 Global Memory Load Transactions of Uphill and its Variations

We have already mentioned the expected number of global memory load transactions of Uphill, Uphill-C1 and Uphill-CA. When we set the size of s-segment as 128 KB, we expect that the resampling algorithms read $\tilde{w}^j$ in a single global memory load transaction. When we set it as 2048, we expect that the resampling algorithms read $\tilde{w}^j$ in at most 16 global memory load transactions. In this section, we experiment with the number of global memory load transactions by using the profiler of CUDA [29].

We select a distribution that has the longest execution time among the ones we use. This is the distribution in (2.33) by setting the value of y to 4. For each resampling algorithm, we create 16 different weight sequences for any N. The resampling algorithms draw 256 (the value of K) offspring sequences from each weight sequence. The results are the averages of these 16 weight sequences. We use the corresponding metric in CUDA profiler which is *gld_transactions_per_request* [29]. It gives us the average number of global memory load transactions performed for each global

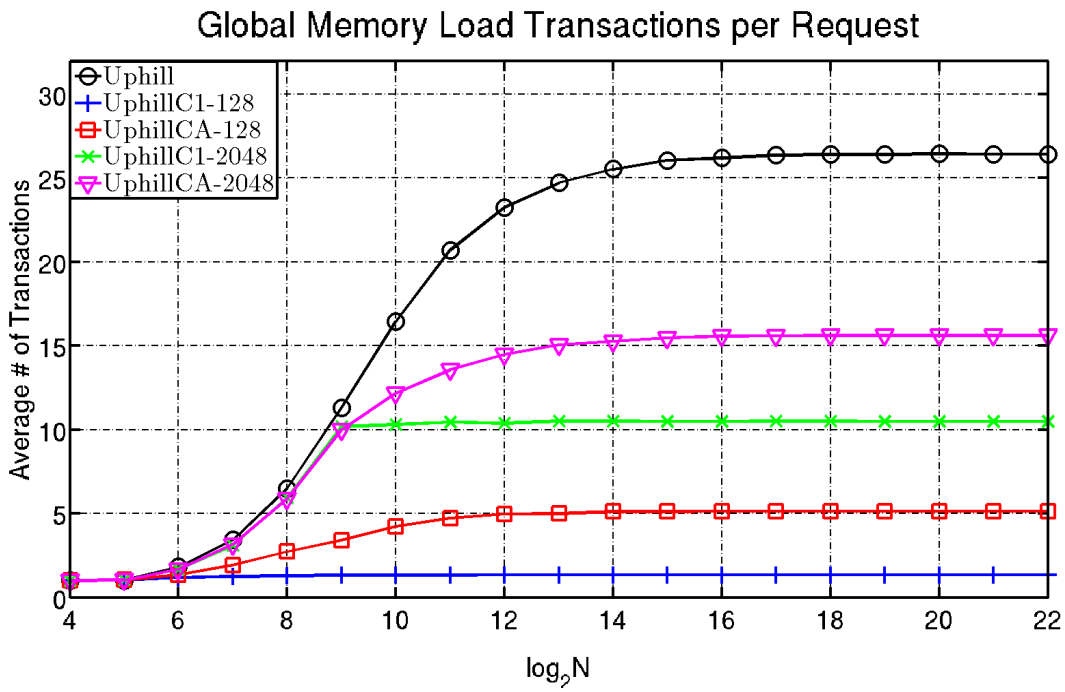

Figure 6.24: The global memory load transactions per request for Uphill, Uphill-C1 and Uphill-CA.

memory load. We only measure the kernels of Uphill, Uphill-C1 and Uphill-CA. The experimental results are given in Figure 6.24.

When there is sufficient data (the number of particles), the resampling algorithms converge to some points in terms of transactions. The average number of transactions of Uphill-C1 with $SS = 128$ is around 1.34 and Uphill-CA with $SS = 128$ is around 5.13. The values are greater than one because of the non-coalesced read of $\tilde{w}^t$. The average number of transactions of Uphill-C1 with $SS = 2048$ is around 10.49 and Uphill-CA with $SS = 2048$ is around 15.61. Both algorithms approach to similar number of transactions for the read of $\tilde{w}^j$. However, the result of Uphill-CA with $SS = 2048$ is around 15.61 because it is more prone to non-coalesced read of $\tilde{w}^t$. The average number of transactions of Uphill is around 26.43.

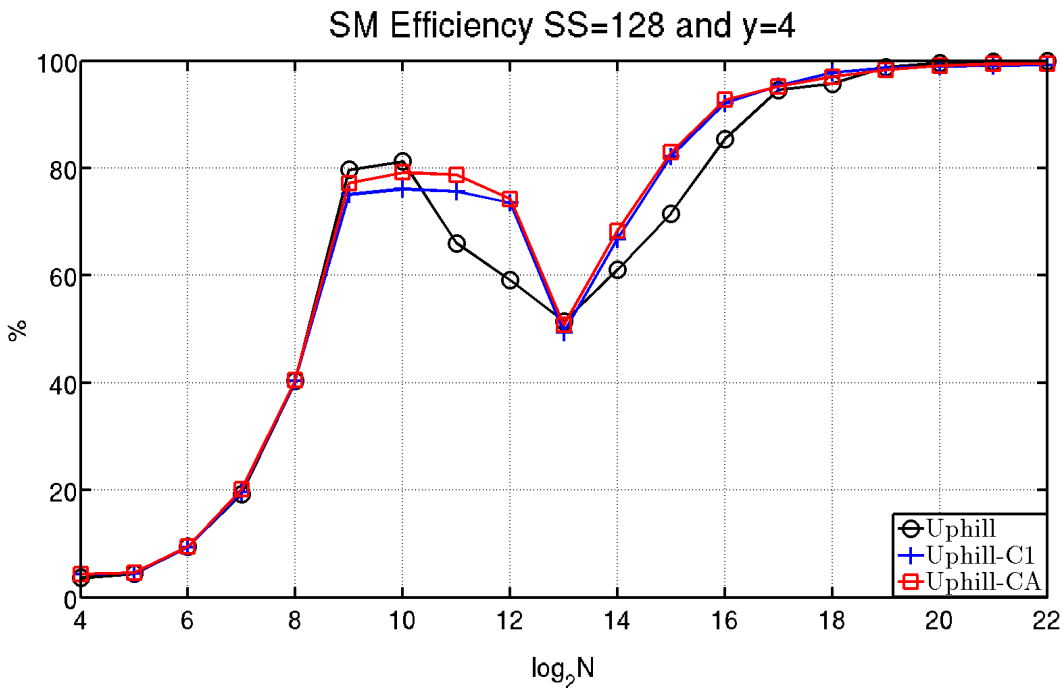

Figure 6.25: SMX Efficiency of Uphill, Uphill-C1 and Uphill-CA.

6.12 SMX Efficiency of Uphill and its Variations

In this section, we investigate the efficiency of SMXs on the board. We select a distribution that has the longest execution time among the ones we use. This is the distribution in (2.33) by setting the value of y to 4. For each resampling algorithm, we create 16 different weight sequences for any N. The resampling algorithms draw 256 (the value of K) offspring sequences from each weight sequence. The results are the averages of these 16 weight sequences. We use the corresponding metric in CUDA profiler which is *sm_efficiency* [29]. It gives us the percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU. We only measure the kernels of Uphill, Uphill-C1 and Uphill-CA. The experimental results are given in Figure 6.25.

