PARALLEL IMPLEMENTATION OF RESAMPLING METHODS FOR PARTICLE FILTERING ON GRAPHICS PROCESSING UNITS

by

MATTHEW A. NICELY

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
The Department of Electrical and Computer Engineering
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2019

In presenting this dissertation in partial fulfillment of the requirements for a doctoral degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this dissertation.

Matt Nicely

Matthew A. Nicely

10/5/19

(date)

DISSERTATION APPROVAL FORM

Submitted by Matthew A. Nicely in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the dissertation committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Engineering.

Buren E. Wells 11/5/2019
Dr. Buren E. Wells (Date) Committee Chair

A Milenkovic 11/5/2019
Dr. Aleksandar Milenkovic (Date)

Laurie L. Joiner 11/5/19
Dr. Laurie L. Joiner (Date)

S Ravindran 11/8/19
Dr. Sivaguru Ravindran (Date)

11/8/19
Dr. Arie Nakhmani (Date)

Dr. Ravi Gorur (Date) Department Chair

11/12/19
Dr. Shankar Mahalingam (Date) College Dean

11/12/19
Dr. David Berkowitz (Date) Graduate Dean

ABSTRACT

School of Graduate Studies
The University of Alabama in Huntsville

Degree Doctor of Philosophy College/Dept. Engineering/Electrical and Computer Engineering

Name of Candidate Matthew A. Nicely

Title Parallel Implementation of Resampling Methods for Particle Filtering on Graphic Processing Units

Particle filter techniques are common methods used to estimate the evolving state of nonlinear, non-Gaussian time-variant systems by utilizing a periodic sequence of noisy measurements. The accuracy of particle filtering has often been shown to be superior to other state estimation techniques, such as the extended Kalman filter (EKF), for many applications. Unfortunately, the high computational cost and highly nondeterministic run-time behavior of particle filters often precludes their use in hard, real-time environments where filter response must meet the strict timing requirements of the application. Particle filter algorithms are composed of three main stages: prediction, update, and resampling.

General purpose graphics processing units (GPGPUs) have been successfully employed in previous research to accelerate the computation of both the prediction and update stages by exploiting their natural fine-grain parallelism. This research focuses on accelerating the resampling stage for GPGPU execution, which is much more difficult to parallelize due to its inherent sequentially. This dissertation introduces a novel GPGPU implementation of the systematic and stratified resampling

algorithms that exploit the monotonically increasing nature of the prefix-sum and the evolutionary nature of the particle weighting process to allow the re-indexing portion of the algorithms to occur in a two phase, multi-threaded manner. This resulting measured factor of performance improvement for the systematic and stratified algorithms, when executed on a GPGPU, was 16x and 33x, respectively, over the serial implementations.

This dissertation also describes the steps taken to optimize the naive parallel implementation to mitigate limitations such as memory dependencies stalls, low thread execution efficiency, and sub-optimal occupancy. These steps range from minimizing global memory accesses, utilizing shared memory, manually coding software prefetching, and grid-stride looping. In all, the optimizations provide a 2x - 4x improvement to execution times over the naive implementation of the resampling methods. Mathematical analysis is provided to describe algorithm efficiencies regarded to time, work, and space. L'Hôpital's rule shows that while time complexity can be reduced from $\mathcal{O}(N) \rightarrow \mathcal{O}(1)$, for the best case scenario, it costs double the amount of work. Lastly, multiple pseudorandom number generators (PRNGs) are evaluated with different implementation techniques to investigate statistical quality and timing performance as it pertains to particle filters.

Abstract Approval: Committee Chair

Dr. Buren E. Wells

Department Chair

Dr. Ravi Gorur

Graduate Dean

Dr. David Berkowitz

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my committee chair, Dr. Buren E. Wells, who has provided continuous guidance and support during my dissertation research over the past several years. I am sincerely grateful to Dr. Aleksander Milenkovic for advising me to continue researching parallel programming on embedded platforms. That guidance created the foundation that eventually led to this research. Also, I would like to thank Dr. Laurie Joiner for her assistance when I began graduate school and her continued support during my years at UAH. Lastly, a very special thanks to Ms. Jacqueline Siniard, who has been there for me any time I have had questions or issues in the course of the graduate process.

I will be eternally grateful to Dr. Thomas Kelly, who counseled me daily for five years during my PhD research. Whether listening to me complain over testing results or advising me through research roadblocks, I sincerely believe I would not have made it to where I am today without his help.

Finally, I would like to thank James Ruf, who helped me build my first computer in high school. It was that computer and the time he spent teaching me everything he knew that sent me down the path of becoming an engineer. Thanks to him I am working in my dream career.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

LIST OF TABLES

TABLE PAGE

LIST OF ACRONYMS

API	Application Programming Interface
ARM	Advanced RISC Machine
BLAS	Basic Linear Algebra Subprograms
BPF	Bootstrap Particle Filter
BSP	Board Support Package
CPU	Central Processing Unit
CTA	Cooperative Thread Array
CUB	CUDA UnBound
CUDA	Compute Unified Device Architecture
DL	Deep Learning
DSP	Digital Signal Processor
EKF	Extended Kalman Filter
ESS	Effective Sample Size
FFN	FinFET NVIDIA
FMA	Fuse-Multiply-Add
FPGA	Field-Programmable Gate Array
GPGPU	General Purpose Graphics Processing Unit

GPU	Graphics Processing Unit
HBM2	High Bandwidth Memory 2
IDE	Integrated Development Environment
ILP	Instruction Level Parallelism
IS	Importance Sampling
LFSR	Linear-Feedback Shift Register
MC	Monte Carlo
MCMC	Markov Chain Monte Carlo
MISE	Mean Integrate Square Error
MPF	Marginalized Particle Filter
PDF	Probability Density Function
PRAM	Parallel Random-Access Machine
PRNG	Pseudorandom Number Generator
QRNG	Quasirandom Number Generator
RNG	Random Number Generator
RAW	Read-After-Write
RBPF	Rao-Blackwellized Particle Filter
RMSE	Root Mean Squared Error
RPF	Regularized Particle Filter

SDK	Software Development Kit
SFU	Special Function unit
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SIR	Sequential Importance resampling
SIS	Sequential Importance Sampling
SM	Streaming Multiprocessor
SOC	System-on-a-Chip
SSL	Secure Sockets Layer
TLP	Thread Level Parallelism
WAR	Write-After-Read
WAW	Write-After-Arite

LIST OF SYMBOLS

SYMBOL	DEFINITION
t	Time
$\mathbf{x}_{t+1}$	Progated State Variable
$\mathbf{y}_t$	Update Measurement
$\mathbf{f}_t$	Arbitrary Nonlinear System Model
$\mathbf{h}_t$	Arbitrary Nonlinear Measurement Model
$\mathbf{w}_t$	Process Noise
$\mathbf{e}_t$	Measurement Noise
$\hat{\mathbf{x}}_{t\|t-1}$	Updated (*a priori*) State Estimate
$\mathbf{P}_{t\|t-1}$	Updated (*a priori*) Estimate Covariance
$\hat{\mathbf{x}}_{t\|t}$	Predicted (*a posteriori*) State Estimate
$\mathbf{P}_{t\|t}$	Predicted (*a posteriori*) Estimate Covariance
$\mathbf{F}_t$	Linear System Model
$\mathbf{H}_t$	Linear Measurement Model
$\mathbf{Q}_t$	Process Noise Covariance
$\mathbf{R}_t$	Measurement Noise Covariance
$\mathbf{K}_t$	Kalman Gain
$\mathbf{S}_t$	Innovation Covariance

To my family, who has been a source of inspiration throughout my life.

If you're not prepared to be wrong, you'll never come up with anything original.