It is seen that when the number of particles is large, the efficiency approaches to 99%. There occurs a sharp decrease when the number of particles is between 2^{10} and 2^{16}. This is because of the residual blocks on several SMXs. To give an example, if we have 16 blocks, 14 SMXs will run 1 block and 1 SMX will run 2 blocks. Or if we have 32 blocks, 13 SMXs will run 2 blocks and 2 SMXs will run 3 blocks. This is the

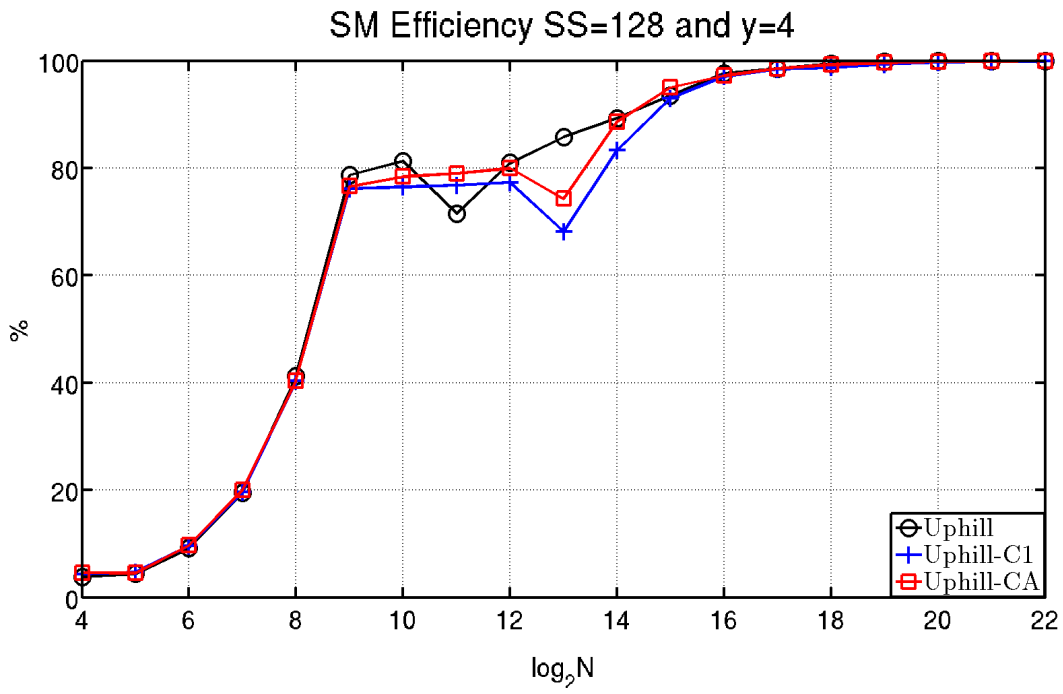

Figure 6.26: SMX Efficiency of Uphill, Uphill-C1 and Uphill-CA.

main reason of the decrease in the SMX efficiency. One way to alleviate this situation is to decrease the size of the blocks appropriately so that the warps of residual blocks can be distributed to SMXs as evenly as possible. Remember that we define the block sizes in (2.28). We reduce the block sizes to 32 for all the number of particles except 2^4 particles and obtain the experiment results given in Figure 6.26.

It is seen that the SMX efficiency improves especially when the number of particles is between 2^{10} and 2^{16}.

6.13 Factors on the Execution Time of Uphill and its Variations

In this section, we investigate the factors that affect the execution times of Uphill and its variations. There are three main factors. These are physical resources and limitations, non-coalesced global memory access and the number of particles. We experiment how they have an effect on the execution times of the resampling algorithms.

We select a distribution that has the longest execution time among the ones we use.

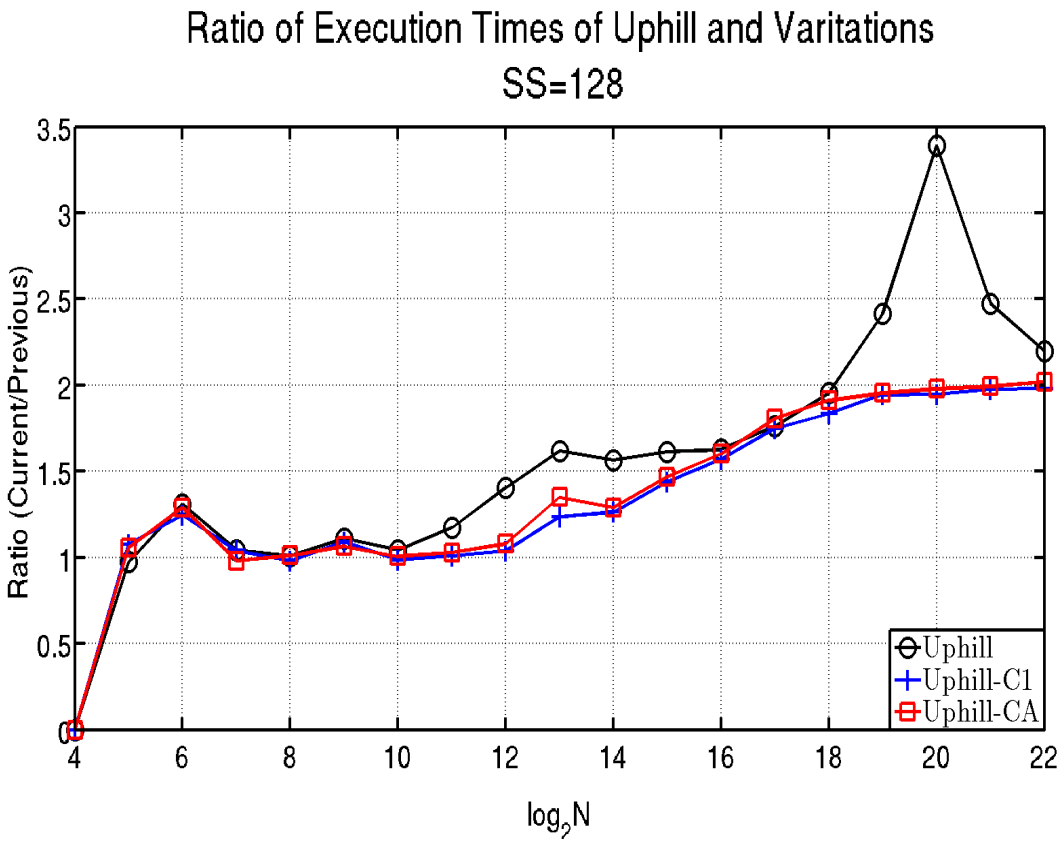

Figure 6.27: The ratio of execution times of Uphill, Uphill-C1 and Uphill-CA.

This is the distribution in (2.33) by setting the value of y to 4. We create 16 different weight sequences for any N. Each resampling algorithm draws 256 (the value of K) offspring sequences from each weight sequence. The results are the averages of these 16 weight sequences. The s-segment size of Uphill-C1 and Uphill-CA is 128 bytes. The experimental results are given in Figure 6.27. The values are the ratios of the current execution times of the resampling algorithms to the previous execution times of them.

We have enough resources up to 2^{11} particles. Remember that Tesla K40 board has 15 SMXs where each have 192 cores. When the number of particles exceeds 2^{11}, some warps wait for available cores to issue their instructions. When a warp completes its job on the cores, another warp in waiting status starts to run on the cores. All blocks are active at the beginning up to 2^{14} particles. When the number of particles exceeds 2^{14}, some blocks wait for available SMX to be scheduled. When a block completes its kernel another block in waiting status becomes active and is scheduled to the SMX.

The size of L2 cache is enough up to 2^{18} particles. Note that the size of L2 cache is 1.5 MB.

Uphill is affected rapidly from the non-coalesced global memory access problem than Uphill-C1 and Uphill-CA as the number of particles increases. It achieves its worst case in terms of global memory load transactions when the number of particles exceeds 2^{15}. However, Uphill-C1 achieves its worst case in smaller number of particles and Uphill-CA achieves its worst case when the number of particles exceeds 2^{11}. This is the main reason why the results of Uphill are larger than those of the others when the number of particles is between 2^{11} and 2^{15}. Uphill is also affected from the lack of L2 cache memory. When the number of particles is 2^{19}, Uphill can not load 27% of its data to L2 cache. When it is 2^{20}, Uphill can not load 64% of its data to L2 cache. When it is 2^{21}, Uphill can not load 82% of its data to L2 cache. This is the reason of the peak point occurred when the number of particles is 2^{20}. The local peak is also occurred because of the ratios of the warps that wait the cores when the number of particles is 2^{13}. It has similar behavior with the effects of lacking L2 cache. 30% of the threads are waiting for the cores when the number of particles is 2^{12}, 65% of the threads when the number of particles is 2^{13} and 83% when 2^{14}. The efficiency of the SMX has also effects on the algorithms. We investigate it in the previous section. When all the factors except the number of particles achieve their worst case or have insignificant effects on the ratios of the execution times of the algorithms, the ratios of all resampling algorithms approach to 2. From that point on, the execution time doubles as the number of particles doubles.

CHAPTER 7

TRACKING PERFORMANCE OF RESAMPLING ALGORITHMS

In this chapter, we discuss the tracking performance of resampling algorithms. We compute the execution time of each stage of the SIR particle filter given in Algorithm 2.4. We discuss the execution time results of resampling stage. Furthermore, we compare the quality of resampling algorithms by measuring their root mean squared error (RMSE).

7.1 Execution time and RMSE Results of Resampling Algorithms

We evaluate the tracking performance of resampling algorithms on a simple application. Our application is target tracking with radar. The target, possibly an aircraft, performs a maneuver. In the system model, we assume nearly constant velocity. In the measurement model, we use the range and bearing of the target. The $x = [p_x p_y v_x v_y]^T$ is the state vector of the target in $2D$. The p_x and p_y are the x position and y position of the target, respectively; v_x and v_y are the velocity of the same. The system model is given in (7.1) and the measurement model is given in (7.2).

$$x_k = \begin{bmatrix} 1 & 0 & T & 0 \\ 0 & 1 & 0 & T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * x_{k-1} + \begin{bmatrix} \frac{T^2}{2} & 0 \\ 0 & \frac{T^2}{2} \\ T & 0 \\ 0 & T \end{bmatrix} * q_k \tag{7.1}$$

Table 7.1: Simulation Parameters (Scenario 1).

Parameters	Value	Description
T	1	Sampling Time
x_0	$[1000\ 1000\ 0\ 0]^T$	Initial State Vector
P_0	$diag[100^2\ 100^2\ 10^2\ 10^2]$	x_0 covariance
Q	$diag[10\ 10]$	q_k covariance
R	$diag[10^2\ (0.1\pi/180)^2]$	v_k covariance

$$y_k = \begin{bmatrix} \sqrt{p_{xk}^2 + p_{yk}^2} \\ \arctan(\frac{p_{yk}}{p_{xk}}) \end{bmatrix} + v_k \tag{7.2}$$

$q_k \sim N(0, Q)$ and $v_k \sim N(0, R)$ are Gaussian process noise and measurement noise, respectively. T is the sampling time, k is the current time step. The measurement units are m and m/sec.

We create 16 different trajectories by adding different measurement noises to the true state of the aircraft which is given in Figure 7.1. We run them 100 times with each one of the six different resampling algorithms. We use these 16 trajectories in all the number of particles. And the results shown in the Table 7.2 and Table 7.3 are the average of the results of these 16 trajectories. The simulation parameters are given in Table 7.1. We measure the quality using the root mean squared error (RMSE) metric [32]. We set ϵ to $1/10$ in the calculation of the B parameter of the Metropolis resampling to accelerate the Metropolis resampling without sacrificing quality much [22]. We set the *–use_fast_math* flag at compile time to enable efficient calculation of special functions such as cosine, square root, and exponential in CUDA math api [28]. We use the SIR particle filter given in Algorithm 2.4 with a variety of resampling methods. The ratios of times spent by the four stages of Algorithm 2.4 are given in Table 7.2. The RMSE results are given in Table 7.3.

It is seen that that the resampling stage (Stage4) takes up the largest portion of the execution time in all the combinations tried. The ratio of the time spent for resampling increases as the number of particles increases, except for the Systematic resampling. It seems that the improvements in execution times by Uphill-CA and Uphill-C1 will

Table 7.2: The ratio of times spent in different stages (Scenario 1). The results are percentage of the total execution times. Total execution times (in seconds) are also given in the last row.

$log_2N = 14$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	13.49	5.53	8.75	10.09	13.09	14.01
Stage2	9.52	3.92	6.23	7.17	9.12	9.73
Stage3	30.84	12.84	20.34	23.47	29.95	32.00
Stage4	46.14	77.72	64.69	59.27	47.84	44.26
Exec. Time	0.030	0.071	0.045	0.039	0.031	0.029
$log_2N = 16$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	22.34	5.60	10.34	12.24	17.04	19.35
Stage2	7.41	1.85	3.42	4.05	5.64	6.40
Stage3	24.55	6.14	11.35	13.43	18.68	21.21
Stage4	45.70	86.41	74.88	70.28	58.64	53.04
Exec. Time	0.046	0.183	0.099	0.084	0.060	0.053
$log_2N = 18$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	31.77	4.94	9.30	10.81	18.66	22.55
Stage2	5.24	0.81	1.53	1.79	3.08	3.72
Stage3	18.57	2.88	5.44	6.32	10.92	13.19
Stage4	44.41	91.36	83.73	81.08	67.34	60.53
Exec. Time	0.105	0.675	0.358	0.308	0.179	0.148
$log_2N = 20$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	36.73	2.64	6.69	6.79	19.42	24.24
Stage2	4.25	0.30	0.77	0.78	2.24	2.80
Stage3	14.71	1.06	2.68	2.72	7.78	9.71
Stage4	44.31	96.00	89.87	89.71	70.56	63.25
Exec. Time	0.343	4.770	1.883	1.856	0.648	0.519
$log_2N = 22$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	38.40	1.90	5.98	5.33	19.10	24.90
Stage2	3.63	0.18	0.57	0.50	1.80	2.35
Stage3	13.08	0.65	2.04	1.82	6.50	8.48
Stage4	44.89	97.28	91.42	92.35	72.59	64.27
Exec. Time	1.285	26.014	8.253	9.258	2.584	1.982

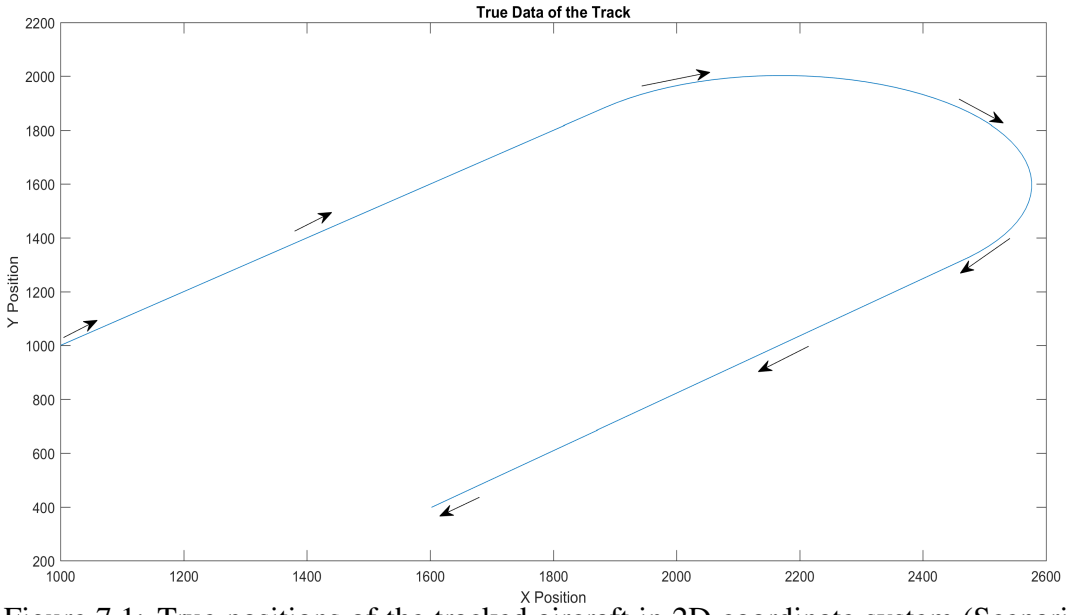

Figure 7.1: True positions of the tracked aircraft in 2D coordinate system (Scenario 1).

Table 7.3: RMSE Results (Scenario 1).