—Ken Robinson

CHAPTER 1

INTRODUCTION

Research is what I'm doing
when I don't know what I'm doing.

—Wernher von Braun

Over the past 15 years, graphics processing units (GPUs) have become fundamental tools in our everyday lives. While their initial purpose was fueled by high computing demands in the gaming industry, they have found an equally important role in the field of accelerated scientific computing and are commonly known as general purpose graphics processing units (GPGPUs). Now, with their applicability in the future of autonomous vehicles, GPUs are being thrust into the realm of embedded systems. In an embedded environment, a heterogeneous system-on-a-chip (SOC) with a central processing unit (CPU) and GPU can be used to balance the workload for optimal efficiency. This opens the door for engineers to find new and innovative ways to solve problems and optimize current solutions.

One such area of interest is location estimation, where an object's location is being estimated from a set of noisy measurements. Because a perfect location sensor does not exist, Bayesian filter techniques can be used to give an estimate of a system's state and the error between measurement updates. State estimation is

a process commonly used to determine the underlying behavior at any point. It can use telemetry streaming from on-board sensors to estimate continuous system parameters. This information can be used to improve control actions and identity failing components. Perhaps the most well-known algorithm for state estimation is the Kalman filter [1]. Given the right conditions, such as a linear, Gaussian system, it can produce an exact, analytical solution. In most real world applications, linear and Gaussian assumptions do not hold true. Therefore, alternative methods must be used. The most common alternative is the extended Kalman filter (EKF) which is a derivation of the Kalman filter. The EKF linearizes system and measurement equations through Taylor Series expansion [2, 3] and is popular for on-line system calculations due to its ease of use and computation efficiency. One of its limitations, however, is that it works well only with weak nonlinearities since it is based on local linear approximation. Thus, for highly nonlinear systems, the EKF is not an optimum estimator [4]. One solution might be to upgrade the system with on-board sensors that produce less error measurement data, but this is not always a viable answer.

1.1 Problem Formulation

Particle filters are a set of Monte Carlo algorithms used to solve filtering problems arising in signal processing and Bayesian statistical inference. They have an advantage in that they have the ability to represent arbitrary probability densities, even in nonlinear, non-Gaussian dynamic systems, and can produce probability densities with higher accuracy as the number of particles increase. Because each additional particle only requires an additional Monte Carlo evaluation, the algorithm can take

advantage of the abundant resources available on modern GPUs. Today's GPUs have anywhere from a few hundred to several thousand cores per processor. Each core particle filter is based on three operations: 1) prediction, 2) updating, and 3) resampling. The prediction and updating steps are computationally intensive, but can be easily implemented in parallel if hardware is available. The resampling step, however, is not naturally suited for parallel processing and execution time is dependent on the number of particles rather than the number of state dimensions. Resampling is an essential step in particle filtering because it replaces the particles whose weights have little significance with new particles that are judiciously placed within the solution space. Traditionally, this is done using collective operations across particles and weights in a serial manner, due to data dependencies. Not only does this cause a bottleneck in the particle filtering process, but if the rest of the algorithm has been implemented on a GPU it will require costly data copies between the GPU and CPU memory banks.

1.2 Contributions

This dissertation makes contributions in the following areas:

1.2.1 Parallel Systematic/Stratified Resampling

A novel parallel approach to systematic [5] and stratified [6] resampling is presented that allows full implementation on GPU architecture while producing a resampling index that is an exact match to the traditional serial method. A first-cut implementation is provided as a baseline and to aid initial understanding. Then, an

improved implementation that reduces warp stalls caused by memory dependencies is provided in detail. The optimized approach contributes to speedups of 16x and 33x for systematic and stratified methods, respectively. Mathematical analysis is provided to evaluate algorithmic efficiencies such as time, work, and space.

1.2.2 Random Number Generation Requirements

This paper presents Monte Carlo techniques that utilize parallel processing on GPUs. To ensure statistical quality, along with optimal performance, various implementation techniques are examined against multiple random number generators (RNGs) that are provided in the NVIDIA cuRAND library. Years of research have gone into developing RNGs that are highly tuned for CPUs utilizing large instruction sets and large amounts of fast memory available to the core(s). These architecture characteristics are not available for modern GPUs, requiring modifications to be made for parallel implementations. This dissertation examines multiple programming implementations across multiple RNGs to determine the optimal RNG for the parallel resampling method.

1.2.3 Parallelization Strategies

Multiple Compute Unified Device Architecture (CUDA) coding techniques are compared for portability, maintainability, and scalability on GPU hardware. This dissertation investigates optimization techniques such as coalesced global memory access, utilizing shared memory storage, software prefetching, grid-stride looping, and cooperative thread arrays. These techniques are used to mitigate common GPU pro-

gramming limitations. In particular, memory dependency stalls, low thread execution efficiency, and sub-optimal occupancy. Overall, speedups of 2x - 4x are achieved across a range of data sets, containing between 2^{10} and 2^{20} elements, respectively. Lastly, multiple streams are utilized to enhance GPU performance when individual kernel performance decreases due to the tail effect.

1.3 Outline

The remainder of this dissertation is organized as follows: Chapter 2 delineates a basic understanding of particle filtering along with pseudocode and algorithmic models of common particle filters and current resampling techniques. Chapter 3 provides background information about GPUs, starting with the programming model and the multiple types of memory within the memory model. It also highlights popular NVIDIA libraries and tools which were evaluated during this research. Chapter 4 provides details on the novel parallel approach to systematic and stratified resampling and optimization techniques to overcome common GPU limitations. This chapter also provides the mathematical analysis of multiple algorithmic efficiencies. Techniques and limitations of implementing RNGs on parallel architectures such as GPUs are described in Chapter 5. In Chapter 6, results on error and accuracy for a 4-state bootstrap particle filter (BPF) benchmark, by Schön [7], are provided in conjunction with execution performance across multiple data sets of variable variances. Finally, in Chapter 7, conclusions and suggestions for future research are provided.

CHAPTER 2

PARTICLE FILTERS

Particle filtering is a Monte Carlo method for performing Bayesian inference in state-space models as the system state evolves through time based on input measurement updates. Particle filtering techniques are used in a wide range of fields including navigation [7], image and signal processing [8, 9], robotics [10], economics [11], and self localization [12]. Particle filters utilize statistical inference, which uses mathematics to draw conclusions in the presence of uncertainty using quantitative data. In the case of particle filters, uncertainty lies within the current state and the quantitative data of the incoming measurements. Both the system and measurement data have contributing random noise. Particle filters apply a technique called recursive Bayesian filtering and are adept to solve Hidden Markov Chain and nonlinear filtering problems. The Bayesian approach is to construct the posterior probability density function (PDF) of the state based on all available information. For most nonlinear, non-Gaussian problems there are no general analytic expressions for the desired PDF. Therefore, particle filtering is more of an approximation of a complex model rather than an exact representation of a simplified model. Two assumptions are used to derive a recursive Bayes filter: 1) the states follow a first-order Markov process 2) the

observations are independent of the given states. This dissertation uses the following notation: bold capital letters for matrices, and bold lowercase letters for vectors.

The general dynamic system consists of a system model along with a measurement model. Both are defined below:

$$\mathbf{x}_{t+1} = \mathbf{f}_t(\mathbf{x}_t, \mathbf{w}_t) \tag{2.1}$$

$$\mathbf{y}_t = \mathbf{h}_t(\mathbf{x}_t, \mathbf{e}_t) \tag{2.2}$$

Here $\mathbf{x}_{t+1}$ is the propagated state variable at time t, $\mathbf{y}_t$ is the update measurement, $\mathbf{w}_t$ is the process noise, $\mathbf{e}_t$ is the measurement noise, and $\boldsymbol{f}$,$\boldsymbol{h}$ are two arbitrary nonlinear functions. The noise densities are independent and are assumed to be known. In a Bayesian setting there is a two-step framework, prediction and update.