Particle No.	16384	65536	262144	1048576	4194304
Systematic	7.87608	7.84466	7.83592	7.83344	7.83247
Metropolis	7.88148	7.84990	7.84052	7.83797	7.83726
Rejection	7.87885	7.84694	7.83657	7.83268	7.83236
Uphill	7.87073	7.83663	7.82887	7.82743	7.82833
Uphill-CA-128	7.87093	7.83890	7.82862	7.82785	7.82808
Uphill-C1-128	7.85419	7.83717	7.83338	7.83353	7.83357

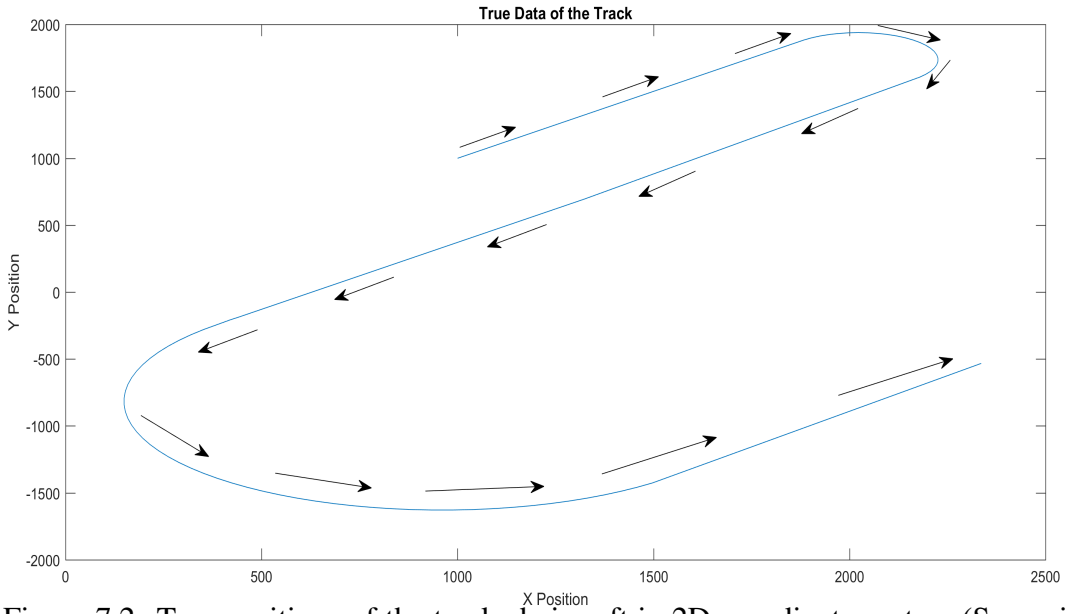

Figure 7.2: True positions of the tracked aircraft in 2D coordinate system (Scenario 2).

be significant in PF applications. As the number of particles increases, the improvements in execution time brought about by Uphill-CA and Uphill-C1 become more pronounced. The results show us the RMSE results of each resampling algorithms are very close to each other. The bias and variance in the Uphill, Uphill-CA and Uphill-C1 resampling would not be much problem. The RMSE results of any resampling methods do not improve as the number of particles increases. This is occurred because we assume all the measurements belong to the target and we set the parameters of the filter optimum. Furthermore, since the noises are Gaussian, the particles do not diverge from the estimated position too much as the number of particles increases. We can challenge against the Systematic resampling in times with Uphill-CA $SS = 128$ and Uphill-C1 $SS = 128$ if numerical stability is the main concern.

We run resampling algorithms in another scenario. The true state of the aircraft is given in Figure 7.2. We use the same environment in the previous experiment. The simulation parameters are given in Table 7.4. The ratios of times spent by the four stages are given in Table 7.5. The RMSE results are given in Table 7.6.

The results are similar to the results in the previous experiment. Uphill compares favorably against the Systematic resampling in times with Uphill-CA $SS = 128$ and Uphill-C1 $SS = 128$ if numerical stability is the main concern.

Table 7.4: Simulation Parameters (Scenario 2).

Parameters	Value	Description
T	1	Sampling Time
x_0	$[1000\ 1000\ 0\ 0]^T$	Initial State Vector
P_0	$diag[100^2\ 100^2\ 10^2\ 10^2]$	x_0 covariance
Q	$diag[10\ 10]$	q_k covariance
R	$diag[10^2\ (0.1\pi/180)^2]$	v_k covariance

Table 7.5: The ratio of times spent in different stages (Scenario 2). The results are percentage of the total execution times. Total execution times (in seconds) are also given in the last row.

$log_2N = 14$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	13.45	5.50	8.69	10.24	13.21	14.06
Stage2	9.49	3.91	6.19	7.28	9.19	9.77
Stage3	30.84	12.88	20.29	23.91	30.26	32.16
Stage4	46.22	77.71	64.83	58.57	47.34	44.01
Exec. Time	0.064	0.151	0.096	0.081	0.066	0.062
$log_2N = 16$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	22.22	5.51	10.16	12.33	17.15	19.49
Stage2	7.35	1.82	3.36	4.07	5.67	6.44
Stage3	24.38	6.04	11.14	13.52	18.81	21.38
Stage4	46.05	86.63	75.34	70.07	58.37	52.69
Exec. Time	0.099	0.399	0.216	0.178	0.128	0.113
$log_2N = 18$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	31.50	4.85	9.05	10.71	18.71	22.81
Stage2	5.19	0.80	1.49	1.77	3.09	3.77
Stage3	18.44	2.84	5.30	6.27	10.95	13.36
Stage4	44.86	91.51	84.16	81.25	67.25	60.06
Exec. Time	0.227	1.476	0.791	0.668	0.382	0.313
$log_2N = 20$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	36.39	2.55	6.46	6.68	19.42	24.64
Stage2	4.21	0.29	0.75	0.77	2.25	2.85
Stage3	14.60	1.02	2.59	2.68	7.79	9.89
Stage4	44.80	96.13	90.21	89.86	70.54	62.62
Exec. Time	0.742	10.610	4.186	4.047	1.392	1.097
$log_2N = 22$	Systematic	Metropolis	Rejection	Uphill	UCA-128	UC1-128
Stage1	38.02	1.86	5.77	5.35	19.17	25.36
Stage2	3.60	0.18	0.55	0.51	1.82	2.40
Stage3	12.98	0.64	1.97	1.83	6.54	8.66
Stage4	45.40	97.32	91.72	92.32	72.47	63.59
Exec. Time	2.786	56.857	18.369	19.804	5.525	4.178

Table 7.6: RMSE Results (Scenario 2).

Particle No.	16384	65536	262144	1048576	4194304
Systematic	7.39098	7.37688	7.37404	7.37347	7.37318
Metropolis	7.39984	7.38456	7.38123	7.38121	7.38068
Rejection	7.39179	7.37746	7.37408	7.37330	7.37320
Uphill	7.37307	7.35783	7.35639	7.35585	7.35575
Uphill-CA-128	7.37518	7.35789	7.35567	7.35543	7.35577
Uphill-C1-128	7.40920	7.39923	7.39643	7.39652	7.39717

CHAPTER 8

CONCLUSION

We propose a new resampling algorithm for the particle filter, called the Uphill resampling, which is suitable for GPU implementation. Based on the theoretical analysis of the Uphill resampling, one can calculate the expected number of replications of each particle with respect to any B, which is the number of iterations of the inner loop. This allows us to calculate the B that yields the minimum square bias without performing an experiment. We can set the expected number of replications of any particle at the beginning by setting the value of B appropriately.

Experimental results show us that we achieve similar or better MSE results in most cases compared to the Metropolis and Rejection resampling. But the contributions of the square bias to MSE are not as small as in the Metropolis and Rejection resampling. However, this does not affect the MSE results significantly unless the relative variance in the weight sequence is large. We believe this bias would not be an issue in most tracking applications as we observe in the tracking experiment of the SIR particle filter on a highly nonlinear equation and a tracking application. The results suggest that the filtering performance of all resampling algorithms are considered similar to each other in terms of RMSE.

The execution time results of the Uphill resampling are better than those of Metropolis and Rejection in most cases. Like the Metropolis and Rejection resampling, Uphill also goes through non-coalesced global memory access patterns with a large number of particles. To ameliorate this problem, we present a memory coalesced version of the Uphill resampling, named Uphill-CA. They are same in expected number of replications of the particles, but the variance of Uphill-CA caused by s-segment and

warp size are worse than the one in the original Uphill resampling. We believe this variance would not be much of a problem in most tracking applications. Furthermore, we present a generic version of the Uphill resampling, named Uphill-C1. As the s-segment size changes, the expectation of Uphill-C1 differs. Hence, we obtain a new resampling method. Uphill is one of the new resampling method of Uphill-C1 when we set the size of s-segment as $4N$.

By comparing two weights at a time rather than calculating the cumulative sum of the weights, Uphill avoids the numerical instability issue. For a practitioner who needs a numerically stable resampling algorithm on the GPU, Uphill is advisable. However, if the bias is crucial, the other algorithms can be chosen especially when the relative variance in the weight sequence is large.

The Uphill-CA resampling is a considerable choice due to its speed. Uphill-CA has same expectation analysis with the Uphill resampling, but it is faster with a variance in the MSE results. As the s-segment size becomes large, Uphill-CA behaves more similarly to Uphill, faster in any case. When the size of s-segment is set to the size of segment of the global memory of the GPU, the minimum, Uphill-CA achieves its highest speed; however, there occurs discernible variance in the MSE results. Whether this variance constitutes a hindrance depends on the application. Uphill-CA competes favorably against the Systematic resampling when speed and numerical stability are the main concerns. The Uphill-C1 resampling is also a considerable choice due to its speed. When the size of s-segment is set to the size of segment of the global memory of the GPU, the minimum, Uphill-C1 achieves the highest speed among Uphill and Uphill-CA. Although there occurs much bias than those of Uphill and Uphill-CA caused by the s-segment size in most of the versions of it, it can be preferred if speed is the primary concern.