The prediction step, or *a priori*, $p(\mathbf{x}_t|\mathbf{y}_{1:t-1})$ is computed from the filtering distribution $p(\mathbf{x}_{t-1}|\mathbf{y}_{1:t-1})$ at time $t-1$.

$$p(\mathbf{x}_t|\mathbf{y}_{1:t-1}) = \int \overbrace{p(\mathbf{x}_t|\mathbf{x}_{t-1})}^{\substack{\text{system}\\ \text{model}}} \overbrace{p(\mathbf{x}_{t-1}|\mathbf{y}_{1:t-1})}^{\substack{\text{previous}\\ \text{posterior}}} \, d\mathbf{x}_{t-1} \tag{2.3}$$

where $p(\mathbf{x}_{t-1}|\mathbf{y}_{t-1})$ is assumed known due to recursion and $p(\mathbf{x}|\mathbf{x}_{t-1})$ is given by Equation 2.1. During the update step, or *a posteriori*, the *a priori* is updated with

new measurement data $\mathbf{y}_t$.

$$p(\mathbf{x}_t|\mathbf{y}_{1:t}) = \frac{\overbrace{p(\mathbf{y}_t|\mathbf{x}_t)}^{\text{measurement model}} \overbrace{p(\mathbf{x}_t|\mathbf{y}_{1:t-1})}^{\text{current prior}}}{\underbrace{p(\mathbf{y}_t|\mathbf{y}_{1:t-1})}_{\text{normalization constant}}} \tag{2.4}$$

This chapter is organized as follows. Section 2.1 discusses the earliest particle filter, BPF, and its limitations. Section 2.2 and Section 2.3 cover alternative schemes that focus on alleviating issues like *weight degeneracy* and *sample impoverishment*. Section 2.4 covers common resampling techniques, such as Multinomial, Systematic, Stratified, Residual, Metropolis, and Rejection.

2.1 Bootstrap Particle Filter

The BPF was introduced in 1993 by Gordon [4]. The key idea is to represent the PDF by random samples, or particles, with associated weights and to compute estimates based on these samples and weights. The basic building block of particle filters is importance sampling (IS). In IS, one approximates a target distribution $p(\mathbf{x})$ using samples drawn from a proposal distribution $q(\mathbf{x})$. To compensate between the two, a weight is added to each sample. IS yields

$$p(\mathbf{x}_{0:t}|\mathbf{y}_{1:t}) \approx \sum_{i=1}^{N} w_t^i \delta(\mathbf{x}_{0:t} - \mathbf{x}_{0:t}^i) \tag{2.5}$$

where N is the number of samples and δ is the delta function centered at $\mathbf{x}_{0:k-1}^i$. The process of running IS at each time step is called sequential importance sampling (SIS).

The purpose of SIS is to update the particles and weights such that they approximate the posterior distribution at the next time step. As $N \to \infty$, the approximation (2.5) approaches the true posterior density. Weight updates are calculated as follows:

$$\mathbf{x}_t^i \approx q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{y}_t) \tag{2.6}$$

$$w_t^i \propto w_{t-1}^i \frac{p(\mathbf{y}_t|\mathbf{x}_t^i)p(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i)}{q(\mathbf{x}_t^i|\mathbf{x}_{t-1}^i, \mathbf{y}_t)} \tag{2.7}$$

where $q(\cdot)$ is the importance density and then normalized to sum 1.

SIS suffers from an unavoidable problem known as weight degeneracy, where after a few iterations most particles have negligible weights. One way to help degeneracy is to increase the number of particles, but that is impractical for most problems due to computational workload. Degeneracy can be measured by estimating the effective sample size (ESS), which is calculated as follows:

$$\hat{N}_{eff} = \frac{1}{\sum_{i=1}^{N}(w_t^i)^2} \tag{2.8}$$

Small $\hat{N}_{eff}$ indicates severe degeneracy. The most common and effective way to mitigate high degeneracy is *resampling*. The idea is to eliminate particles with low importance weights and replicate particles with high importance weights. This, in combination with SIS, is often referred to as sequential importance resampling (SIR). A BPF is described in Algorithm 1.

Algorithm 1 Generic Particle Filter

1: **for** $i \leftarrow 1, N$ **do**
2: Initialization $\mathbf{x}_t^i \sim q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{y}_t)$
3: Compute importance weights, w_k^i: per (2.7)
4: **end for**
5: Calculate total weight: $\sum_{i=1}^{N} w_t^i$
6: **for** $i \leftarrow 1, N$ **do**
7: Normalize weights
8: **end for**
9: Compute state estimates
10: Calculate ESS ($\hat{N}_{eff}$): per (2.8)
11: **if** $\hat{N}_{eff} < N_T$ **then**
12: Resample ▷ Method may vary
13: **end if**
14: **for** $i \leftarrow 1, N$ **do**
15: Perform particle transition
16: **end for**

2.2 Marginalized Particle Filter

Computational complexity of a particle filter escalates quickly as the number of state dimensions increase. This is sometimes referred to as the *curse of dimensionality*. One countermeasure, using Bayes theorem, is to marginalize linear state variables and solve them using a Kalman filter [1]. Kalman filters are the optimal filters for these linear states and should provide better estimates. Nonlinear state variables are then solved using particle filters. In some cases, where state variables are *weakly nonlinear*, an EKF [13] may be used. An EKF linearizes a state variable about a working point. This hybrid method is called the marginalized particle filter (MPF) [7] or Rao-Blackwellized particle filter (RBPF) [14]. The MPF encompasses the BPF along with the update and prediction steps of a Kalman filter. For the

Kalman filter, equations (2.1) and (2.2) can be rewritten as follows:

$$\mathbf{x}_{t+1} = \mathbf{F}_t \mathbf{x}_t + \mathbf{w}_t \tag{2.9}$$

$$\mathbf{y}_t = \mathbf{H}_t \mathbf{x}_t + \mathbf{e}_t \tag{2.10}$$

where $\mathbf{F}_t$ and $\mathbf{H}_t$ are known matrices defining linear functions. Covariances of $\mathbf{w}_t$ and $\mathbf{e}_t$ are, $\mathbf{Q}_t$ and $\mathbf{R}_t$, respectively. Kalman filter prediction equations

$$\hat{\mathbf{x}}_{t|t-1} = \mathbf{F}_t \hat{\mathbf{x}}_{t-1|t-1} \tag{2.11a}$$

$$\mathbf{P}_{t|t-1} = \mathbf{F}_t \mathbf{P}_{t|t-1} \mathbf{F}_t^T + \mathbf{Q}_t \tag{2.11b}$$

are the *a priori* state estimation and the *a priori* estimation covariance, respectively. Kalman filter update equations are listed below, where $\mathbf{S}_t$ is the innovation covariance, $\mathbf{K}_t$ is the Kalman gain, $\hat{\mathbf{x}}_{t|t}$ is the *a posteriori* state estimation, and $\mathbf{P}_{t|t}$ is the *a posteriori* estimation covariance.

$$\mathbf{S}_t = \mathbf{H}_t \mathbf{P}_{t|t-1} \mathbf{H}_t^T + \mathbf{R}_t \tag{2.12a}$$

$$\mathbf{K}_t = \mathbf{P}_{t|t-1} \mathbf{H}_t^T \mathbf{S}_t^{-1} \tag{2.12b}$$

$$\hat{\mathbf{x}}_{t|t} = \hat{\mathbf{x}}_{t|t-1} + \mathbf{K}_t (\mathbf{y}_t - \mathbf{H}_t \hat{\mathbf{x}}_{t|t-1}) \tag{2.12c}$$

$$\mathbf{P}_{t|t} = \mathbf{P}_{t|t-1} - \mathbf{K}_t \mathbf{H}_t \mathbf{P}_{t|t-1} \tag{2.12d}$$

An MPF is described in Algorithm 2. Once marginalized, a linear system will be formed for each particle and estimated by a Kalman filter. By solving linear states

in this fashion, some variance can be removed from the nonlinear states. For further accuracy improvements, the nonlinear estimates can be used as an additional measurement in the Kalman filter timing update. While the workload per particle is greater for the MPF, it has been shown by Schön [7] that only a portion of the particles required for a BPF are needed; in turn, minimizing the computational time of the resampling step which is performed every timestep.

Algorithm 2 Marginalized Particle Filter

1: **for** $i \leftarrow 1, N$ **do**
2: Initialization $\mathbf{x}_t^i \sim q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{y}_t)$
3: Compute importance weights, w_k^i: per (2.7)
4: **end for**
5: Calculate total weight: $\sum_{i=1}^{N} w_t^i$
6: **for** $i \leftarrow 1, N$ **do**
7: Normalize weights
8: **end for**
9: Compute nonlinear state estimates
10: Resample N particles and covariance matrices
11: **for** $i \leftarrow 1, N$ **do**
12: Perform Kalman filter: measurement update
13: **end for**
14: Compute linear state estimates
15: **for** $i \leftarrow 1, N$ **do**
16: Perform particle transition ▷ Only nonlinear states
17: **end for**
18: **for** $i \leftarrow 1, N$ **do**
19: Perform Kalman filter: timing update
20: **end for**

2.3 Regularized Particle Filter

Systems with little to no process noise resampling can introduce sample impoverishment, which is the loss of diversity among particles. In the most severe case, all particles can occupy the same state-space position. A modified version of the

BPF is proposed in [15], called a regularized particle filter (RPF). It is identical to the BPF with the exception of the resampling step. It samples from a continuous approximation rather than a discrete distribution (2.5). The approximation is

$$p(\mathbf{x}_t|\mathbf{y}_{1:t}) \approx \sum_{i=1}^{N} w_t^i K_h(\mathbf{x}_t - \mathbf{x}_t^i) \tag{2.13}$$

where

$$K_h(\mathbf{x}) = \frac{1}{h^{n_x}} K\left(\frac{\mathbf{x}}{h}\right) \tag{2.14}$$

is the rescaled kernel density $K(\cdot)$ and h is the kernel bandwidth, which must be positive. n_x denotes the dimension of $\mathbf{x}$ and w_t^i is the normalized particle weight. Now samples are drawn from an empirical representation that closely matches the true posterior density. This is accomplished by choosing a kernel and bandwidth to minimize the mean integrate square error (MISE) between the two. In the special case of a classical equally weighted sample $w_t^i = 1/N$ for $i = 1, \ldots, N$, the density estimation theory [16,17] provides the optimal choice for the kernel as the Epanechnikov kernel [15]. It is defined as follows:

$$f(n) = \begin{cases} \dfrac{n_x + 2}{2c_{n_x}}(1 - ||\mathbf{x}||^2), & \text{if } ||\mathbf{x}|| < 1 \\ 0, & \text{otherwise} \end{cases} \tag{2.15}$$

where c_{n_x} is the volume of the unit sphere $\mathbb{R}^{n_x}$. The choice of the optimal bandwidth h_{opt} follows as [15]

$$h_{opt} = AN^{1/(n_x+4)} \tag{2.16}$$

$$A = [8c_{n_x}^{-1}(n_x+4)(2\sqrt{\pi})^{n_x}]^{1/(n_x+4)} \tag{2.17}$$

In an real-world environment this special case can be used to find a sub-optimal filter. In practical scenarios, the regularized particle filter (RPF) performance is better than the BPF in cases where sample impoverishment is severe, such as when the process noise is small [18]. An RPF is described in Algorithm 3.

Algorithm 3 Regularized Particle Filter

for $i \leftarrow 1, N$ **do**
    Initialization $\mathbf{x}_t^i \sim q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{y}_t)$
    Compute importance weights, w_k^i: per (2.7)
end for
Calculate total weight: $\sum_{i=1}^{N} w_t^i$
for $i \leftarrow 1, N$ **do**
    Normalize weights
end for
Compute state estimates
Calculate ESS ($\hat{N}_{eff}$): per (2.8)
if $\hat{N}_{eff} < N_T$ **then**
    Calculate empirical covariance matrix S_t
    Compute D_t such that $D_t D_t^T = S_t$
    Resample ▷ Method may vary
    for $i \leftarrow 1, N$ **do**
        Draw from Epanechnikov Kernel
    end for
end if

2.4 Resampling Techniques

While early forms of particle filters offered an alternative for nonlinear, non-Gaussian state estimation, it was the introduction of the resampling step [4] that made them a viable option in real-world applications. Before resampling, particle weights would be updated in a iterative manner as the next observation became available. With no method to discard weights with low or no discernible effect, the variance between particle weights would increase with time [19]. After a series of iterations, most particles retain a negligible weight and resources would be wasted propagating useless particles. A naive approach is to add an exorbitant amount of particles because the computational workload makes it impractical for most applications.

Two methods are often used to alleviate the effect of degeneracy: choosing an optimal importance density function and resampling. An optimal importance function helps minimize the variance of particle weight. The optimal importance function [20] is shown as follows:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{y}_{0:t}) = p(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{y}_t) \tag{2.18}$$

$$= \frac{p(\mathbf{y}_t|\mathbf{x}_t, \mathbf{x}_{t-1}^i)\ p(\mathbf{x}_t|\mathbf{x}_{t-1}^i)}{p(\mathbf{y}_t|\mathbf{x}_{t-1}^i)} \tag{2.19}$$

Unfortunately, one must be able to sample from $p(\mathbf{x}_t|\mathbf{x}_{t-1}^i, \mathbf{y}_t)$ and calculate $p(\mathbf{y}_t|\mathbf{x}_{t-1}^i)$, which is often not possible [21]. Therefore, a suboptimal importance function must be chosen. A good choice is the transitional *a priori*, $p(\mathbf{x}_t|\mathbf{x}_{t-1}^i)$. Fine tuning the

importance density for a given problem will yield an appropriate trade-off between the number of particles and computational expense to process each particle [18].

The second approach to improve weight degeneracy is resampling. Since its conception, resampling has proven to be beneficial both practically and theoretically [10]. Resampling is performed by removing particles with small weights and replacing them with neighboring particles with high weights. Once the remaining particles have been redistributed, all weights are set to a constant value $1/N$. A visual aid is provided in Figure 2.1. Traditionally, resampling is executed when the ESS has dropped below a given threshold, N_T. The threshold is expressed as a proportion of the number of particles and is sometimes defaulted to 50% [22]. This is because most resampling methods are serial by nature and create a bottleneck. Historically, this has caused programmers to balance performance and accuracy.