We also devised two techniques, designated Metropolis-C1 and Metropolis-C2, to ameliorate the non-coalesced global memory access problem of the Metropolis resampling. Experimental results indicate that C1 and C2 achieve their fastest results when the size of s-segment is chosen as the size of segment of the global memory of the GPU. By increasing the size of s-segment both techniques yield results with quality comparable to the original Metropolis resampling, at the expense of execution

time. In any case, they remain remarkably faster than the original Metropolis resampling at any comparable level of quality. Hence, C1 and C2 variations of Metropolis provide a spectrum of speed vs quality trade-off for the users.

In the future, we would like to analyze in detail the effects of the bias of the Uphill resampling on a wide variety of tracking applications. We focus on whether we can benefit from the bias of the Uphill resampling in some special cases. We would also like to investigate the issue of avoiding slow resampling process caused by large relative variances on the weight sets for Metropolis and Uphill. To achieve this, we try to reduce the parameter B adaptively and we expect to accelerate the resampling stage of the particle filter without sacrificing quality much. We also want to accelerate finding B process of the Uphill resampling with learning mechanisms.

A promising line of the study is the utility of the Uphill resampling in interacting multiple model (IMM) filters in maneuvering scenarios. IMM consists of more than one filter to handle the model mismatch scenarios in a tracking application. There are more than one filter and each runs a different model. The mean of the results of these filters constitutes the result of IMM filter. We want to investigate running more than one Uphill resampling method at the same time with different setting of B parameter.

It would be interesting to investigate the performance of multi-processor/multi-core systems, such as Intel Xeon Phi, for our proposed methods and compare them with the GPUs. Performance of multi-GPU and CPU/GPU combined systems is also worthy of future investigation. Running IMM particle filter on a single GPU or multi-GPU can be a promising line of research.

REFERENCES

[1] Alejandro Rodríguez Aguilera, Alejandro León Salas, Domingo Martín Perandrés, and Miguel A Otaduy. A parallel resampling method for interactive deformation of volumetric models. *Computers & Graphics*, 53:147–155, 2015.

[2] M Sanjeev Arulampalam, Simon Maskell, Neil Gordon, and Tim Clapp. A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

[3] Balakumar Balasingam, Miodrag Bolić, Petar M Djurić, and Joaquín Míguez. Efficient distributed resampling for particle filters. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 3772–3775. IEEE, 2011.

[4] Miodrag Bolic, Petar M Djuric, and Sangjin Hong. Resampling algorithms and architectures for distributed particle filters. *IEEE Transactions on Signal Processing*, 53(7):2442–2450, 2005.

[5] Kimiko Osada Bowman and Leonard R Shenton. *Properties of Estimators for the Gamma Distribution*. Marcel Dekker, 1988.

[6] Min-An Chao, Chun-Yuan Chu, Chih-Hao Chao, and An-Yeu Wu. Efficient parallelized particle filter design on cuda. In *Signal Processing Systems (SIPS), 2010 IEEE Workshop on*, pages 299–304. IEEE, 2010.

[7] Mehdi Chitchian, Andrea Simonetto, Alexander S van Amesfoort, and Tamás Keviczky. Distributed computation particle filters on gpu architectures for real-time control applications. *IEEE Transactions on Control Systems Technology*, 21(6):2224–2238, 2013.

[8] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU computing series. Morgan Kaufmann, 2013.

[9] Özcan Dülger, Halit Oğuztüzün, and Mübeccel Demirekler. Memory coalescing implementation of metropolis resampling on graphics processing unit. *Journal of Signal Processing Systems*, pages 1–15, 2017. https://doi.org/10.1007/s11265-017-1254-6.

[10] Peng Gong, Yuksel Ozan Basciftci, and Fusun Ozguner. A parallel resampling algorithm for particle filtering on shared-memory architectures. In *Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1477–1483. IEEE, 2012.

[11] Mark Harris. Optimizing parallel reduction in CUDA, NVIDIA developer technology [online]. `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`, 2007.

[12] Gustaf Hendeby, Jeroen D Hol, Rickard Karlsson, and Fredrik Gustafsson. A graphics processing unit implementation of the particle filter. In *Signal Processing Conference, 2007 15th European*, pages 1639–1643. IEEE, 2007.

[13] Gustaf Hendeby, Rickard Karlsson, and Fredrik Gustafsson (EURASIP Member). Particle filtering: The need for speed. *EURASIP Journal on Advances in Signal Processing*, 2010(1):181403, Jun 2010.

[14] Sangjin Hong, Shu-Shin Chin, Petar M Djurić, and Miodrag Bolić. Design and implementation of flexible resampling mechanism for high-speed parallel particle filters. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 44(1-2):47–62, 2006.

[15] Kyuyeon Hwang and Wonyong Sung. Load balanced resampling for real-time particle filtering on graphics processing units. *IEEE Transactions on Signal Processing*, 61(2):411–419, 2013.

[16] Wen-mei Hwu. A work-efficient parallel scan kernel. [online]. `http://ece408.hwu-server2.crhc.illinois.edu/Shared%20Documents/Slides/Lecture-4-6-work-efficient-scan-kernel.pdf`, 2016.

[17] Pierre E Jacob, Lawrence M Murray, and Sylvain Rubenthaler. Path storage in the particle filter. *Statistics and Computing*, 25(2):487–496, 2015.

[18] David Kirk and Wen-mei Hwu. Lectures 5 and 6: Memory model and locality. [online]. `http://ece408.hwu-server2.crhc.illinois.edu/Shared%20Documents/Slides/ece408-lecture5-6-CUDA-memory-model-2015.pptx`, 2015.

[19] Tian-cheng Li, Gabriel Villarrubia, Shu-dong Sun, Juan M Corchado, and Javier Bajo. Resampling methods for particle filtering: identical distribution, a new method, and comparable study. *Frontiers of Information Technology & Electronic Engineering*, 16(11):969–984, 2015.

[20] Tiancheng Li, Miodrag Bolic, and Petar M Djuric. Resampling methods for particle filtering: classification, implementation, and strategies. *IEEE Signal Processing Magazine*, 32(3):70–86, 2015.

[21] Shuanglong Liu, Grigorios Mingas, and Christos-Savvas Bouganis. Parallel resampling for particle filters on fpgas. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 191–198. IEEE, 2014.

[22] Lawrence M. Murray, Anthony Lee, and Pierre E. Jacob. Parallel resampling in the particle filter. *Journal of Computational and Graphical Statistics*, 25(3):789–805, 2016.

[23] NVIDIA. Tesla K40 GPU active accelerator: Board specification. [online]. `https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001_v03.pdf`, 2013.

[24] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110/210. [online]. `http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`, 2014.

[25] NVIDIA. CURAND Library: Programming guide. [online]. `http://docs.nvidia.com/cuda/pdf/CURAND_Library.pdf`, 2015.

[26] NVIDIA. CUDA C best practices guide. [online]. http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf, 2016.

[27] NVIDIA. KEPLER TUNING GUIDE.[online]. http://docs.nvidia.com/cuda/kepler-tuning-guide, 2017.

[28] NVIDIA. NVCC.[online]. http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc, 2017.

[29] NVIDIA. PROFILER.[online]. http://docs.nvidia.com/cuda/profiler-users-guide, 2017.

[30] James M. Ortega. *Numerical Analysis: A Second Course*. Academic Press, New York, 1972.

[31] Yun Pan, Ning Zheng, Qinglin Tian, Xiaolang Yan, and Ruohong Huan. Hierarchical resampling algorithm and architecture for distributed particle filters. *Journal of Signal Processing Systems*, 71(3):237–246, 2013.

[32] Branko Ristic, Sanjeev Arulampalam, and Neil Gordon. *Beyond the Kalman Filter: Particle Filters for Tracking Applications*. Artech House, 2004.

[33] Kristina M Ropella. Introduction to statistics for biomedical engineers. *Synthesis Lectures on Biomedical Engineering*, 2(1):1–94, 2007.