It is important to note that while resampling reduces degeneracy, it introduces sample impoverishment when system process noise is small. The particles lose diversity by removing all negligible particle weights and repeatedly replicating high particle weights. They are no longer statistically independent. This approach is called total sampling [23], where the resulting particle set only contains new-born particles. One technique to solve this problem is the resample-move algorithm described by Gilks [24]. The resample-move algorithm has a move step after the resampling step based on Markov chain Monte Carlo (MCMC) sampling. The move step is performed on each particle to rejuvenate diversity. Other approaches to alleviate sample impoverishment can be found in [23, 25, 26]. The following sections cover some common unbiased resampling techniques in literature such as multinomial [4], resid-

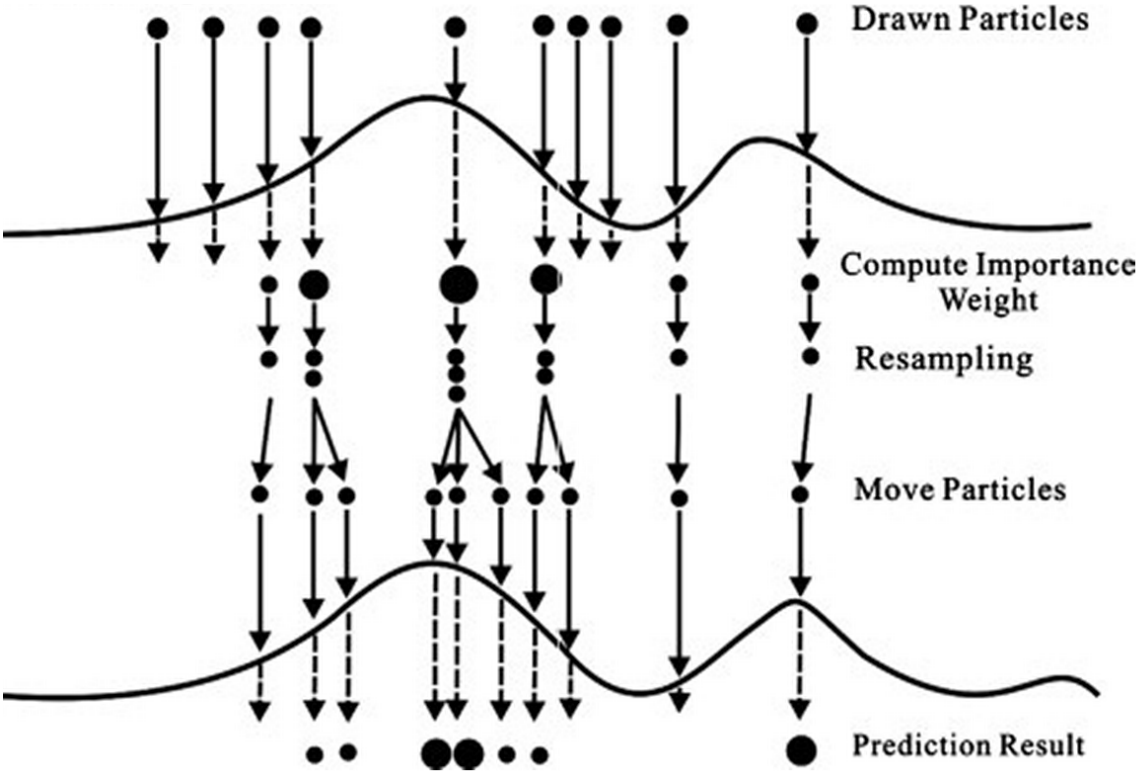


Figure 2.1: Traditional resampling

ual [27], stratified [5], and systematic [6]. The parallel resampling techniques known as Metropolis and Rejection [28] are also discussed.

2.4.1 Multinomial

Multinomial resampling was the first method to be paired with SIS to produce a BPF [4]. It requires generating N ordered random numbers of uniform distribution. Those numbers are then used to select particles from the approximation filtering distribution used to represent $p(\mathbf{x}_t|\mathbf{y}_{1:t})$ [29]. Particles are chosen when they meet the

following condition

$$Q_t^{m-1} < u_t^n \leq Q_t^m \tag{2.20}$$

where Q_t^m is the cumulative sum of all particle weights up to particle m. This approach is inefficient and provides unsatisfactory Monte Carlo (MC) variations between particle weights [6]. The computational complexity of multinomial resampling is of order $\mathcal{O}(NK)$, where the K factor arises from the search of the required k at line (6) in Algorithm 4.

Algorithm 4 Multinomial Resampling

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count (w)
2: $c \leftarrow$ Inclusive-Prefix-Sum (w)
3: **for** $i \leftarrow 1, N$ **do**
4: $\quad u \sim U[0, 1)$;
5: $\quad k \leftarrow 1$
6: $\quad$ **while** $c(k) < u$ **do**
7: $\quad\quad k \leftarrow k + 1$
8: $\quad$ **end while**
9: $\quad idx(i) \leftarrow k$
10: **end for**

2.4.2 Stratified

Stratified resampling was first proposed by Kitagawa [5]. In the algorithm, it is assumed that division into strata, or layers, is performed. In each stratum, resampling can be executed simultaneously since random numbers are drawn independently. The stratified method has a complexity of $\mathcal{O}(N)$ due to the iteration of the for-loop at line (5) in Algorithm 5.

Algorithm 5 Stratified Resampling

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count (w)
2: $c \leftarrow$ Inclusive-Prefix-Sum (w)
3: $u \leftarrow ((n-1) + u_n)/N$ ▷ $u_n \sim U[0,1)$; $n = 1, \ldots, N$
4: $k \leftarrow 1$
5: **for** $i \leftarrow 1, N$ **do**
6: **while** $c(k) < u(i)$ **do**
7: $k \leftarrow k + 1$
8: **end while**
9: $idx(i) \leftarrow k$
10: **end for**

2.4.3 Systematic

Systematic resampling [6] is very similar to stratified except that it tries to reduce particle discrepancy by choosing the strata and the number of samples more effectively. This is achieved by choosing only one uniform random number and adding it to the entire ordered set. The samples are no longer independent and are at the same position in the stratum, shown by the generation of a single random number at line (3) of Algorithm 6. Systematic also has a complexity of $\mathcal{O}(N)$. It is more efficient and often the preferred method due to its simplistic implementation and its ability to minimize MC variation.

2.4.4 Residual

Residual sampling [27] consists of two stages. First, particle weights, M, that are greater than $1/N$ are deterministically replicated. The remaining particles, or the residuals, $R = N - \sum_{m=1}^{M} N_t^{(m)}$, are selected using one of the previously mentioned resampling steps during stage two. The first stage represents a deterministic repli-

Algorithm 6 Systematic Resampling

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count (w)
2: $c \leftarrow$ Inclusive-Prefix-Sum (w)
3: $u \leftarrow ((n-1) + u_0)/N$ ▷ $u_0 \sim U[0, 1)$
4: $k \leftarrow 1$
5: **for** $i \leftarrow 1, N$ **do**
6: **while** $c(k) < u(i)$ **do**
7: $k \leftarrow k + 1$
8: **end while**
9: $idx(i) \leftarrow k$
10: **end for**

cation so that the variation of the number of times a particle is resampled is only attributed to the second stage [29]. Therefore, the complexity is $\mathcal{O}(M) + \mathcal{O}(R)$, or $\mathcal{O}(N)$. Algorithm 7 shows pseudocode for Residual resampling.

Algorithm 7 Residual Resampling

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count (w)
2: **for** $i \leftarrow 1, N$ **do**
3: $N_s(i) \leftarrow floor(N * w(i))$
4: **end for**
5: **for** $j \leftarrow 1, N$ **do**
6: **for** $k \leftarrow 1, N_s(j)$ **do**
7: $idx(i) \leftarrow j$
8: $i \leftarrow i + 1$
9: **end for**
10: **end for**
11: $R = N - sum(N_s)$ ▷ Remaining particle for stage 1
12: **for** $i \leftarrow 1, N$ **do**
13: $w_s \leftarrow (N * w(i) - N_s(i))/R$
14: **end for**
15: $c \leftarrow$ Inclusive-Prefix-Sum (w_s)
16: **Execute** MULTINOMIAL/STRATIFIED/SYSTEMATIC

2.4.5 Metropolis

Metropolis resampling was introduced as a way to accelerate particle filters with the assistance of GPUs through parallelization [28]. Like multinomial and stratified, it requires the creation of a relatively large set of uniformly distributed random numbers. With Metropolis, weights are not summed cumulatively or normalized. Therefore, it removes dependency between weights and improves parallelization. Figure 2.2 provides a visual aid comparing techniques where arcs along the perimeter of the circles represent particles by weight. Arrows indicate selected particles and are positioned (a) uniformly random in multinomial resampling, (b & c) by evenly slicing the circle into strata and randomly selecting an offset (stratified resampling) or using the same offset (systematic resampling) into each stratum, or (d) initializing multiple Markov chains and simulating to convergence in Metropolis resampling [28].