[34] Mahdi Shabany. An efficient architecture for sequential monte carlo receivers in wireless flat-fading channels. *Journal of Signal Processing Systems*, 68(3):303–315, 2012.

[35] Qinglin Tian, Yun Pan, Zoran Salcic, and Ruohong Huan. Dart: distributed particle filter algorithm with resampling tree for ultimate real-time capability. *Journal of Signal Processing Systems*, 88(1):29–42, 2017.

[36] Manuel Ujaldon. NEW HARDWARE FEATURES IN KEPLER, SMX AND TESLA K40.[online]. http://gpu.cs.uct.ac.za/Slides/Kepler2.pdf, 2014.

[37] Yong Wu, Jun Wang, and Yun-he Cao. Particle filter based on iterated importance density function and parallel resampling. *Journal of Central South University*, 22(9):3427–3439, 2015.

APPENDIX A

ANALYSES OF THE PROPOSED RESAMPLING ALGORITHMS

A.1 Analysis of the Uphill Resampling Algorithm

In this section, we present the theoretical analysis of the Uphill resampling method. The analysis gives us the expected number of replications of each particle or the probability of selection of each particle with respect to the B parameter.

The algorithm is analyzed by computing the expected number of replications of the particles corresponding to a given weight sequence. For the purpose of analysis, assume that it is ordered such that $\tilde{w}_1 < \tilde{w}_2 < ... < \tilde{w}_N$. This assumption does not cause loss of generality. At each iteration, the algorithm selects the index i at the beginning and selects an index $j \in \{1, .., N\}$, drawn from a uniform distribution, B times (the probability of selection of any index is $1/N$). The maximum of these indices indicates the particle that will be replicated. A particle can be selected in two non-overlapping cases:

1. The particle is selected in its own thread. That is, when the algorithm starts with the ith index, the possibility of selecting the indices less than i is eliminated; ith index is selected if B other selections are less than or equal to i. So the probability of selecting ith particle is as follows:

$$\left(\frac{i}{N}\right)^B \tag{A.1}$$

2. The particle is selected in another particle's thread. The probability of selection

of jth particle in the inner loop is calculated under the following (mutually exclusive) events:

- Among all B selections j is selected only once and others are less than j:

$$\frac{1}{N}\binom{B}{1}\left(\frac{j-1}{N}\right)^{B-1} \text{ (Notation: } \binom{B}{1} \text{ is a binomial coefficient.)} \tag{A.2}$$

- Among all B selections j is selected twice and others are less than j:

$$\frac{1}{N^2}\binom{B}{2}\left(\frac{j-1}{N}\right)^{B-2} \tag{A.3}$$

- Among all B selections j is selected k times and others are less than j:

$$\frac{1}{N^k}\binom{B}{k}\left(\frac{j-1}{N}\right)^{B-k} \tag{A.4}$$

- Among all B selections j is selected B times:

$$\frac{1}{N^B} \tag{A.5}$$

therefore the probability of selecting the jth element where $j > i$ is:

$$\frac{1}{N^B}\sum_{k=1}^{B}\binom{B}{k}(j-1)^{B-k} = \frac{1}{N^B}\left(\left(\sum_{k=0}^{B}\binom{B}{k}\alpha^{B-k}\right)-\alpha^B\right) \text{ where } \alpha = (j-1) \tag{A.6}$$

$$\frac{1}{N^B}((\alpha+1)^B-\alpha^B) \tag{A.7}$$

$$\frac{1}{N^B}(j^B-(j-1)^B) \tag{A.8}$$

The same probability occurs for each i that is less than j then the overall expectation of case 2 becomes:

$$\frac{(j-1)}{N^B}(j^B-(j-1)^B) \tag{A.9}$$

As a result the expected number of replications of jth particle is the sum of (A.1) and (A.9):

$$\left(\frac{j}{N}\right)^B + \frac{(j-1)}{N^B}(j^B - (j-1)^B) \tag{A.10}$$

Note1: The above analysis assumes that all the weights of the particles are unequal. When some of the weights are equal, the position of these weights must be the position of the one with the largest index among these weights in the weight sequence.

Note2: We do a simplification about the expected number of replications of all particles and show that the sum of their probability is equal to one below:

- $\sum_{j=1}^{N}\left(\frac{j}{N}\right)^B + \sum_{j=1}^{N}\left(\frac{(j-1)}{N^B}(j^B - (j-1)^B)\right)$
  $= \sum_{j=1}^{N}\left[\frac{\cancel{j^B}}{\cancel{N^B}} + \frac{j^{B+1}}{N^B} - \frac{\cancel{j^B}}{\cancel{N^B}} - \frac{(j-1)^{B+1}}{N^B}\right]$
- $= \frac{1}{N^B}\sum_{j=1}^{N}(j^{B+1} - (j-1)^{B+1})$
- $= \frac{1}{N^B}[\cancel{1} - 0 + \cancel{2^{B+1}} - \cancel{1^{B+1}} + 3^{B+1} - \cancel{2^{B+1}} + \ldots N^{B+1} - \cancel{(N-1)^{B+1}}]$
- $\frac{N^{B+1}}{N^B} = N$

Since the probability of drawing a particle is $\frac{1}{N}$, the overall probability is 1.

A.2 Analysis of the Uphill-CA Resampling Algorithm (Version 1)

In this section, we present the theoretical analysis of the Uphill-CA resampling method. The analysis gives us the expected number of replications of each particle or probability of each particle with respect to the B parameter. In this analysis, the actual positions of the particles in the s-segment are considered. We prove that the analysis of Uphill-CA is same with the analysis of Uphill as follows:

For each warp wp the probability of selecting a particle can be calculated in two non-overlapping cases:

1. The particle is selected in its own thread. The probability of selecting ith particle in one iteration of the inner loop is as follows:

$$\sum_{k=1}^{SC} \frac{1}{SC} \frac{p_{ik}}{DC} \tag{A.11}$$

where p_{ik} is the position of the highest element whose weight is less than or equal to $\tilde{w}_i$ in segment k (assume segment is ordered).

The probability of selecting ith particle after B iteration becomes:

$$\left(\sum_{k=1}^{SC} \frac{1}{SC} \frac{p_{ik}}{DC} \right)^B = \frac{1}{(SCDC)^B} \left(\sum_{k=1}^{SC} p_{ik} \right)^B \tag{A.12}$$

2. The particle is selected in another particle's thread. The probability of selection of jth particle in the inner loop is calculated as follows:

   - Among all B selections j is selected only once and others are less than j:

$$\frac{1}{(SCDC)} \binom{B}{1} \left(\frac{(\sum_{k=1}^{SC} p_{jk}) - 1}{(SCDC)} \right)^{B-1} \tag{A.13}$$

     where $\binom{B}{1}$ is a binomial coefficient and p_{jk} is same with p_{ik} in the first case. Lets call $\sum_{k=1}^{SC} p_{jk}$ as c.

   - Among all B selections j is selected k times and others are less than j:

$$\frac{1}{(SCDC)^k} \binom{B}{k} \left(\frac{(\sum_{k=1}^{SC} p_{jk}) - 1}{(SCDC)} \right)^{B-k} \tag{A.14}$$

   The probability of selecting jth particle becomes:

$$\frac{1}{(SCDC)^B} (c^B - (c-1)^B) \tag{A.15}$$

   The same probability occurs for each i that is less than j in warp wp then the overall expectation of case 2 becomes:

$$\frac{j_{wp}(c^B - (c-1)^B)}{(SCDC)^B} \tag{A.16}$$

   where j_{wp} is the number of i that is smaller than j in warp wp.

As a result the expected number of replications of jth particle in a warp is the summation of (A.12) and (A.16):

$$E(j,wp) = \frac{1}{(SCDC)^B}\left(\left(\sum_{k=1}^{SC} p_{jk}\right)^B + j_{wp}\left(\left(\sum_{k=1}^{SC} p_{jk}\right)^B - \left(\left(\sum_{k=1}^{SC} p_{jk}\right) - 1\right)^B\right)\right) \quad \text{(A.17)}$$

where $\sum_{j=1}^{N} E(j,wp) = WS$. Note that WS is the number of threads in a warp.

Therefore, the total expected number of replications of jth particle is $\sum_{wp=1}^{WC} E(j,wp)$ where WC represents the number of warps.

Note 1: $\sum_{j=1}^{N}\sum_{wp=1}^{WC} E(j,wp)$ is equal to N.

Note 2: p_{ik} is zero, if the ith particle $\notin$ warp wp, and $\sum_{wp=1}^{WC} j_{wp}$ is equal to $j-1$.

Note 3:$\sum_{k=1}^{SC} p_{jk}$ is equal to j and $SCDC$ is equal to N therefore the expected number of replication of jth particle is equal to $\frac{1}{N^B}(j^B + (j-1)(j^B - (j-1)^B))$. This is exactly same with the analysis of the Uphill resampling method.

A.3 Analysis of the Uphill-CA Resampling Algorithm (Version 2)

The first analysis depends on the positions of the weights in the actual weight sequence and has high computational cost. The second version of the analysis of the Uphill-CA algorithm approaches as a general way which does not depend on the positions of the weights in the actual weight sequence.

For each warp wp the probability of selecting a particle can be calculated in two non-overlapping cases:

1. The particle is selected in its own thread. The probability of selecting ith particle in one iteration of the inner loop is as follows:

$$\sum_{k=1}^{SC}\left(\frac{1}{SC}\left(\sum_{l=1}^{DC}(P1(i=l,DC)\frac{l}{DC})\right)\right) \quad \text{(A.18)}$$

where $P1(i = l, DC)$ is the probability of the order of the highest weight whose value is less than or equal to $\tilde{w}_i$ in a segment is equal to l and is calculated as $\left(\prod_{m=1}^{l} \frac{i-m+1}{N-m+1}\right)\left(\prod_{m=l+1}^{DC} \frac{N-i+l-m+1}{N-m+1}\right)\binom{DC}{DC-1}$

The probability of selecting ith particle after B iteration becomes:

$$\begin{aligned}&\left(\sum_{k=1}^{SC}\left(\frac{1}{SC}\left(\sum_{l=1}^{DC}(P1(i=l,DC)\frac{l}{DC})\right)\right)\right)^{B}\\&=\frac{1}{(SCDC)^{B}}\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(i=l,DC)l)\right)^{B}\end{aligned} \tag{A.19}$$

2. The particle is selected in another particle's thread. The probability of selection of jth particle in the inner loop is calculated as follows:

$$\frac{1}{(SCDC)^{B}}\left(\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)^{B}-\left(\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)-1\right)^{B}\right) \tag{A.20}$$

The same probability occurs for each i that is less than j in warp wp then the overall expectation of case 2 becomes:

$$\begin{aligned}&\frac{1}{(SCDC)^{B}}\left(\sum_{k=1}^{WS+1}P2(j=k,WS)(k-1)\right)\\&\left(\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)^{B}-\left(\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)-1\right)^{B}\right)\end{aligned} \tag{A.21}$$

where $P2(j = k, WS)$ is the probability of the number of weights that is less than $\tilde{w}_j$ in a warp is equal to $k-1$ and is calculated as $\left(\prod_{m=1}^{k-1} \frac{j-m}{N-m+1}\right)\left(\prod_{m=k}^{WS} \frac{N-j+k-m+1}{N-m+1}\right)\binom{WS}{WS-k+1}$ Note that WS is the number of threads in a warp.

As a result the expected number of replications of jth particle in a warp is the sum-

mation of (A.19) and (A.21):

$$E(j,wp) = \frac{1}{(SCDC)^B}\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)^B + \left(\sum_{k=1}^{WS+1} P2(j=k,WS)(k-1)\right)$$
$$\left(\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)^B - \left(\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)-1\right)^B\right)\frac{1}{(SCDC)^B} \tag{A.22}$$

where $\sum_{j=1}^{N} E(j,wp) = WS$.

Therefore, the total expected number of replications of jth particle is $\sum_{wp=1}^{WC} E(j,wp)$ where WC represents the number of warps.

Note 1: $\sum_{j=1}^{N}\sum_{wp=1}^{WC} E(j,wp)$ is equal to N.

Note 2: Case 1 occurs in a single warp and the results of case 2 are equal on each warp, for the simplicity of the computation, the expected number of replications of jth particle can be written as:

$$\frac{1}{(SCDC)^B}\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)^B + WC\left(\sum_{k=1}^{WS+1} P2(j=k,WS)(k-1)\right)$$
$$\left(\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)^B - \left(\left(\sum_{k=1}^{SC}\sum_{l=1}^{DC}(P1(j=l,DC)l)\right)-1\right)^B\right)\frac{1}{(SCDC)^B} \tag{A.23}$$

A.4 Analysis of the Uphill-C1 Resampling Algorithm (Version 1)

In this section, we present the theoretical analysis of the Uphill-C1 resampling method. The analysis gives us the expected number of replications of each particle or probability of each particle with respect to the B parameter. In this analysis, the actual positions of the particles in the s-segment are considered.

For each warp wp the probability of selecting a particle can be calculated in two non-overlapping cases:

1. The particle is selected in its own thread. For any selected segment, the probability of selecting ith particle is as follows:

$$\frac{1}{SC}\left(\frac{p_{ik}}{DC}\right)^{B} \tag{A.24}$$

where p_{ik} is the position of the highest element whose weight is less than or equal to $\tilde{w}_i$ in segment k (assume segment is ordered).

When we consider all possible segments, the probability of selecting ith particle is as follows:

$$\sum_{k=1}^{SC}\frac{1}{SC}\left(\frac{p_{ik}}{DC}\right)^{B} = \frac{1}{SC}\frac{1}{DC^{B}}\sum_{k=1}^{SC}(p_{ik})^{B} \tag{A.25}$$

2. The particle is selected in another particle's thread. The probability of selection of jth particle in the inner loop is as follows:

$$\frac{1}{SC}\left(\frac{1}{DC^{B}}\left(c_j^{B}-(c_j-1)^{B}\right)\right) \tag{A.26}$$

where c_j is the position of jth element on its own segment (assume segment is ordered).

The same probability occurs for each i that is less than j in warp wp then the overall expectation of case 2 becomes:

$$\frac{j_{wp}(c_j^{B}-(c_j-1)^{B})}{SCDC^{B}} \tag{A.27}$$

where j_{wp} is the number of i that is smaller than j in warp wp.

As a result the expected number of replications of jth particle in a warp is the summation of (A.25) and (A.27):

$$E(j,wp) = \frac{1}{SCDC^{B}}\left(\sum_{k=1}^{SC}(p_{jk})^{B} + j_{wp}(c_j^{B}-(c_j-1)^{B})\right) \tag{A.28}$$

where $\sum_{j=1}^{N} E(j,wp) = WS$. Note that WS is the number of threads in a warp.

Therefore, the total expected number of replications of jth particle is $\sum_{wp=1}^{WC} E(j, wp)$ where WC represents the number of warps.

Note 1: $\sum_{j=1}^{N} \sum_{wp=1}^{WC} E(j, wp)$ is equal to N.

Note 2: p_{ik} is zero, if the ith particle $\notin$ warp wp, and $\sum_{wp=1}^{WC} j_{wp}$ is equal to $j-1$. For the simplicity of the computation, the expected number of replications of jth particle can be written as:

$$\frac{1}{SCDC^B}\left(\sum_{k=1}^{SC}(p_{jk})^B + (j-1)(c_j^B - (c_j-1)^B)\right) \tag{A.29}$$

Note 3:When we take $SC = 1$ and $DC = N$, p_{jk} becomes j and c_j becomes j therefore the expected number of replication of jth particle becomes $\frac{1}{N^B}(j^B + (j-1)(j^B - (j-1)^B))$. This is exactly same with the analysis of the Uphill resampling method.

A.5 Analysis of the Uphill-C1 Resampling Algorithm (Version 2)

The first analysis depends on the positions of the weights in the actual weight sequence and has high computational cost. The second version of the analysis of the Uphill-C1 algorithm approaches as a general way which does not depend on the positions of the weights in the actual weight sequence.