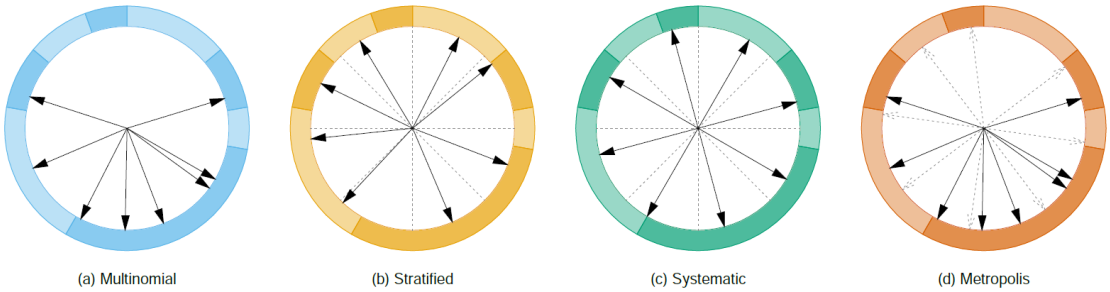


Figure 2.2: Visualization of resampling methods [28].

The concept is to execute N threads in parallel over B iterations comparing randomly selected weights. If the ratio of a randomly selected particle and the previous particle is greater than a uniform random number, then it shall be selected

for further comparisons. See line (8) in Algorithm 8. However, if the ratio of the randomly selected particle is smaller, it shall be discarded. After B comparisons, the current particle is passed for replication. Careful consideration must be taken when choosing B; if B is too small, the sample size will be more biased and possibly not converge. The selection of B is a trade-off of performance and accuracy. Algorithm 8 shows pseudocode for Metropolis resampling.

Algorithm 8 Metropolis Resampling

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count (w)
2: $B \leftarrow$ Number of iterations
3: **for** $i \leftarrow 1, N$ **do**
4: $\quad p \leftarrow i$
5: $\quad$ **for** $j \leftarrow 1, B$ **do**
6: $\quad\quad u \sim U[0, 1]$
7: $\quad\quad q \sim U\{1, \ldots, N\}$
8: $\quad\quad$ **if** $u \leq w(q)/w(p)$ **then**
9: $\quad\quad\quad p \leftarrow q$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: $\quad idx(i) \leftarrow p$
13: **end for**

2.4.6 Rejection

In the same paper, Murray [28] states that if an upper bound of the particle weights is known, rejection sampling is possible. Similar to Metropolis, rejection sampling does not require collective operation, such as *prefix-sum*, and is not affected by particle sets larger than 2^{20}. It also offers the following advantages:

1. it is unbiased

2. it permits a first deterministic proposal that $a^i = i$

A major difference between Metropolis and rejection resampling methods is thread execution. Thread execution in Metropolis is deterministic because the number of inner for-loop iterations is set to B. On the other hand, the inner loop of rejection is a while-loop, at line (6) of Algorithm 9. This causes the runtime of independent threads to vary, which is an example of a *variable task-length problem* [28]. It is important to ensure that the upper bound, $w_{\max}$, is tight. Otherwise the method may perform poorly. While empirical calculations of $w_{\max}$ can be performed, $w_{\max} = max\{w^1, \ldots, w^N\}$, it would defeat the purpose of the approach by introducing a collective operation. Due to the variable task-length, Metropolis may be the preferred choice if its bias is acceptable and an appropriate B is selected.

Algorithm 9 Rejection Resampling

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count (w)
2: $B \leftarrow$ Number of iterations
3: **for** $i \leftarrow 1, N$ **do**
4: $\quad p \leftarrow i$
5: $\quad u \sim U[0, 1]$
6: $\quad$ **while** $u \leq w(p)/w_{max}$ **do**
7: $\quad\quad q \sim U\{1, \ldots, N\}$
8: $\quad\quad u \sim U[0, 1)$
9: $\quad$ **end while**
10: $\quad idx(i) \leftarrow p$
11: **end for**

2.4.7 Coalesced Metropolis

Dülger [30] recently improved performance of Metropolis by implementing it on a GPU with coalesced memory accesses to global memory. While Metropolis

resampling is well suited to utilize the multi-core architecture of a GPU, it does not perform efficient global memory accesses. This is because of the way it randomly generates an index in line (7) of Algorithm 8, and reads that element from the particle weight array. This has a negative impact on performance because the GPU tries to coalesce global memory loads and stores issued by threads of a *warp* into as few transactions as possible to minimize DRAM bandwidth. A warp is a maximal subset of threads from a single cooperative thread array (CTA), such that the threads execute the same instructions at the same time. This will be explained in detail in Chapter 3. When concurrent threads simultaneously access memory addresses that are very far apart in physical memory, there is no chance for the hardware to combine the accesses. Uncoalesced accesses can cause up to a 57.0% performance loss [31].

Dülger provides two methods, both of which are faster than Metropolis but at the expense of quality. The two techniques are designated as Metropolis-C1 (abbreviated C1) and Metropolis-C2 (abbreviated C2). From this point forward, the original, or uncoalesced, Metropolis will be referred to simply as Metropolis. Because read-/write operations to global memory are performed in segments, these modifications constrain threads within a warp to only read and write within these segments. A segment is defined as a fixed number of contiguous elements. All the threads in the warp select random weights within a segment.

C1 is presented in Algorithm 10, SC is the number of segments and DC is the number of elements in each segment. All threads in a warp will select the same s, where s is the index of the selected segment drawn from a uniform distribution. This can be ensured by passing the warp index to the random number generator. Next, p

Algorithm 10 Metropolis-C1 Resampling

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count (w)
2: $B \leftarrow$ Number of iterations
3: **for** $i \leftarrow 1, N$ **do**
4: $\quad p \leftarrow i$
5: $\quad s \sim U\{0, \ldots, SC\}$
6: $\quad$ **for** $j \leftarrow 1, B$ **do**
7: $\quad\quad u \sim U[0, 1]$
8: $\quad\quad q \sim U\{(s-1) * DC + 1, \ldots, s * DC\}$
9: $\quad\quad$ **if** $u \leq w(q)/w(p)$ **then**
10: $\quad\quad\quad p \leftarrow q$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad idx(i) \leftarrow p$
14: **end for**

is a random index between the first and last elements of the segment. Although not explicitly stated in the paper, if DC is larger than the size of a warp, currently 32, then it is possible that global memory accesses for that warp will not be coalesced. By definition, a coalesced read occurs when threads in a warp access all required memory locations using a single cache. This is explained in detail in the CUDA Programming Guide [32] under the Best Practices section.

C2 in Algorithm 11 has the same parameters as Algorithm 10. The only difference is when the selection of s is performed. For C2, it is performed in the inner-loop during each iteration of B. C2 is slower than C1 because of the additional random numbers generated in the inner loop. However, C2 provides higher quality results because it encounters more variety in the selection of weights. To reiterate, while C1 and C2 are faster than Metropolis, they produce worse quality because they select weights from a limited portion of the particle weight array. Therefore, C1 and

Algorithm 11 Metropolis-C2 Resampling

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $N \leftarrow$ count (w)
2: $B \leftarrow$ Number of iterations
3: **for** $i \leftarrow 1, N$ **do**
4: $\quad p \leftarrow i$
5: $\quad$ **for** $j \leftarrow 1, B$ **do**
6: $\quad\quad u \sim U[0, 1)$
7: $\quad\quad s \sim U\{0, \ldots, SC\}$
8: $\quad\quad q \sim U\{(s-1) * DC + 1, \ldots, s * DC\}$
9: $\quad\quad$ **if** $u \leq w(q)/w(p)$ **then**
10: $\quad\quad\quad p \leftarrow q$
11: $\quad\quad$ **end if**
12: $\quad$ **end for**
13: $\quad idx(i) \leftarrow p$
14: **end for**

C2 variations of Metropolis provide a spectrum of speed versus quality trade-off for users.

CHAPTER 3

GRAPHICS PROCESSING UNIT (GPU)

GPUs were not always the parallel processing powerhouses they are today. At their conception, GPUs were to provide current hardware with a more efficient work flow for graphics processing. The original GPUs were modeled after the concept of a graphics pipeline and used fixed purpose hardware. The graphics pipeline refers to the process and steps of rendering images to a computer display. Even though transferring more of the graphics pipeline to the GPU progressed significantly through the 1980s and 1990s, they still required much help from the CPU. It was not until 1999, when NVIDIA implemented the last step of the pipeline in hardware, that reliance on the CPU was eliminated. Thus, the first consumer GPU was created. NVIDIA was the first to coin the term *GPU*.

One pitfall of the graphics pipeline is that it only allowed one pixel output per clock cycle, meaning CPUs could still send more triangles, the basic texture facet used in graphics processing, to the GPU than it could handle. This leads the way to introducing multiple pipelines in parallel on a GPU. Another downside to early GPUs is that the pipeline was a fixed-function pipeline, meaning once graphics data entered the pipeline it could not be modified. This was fixed with the introduction of

a programmable pipeline. A programmable pipeline allows a programmer access to manipulate and operate on data while it is in the pipeline. Application programming interfaces (APIs), such as Brook and Sh, removed the need for programmers to reformulate computational problems into terms of graphics primitives. Newer languages, such as NVIDIA's CUDA, allows the user to focus on high-performance computing concepts and less on earlier basic graphical concepts.

3.1 General Purpose Graphics Processing Unit (GPGPU)

With the introduction of programmable pipelines and streaming multiprocessors (SMs), GPUs began to be utilized as GPGPUs, performing calculations in applications conventionally performed by the CPU. An SM has multiple cores that can access and execute multiple threads or operations simultaneously. Although GPUs can only process independent fragments, it can do them in parallel. Where a CPU might have 4, 8, 16, or 32 cores, a GPU can have up to several thousand. These cores are accessed through *kernels*, which are functions that work on each element. GPUs are extremely efficient at the single instruction, multiple data (SIMD) paradigm, or data parallelism. GPU processing can perform mathematically intensive computations on very large data sets, while a CPU can run the operating system and perform traditional serial tasks. This is an example of heterogeneous processing, which refers to systems that use more than one kind of processor.

3.2 Programming Model

With the increasing popularity of GPUs outside of the graphics domain, the CUDA API was introduced by NVIDIA in 2006. CUDA offers a mature development environment via an extension to the C/C++ programming language. An alternative to CUDA is OpenCL, provided by the Khronos Group in 2008. It is an open and royalty-free standard that can be utilized on a wide selection of hardware, including multi-core CPUs, GPUs (AMD and NVIDIA), field-programmable gate arrays (FPGAs), and digital signal processors (DSPs). While the programming scheme is similar, it does not provide the same performance as CUDA on NVIDIA GPUs since CUDA is tied closer to the hardware. This can become important when trying to achieve optimal performance.

CUDA provides a heterogeneous environment where programs are divided between the *host* and the *device*. CUDA allows programmers to define special functions, called kernels, that are called by the host code running on the CPU. Kernels can then be executed in parallel on the GPU by a collection of threads. Kernels are launched in a *grid* made up of a group of *blocks* that contain numerous threads. Once blocks are distributed to SMs they are divided into warps. All warps contain 32 threads and are executed concurrently through a series of warp schedulers. While the number of threads per warp has stayed constant through the evolution of GPGPU development, the number of warp schedulers vary between architecture. Also, entire blocks are required to reside on a SM. Therefore, the number of threads in a block are limited by resources. Threads are organized in a block, and blocks are organized in a grid in a

one-, two- or three-dimensional fashion. Each thread and block is designated a unique ID that can be accessed in the kernel by a built-in variable `threadIdx.x (,y,z)` and `blockIdx.x (,y,z)`, respectively. With the release of CUDA 9.X, threads within a block are synchronized with Cooperative Groups. The compute capability of a device is represented by a version number. The version number identifies the features supported by the GPU hardware and is used by applications at runtime to determine which hardware features and/or instructions are available on the target GPU. Table 3.1 provides a sublist of available resources per thread block by compute capability.

Table 3.1: GPU resources available (per thread block), adapted from [32].

	Compute Capability											
Technical Specifications	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.5
Threads per block	1024											
Max dim size of block	1024 x 1024 x 64											
Max dim size of grid	2147483647 x 65535 x 65535											
Warp size	32											
Registers (K)	64	32	64				32	64		32	64	
Texture Memory (1D)	2^{27}											
Shared Memory (KB)	48										96	64
Constant Memory (KB)	64											

3.2.1 Memory Model

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 3.1. Pros and cons per memory space may vary and great care should be taken when choosing the appropriate memory space for a given

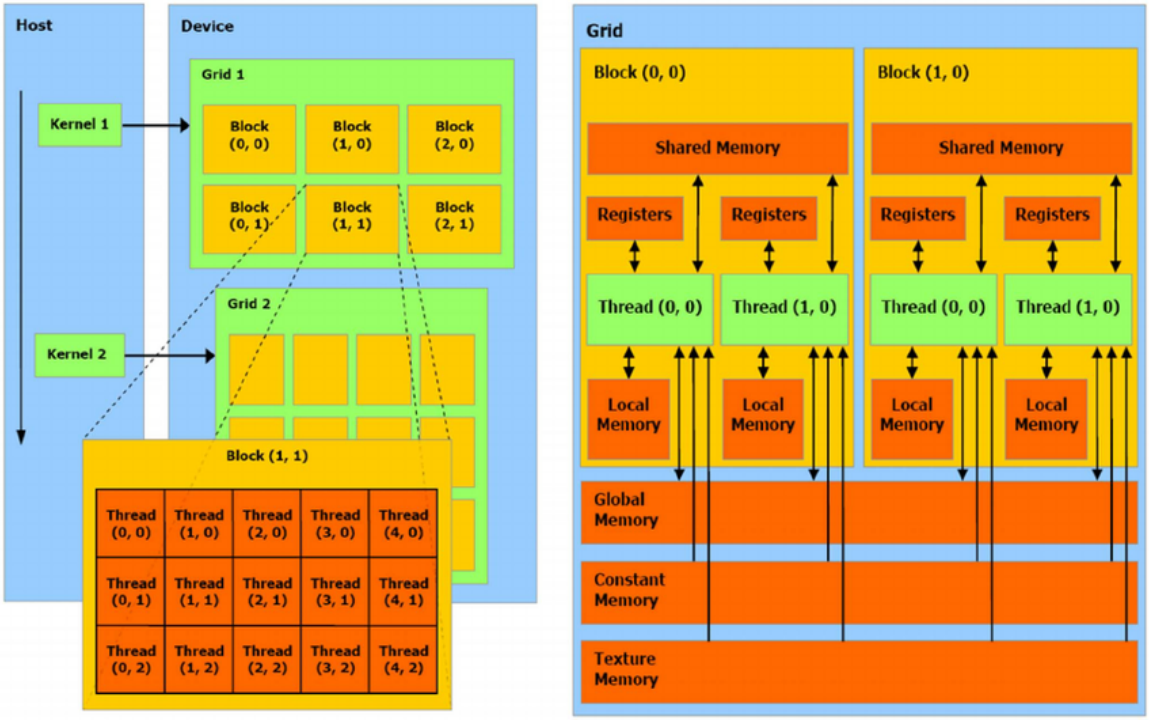


Figure 3.1: CUDA memory model showing the hierarchy of thread layout and memory spaces [32].

task. Some memory is available during the life of a thread block, such as registers and shared memory. Other spaces are available during the life of the program, such as global and constant memory. Specific details for each memory space will be provided in the following subsections.