For each warp wp the probability of selecting a particle can be calculated in two non-overlapping cases:

1. The particle is selected in its own thread. The probability of selecting ith particle is as follows:

$$\frac{1}{SC}\sum_{l=1}^{DC} P1(i=l, DC)\left(\frac{l}{DC}\right)^B \tag{A.30}$$

where $P1(i=l, DC)$ is the probability of the order of the highest weight whose value is less than or equal to $\tilde{w}_i$ in a segment is equal to l and is calculated as $\left(\prod_{m=1}^{l} \frac{i-m+1}{N-m+1}\right)\left(\prod_{m=l+1}^{DC} \frac{N-i+l-m+1}{N-m+1}\right)\binom{DC}{DC-1}$

When we consider all possible segments, the probability of selecting ith particle is as follows:

$$\sum_{k=1}^{SC}\left(\frac{1}{SC}\sum_{l=1}^{DC}P1(i=l,DC)\left(\frac{l}{DC}\right)^{B}\right)=\frac{1}{SC}\frac{1}{DC^{B}}SC\sum_{l=1}^{DC}P1(i=l,DC)l^{B} \tag{A.31}$$

2. The particle is selected in another particle's thread. The probability of selection of jth particle in the inner loop is calculated as follows:

$$\begin{aligned}&\frac{1}{SC}\sum_{l=1}^{DC}P2(j=l,DC-1)\left(\frac{l^{B}-(l-1)^{B}}{DC^{B}}\right)=\\&\frac{1}{SC}\frac{1}{DC^{B}}\sum_{l=1}^{DC}P2(j=l,DC-1)(l^{B}-(l-1)^{B})\end{aligned} \tag{A.32}$$

where $P2(j=l,DC-1)$ is the probability of the order of jth particle in its own segment is equal to l and is calculated as $\left(\prod_{m=1}^{l-1}\frac{j-m}{(N-1)-m+1}\right)\left(\prod_{m=l}^{DC-1}\frac{(N-1)-j+l-m+1}{(N-1)-m+1}\right)\binom{DC-1}{(DC-1)-l+1}$

The same probability occurs for each i that is less than j in warp wp then the overall expectation of case 2 becomes:

$$\frac{1}{SC}\frac{1}{DC^{B}}\left(\sum_{k=1}^{WS+1}P3(j=k,WS)(k-1)\right)\sum_{l=1}^{DC}P2(j=l,DC-1)(l^{B}-(l-1)^{B}) \tag{A.33}$$

where $P3(j=k,WS)$ is the probability of the number of weights that is less than $\tilde{w}_j$ in a warp is equal to $k-1$ and is calculated as $\left(\prod_{m=1}^{k-1}\frac{j-m}{N-m+1}\right)\left(\prod_{m=k}^{WS}\frac{N-j+k-m+1}{N-m+1}\right)\binom{WS}{WS-k+1}$ Note that WS is the number of threads in a warp.

As a result the expected number of replications of jth particle in a warp is the sum-

mation of (A.31) and (A.33):

$$E(j,wp) = \frac{1}{SC}\frac{1}{DC^B}\left(SC\sum_{l=1}^{DC} P1(j=l,DC)l^B\right) + \frac{1}{SC}\frac{1}{DC^B}$$
$$\left(\sum_{k=1}^{WS+1} P3(j=k,WS)(k-1)\right)\left(\sum_{l=1}^{DC} P2(j=l,DC-1)(l^B-(l-1)^B)\right) \tag{A.34}$$

where $\sum_{j=1}^{N} E(j,wp) = WS$.

Therefore, the total expected number of replications of jth particle is $\sum_{wp=1}^{WC} E(j,wp)$ where WC represents the number of warps.

Note 1: $\sum_{j=1}^{N}\sum_{wp=1}^{WC} E(j,wp)$ is equal to N.

Note 2: Case 1 occurs in a single warp and the results of case 2 are equal on each warp, for the simplicity of the computation, the expected number of replications of jth particle can be written as:

$$\frac{1}{SC}\frac{1}{DC^B}\left(SC\sum_{l=1}^{DC} P1(j=l,DC)l^B\right) + \frac{1}{SC}\frac{1}{DC^B}$$
$$WC\left(\sum_{k=1}^{WS+1} P3(j=k,WS)(k-1)\right)\left(\sum_{l=1}^{DC} P2(j=l,DC-1)(l^B-(l-1)^B)\right) \tag{A.35}$$

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Dülger, Özcan
Nationality: Turkish (TC)
Address: Department of Computer Engineering, Middle East Technical University, Üniversiteler Mahallesi, Dumlupınar Bulvarı No:1 06800 Çankaya Ankara/TURKEY
Email: odulger[at]ceng.metu.edu.tr
Phone: +90 312 2105592
Fax: +90 312 2105544
Web: `www.ceng.metu.edu.tr/~odulger`

EDUCATION

- [2010-2017] Ph.D. Computer Engineering, Middle East Technical University, Ankara, Turkey
- [2007-2009] M.S. Computer Engineering, Pamukkale University, Denizli, Turkey
- [2003-2007] B.S. Computer Engineering, Pamukkale University, Denizli, Turkey
- [1996-2003] High School Hacı Malike Mehmet Bileydi Anadolu Lisesi, Antalya, Turkey

PROFESSIONAL EXPERIENCE

- [2016-2017] Visiting Scholar: Center for Automotive Research, The Ohio State University, Columbus, Ohio, USA

- [2010-2017] Research Assistant: Computer Engineering, Middle East Technical University, Ankara, Turkey
- [2005-2007] Part Time Student: IT Department of Faculty of Medicine, Pamukkale University, Denizli, Turkey

INTERNSHIP

- [2006] Microsoft Summer School, İzmir, Turkey
- [2005] TÜBİTAK (The Scientific and Technological Research Council of Turkey) - Software Development Department, Ankara, Turkey

AWARD

- [2016–2017]: TÜBİTAK (The Scientific and Technological Research Council of Turkey) PhD 2214/A International Research Fellowship
- [2010–2015]: TÜBİTAK (The Scientific and Technological Research Council of Turkey) PhD 2211/C scholarship
- [2007–2009]: TÜBİTAK (The Scientific and Technological Research Council of Turkey) MSc 2228 scholarship

PROJECT

- RAYAP (Radar building blocks development project) project. ODTÜ DSİM project no: 09-03-01-2-00-26. Supported by ASELSAN A.Ş.., 5.7.2009 – 4.7. 2011

SUMMER SCHOOL

- Programming and Tuning Massively Parallel Systems summer school (PUMPS) at Universitat Politecnica de Catalunya, Barcelona, Spain (July 2014)

PUBLICATION

International SCI Expanded Journals

- Dülger, Ö., Oğuztüzün, H. & Demirekler, M., "Memory Coalescing Implementation of Metropolis Resampling on Graphics Processing Unit", Journal of Signal Processing Systems (2017). doi:10.1007/s11265-017-1254-6

International Journals

- Ö. Dülger, “Predicting Mathematics 1 Course Success by Using Hierarchical Adaptive Network Based Fuzzy Inference System”, Pamukkale University Journal of Engineering Sciences (PAJES), vol. 30, no. 5, pp. 166-173, 2014

International Conference Publications

- Dulger, O., Oguztuzun, H. and Demirekler, M., "Implementation of the sampling importance resampling particle filter algorithm in graphics processing unit", Signal Processing and Communications Applications Conference (SIU), 2015 23th, pp. 2195-2198, 16-19 May 2015
- Ö. Dülger, “Solving Weekly Course Timetabling Problem with Genetic Algorithm and Local Search”, The 3rd International Symposium on Computing in Science & Engineering (ISCSE 2013), 24-25 September 2013, Kuşadası, Aydın, Türkiye
- Akkus, M.A.; Karagoz, O.; Dulger, O., "Automated land reallotment using genetic algorithm," Innovations in Intelligent Systems and Applications (INISTA), 2012 International Symposium on , pp.1-5, 2-4 July 2012

National Conference Publications

- Dulger, O., Oguztuzun, H. and Demirekler, M., "Parçacık Süzgeci için Metropolis Yeniden Örnekleme Yönteminin Grafik İşleme Biriminde Gerçekleştirimi", 4. Ulusal Yüksek Başarımlı Hesaplama Konferansı (BAŞARIM), 2015 1-3 October 2015

- Ö. Dülger, “Üç Tank Sisteminin Yapay Sinir Ağı ile Modellenmesi ve Bulanık Mantık ile Kontrolü”, Otomatik Kontrol Türk Milli Komitesi 2011 Ulusal Toplantısı (TOK 2011), 14-16 September 2011, İzmir, Türkiye
- Ö. Dülger, “Project Management Optimization with Constrained Resources using Genetic Algorithms”, Yöneylem Araştırması ve Endüstri Mühendisliği 31. Ulusal Kongresi (YAEM 2011), 5-7 July 2011, Sakarya, Türkiye

Workshops

- CSW-2012 3rd Computer Science Student Workshop, Program Committee
- "Timetable Optimization with Genetic Algorithm and Local Search", CSW-2011 2nd Computer Science Student Workshop, Poster Presentation