3.2.1.1 Global Memory

Global memory is stored off-chip and is the slowest and largest available memory on the GPU. It is accessed through 32-, 64- or 128-byte memory transactions. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory

transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, higher throughput can be achieved with coalesced memory accesses.

3.2.1.2 Local Memory

The local memory space resides in device memory, so local memory accesses have the same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing as global memory. Structures and arrays that consume too many registers are stored in local memory. If local variables in a kernel exceed available resources, the excess will be stored in local memory. Local memory usage can have a negative impact on performance through increased memory traffic and instruction count.

3.2.1.3 Shared Memory

Shared memory is a fast on-chip memory available to all threads within a thread-block. It is much faster than the off-chip global memory but is generally much smaller, depending on the hardware generations, and does not persist through kernel calls. To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called *banks*, which can be accessed simultaneously. However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict, and the access has to be serialized. If too many bank conflicts occur the advantages of shared memory can be negated.

3.2.1.4 Constant Memory

The constant memory space resides in device memory and is cached in the constant cache. Maximum throughput is achieved when all threads within a warp access the same constant memory location during an instruction execution. Thread calls that access different memory locations will be serialized. Constant memory space is a limited resource with historically 64 KB per SM. Since constant memory is static, it can be accessed by any kernel in the context space without being passed as a parameter. Data can be stored using `cudaMemcpyToSymbol`. It is important to note that constant variables have a file scope linkage, meaning `cudaMemcpyToSymbol` can only be called within the same *.cu* file. The *.cu* file extension is given to developer files written for CUDA and are compiled with **nvcc**, a compiler program included in the NVIDIA CUDA toolkit.

3.2.1.5 Texture Memory

Texture memory space is read-only and cached. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. Textures can be initialized as a texture object or texture reference and can be an advantageous alternative to reading device memory from global or constant memory. The space provides features such as linear filtering, interpolation between texels, normalized texture coordinates, and addressing modes that are not provided by any other memory space.

3.2.1.6 Registers

Registers are the fastest and most limited memory space on a GPU. Generally, accessing a register consumes zero extra clock cycles per instruction. As shown in Table 3.1, there are 32K or 64K registers per block and 255 registers limited per thread in a block. Registers can suffer from bank conflicts. Therefore, the numbers of threads per block should be a multiple of 64 for best results. Programmers should be aware of *register pressure* and *spillage*, which occurs when there are not enough registers available for a given task, and data must be stored in local memory.

3.3 Libraries and Tools

Writing, tuning, and maintaining kernel code is perhaps the most challenging and time-consuming aspect of CUDA programming. NVIDIA provides a plethora of GPU-accelerated libraries and tools to speed up application development and performance. GPU-accelerated libraries for linear algebra, signal processing, image and video processing lay the foundation for compute-intensive applications in areas such as molecular dynamics, computational chemistry, medical imaging, and seismic exploration. NVIDIA libraries are optimized for maximum performance and are tuned to take advantage of the latest hardware features. Programmers are able to get performance improvements by compiling their program with a newer CUDA toolkit, which in turn reduces on-going development cost.

3.3.1 CUDA UnBound (CUB)

CUDA UnBound (CUB) is an NVIDIA library containing collective kernel-level primitives designed around reusable software components, such as warps and blocks for the single instruction, multiple threads (SIMT) paradigm. Collective software primitives are essential for constructing high-performance, maintainable CUDA kernel code. The library also provides global primitives, built from collectives, that provide performance portability. Historically, programmers have had to rewrite kernels to take advantage of new features that became available with the latest hardware. CUB reduces development cost and encapsulates complexity by insulating changing capabilities of underlying hardware from the programmer. It is continuously evolving to accommodate new architecture-specific features and instructions.

3.3.2 cuRAND

The cuRAND library provides facilities that focus on the simple and efficient generation of high-quality pseudorandom and quasirandom numbers. Quasirandom numbers generate successive numbers as far away as possible from existing numbers in the set. While they are too uniform to pass randomness tests, they avoid clustering and speed up convergence. Pseudorandom numbers are more appropriate for applications that require more randomness. cuRAND consist of host and device API calls to access device memory. The possible random number generators are listed in Table 3.2.

Table 3.2: Random number generators available in cuRAND.

Generator Types	
Pseudorandom	Quasirandom
MRG32k3a	Scrambled Sobol (32-bit)
MT19937	Scrambled Sobol (64-bit)
MTGP32	Sobol (32-bit)
Philox_4x32_10	Sobol (64-bit)
XORWOW	

3.3.3 cuBLAS

The cuBLAS library is an implementation of basic linear algebra subprograms (BLAS) in CUDA. BLAS was originally written in Fortran and has been highly optimized for multiple architectures over its history. cuBLAS consists of two sets of API, the regular cuBLAS API and cuBLASXT API which utilizes multi-GPU capabilities. BLAS is structured in three levels; level 1) scalar and vector based operations, level 2) matrix-vector operations, and level 3) matrix-matrix operations. Levels 1 and 2 are memory limited, while level 3 is compute limited. It can handle multiple data types such as half-, single-, and double precision, complex, and double-complex. CUDA 9.X provides access to Tensor Cores that were introduced with the Volta architecture for matrix-matrix multiplication.

3.3.4 CUDA-MEMCHECK

An important consideration of any program is error-checking and CUDA-MEMCHECK is a functional correctness checking suite included in the CUDA toolkit.

It provides a series of tools that consists of different types of checks. The memcheck tool is a runtime error detection tool, capable of precisely detecting and attributing out of bounds and misaligned memory access errors in multiple memory spaces. To accurately detect memory leaks, a CUDA context must be destroyed on program termination. This can easily be done with `cudaDeviceReset()`. The racecheck tool can report shared memory data access hazards that can cause data races, also called a race condition. Race conditions are computational hazards that arises when the results of the program depend on the timing of uncontrollable events, such as the execution order of threads. It checks for three hazards: write-after-write (WAW), write-after-read (WAR) and read-after-write (RAW). The initcheck tool can report cases where the GPU performs uninitialized accesses to global memory. The synccheck tool can report cases where the application is attempting invalid usages of synchronization primitives. When ran in combination with the debug compile file, -G, the line number at which an error occurs will be reported along with call hierarchy. More details about this error-checking suite are provided in the CUDA Toolkit Programming Guide [32].

3.3.5 Profilers

NVIDIA provides profiling tools in its CUDA toolkit to help understand and optimize CUDA applications. Nsight Systems is a graphical profiling tool that displays a timeline of an application's CPU and GPU activity at the system level and provides low overhead performance analysis with insights developers need to optimize their software. It provides information on concurrent kernels, unified memory, and power, clock, and thermal profiling. It has the ability to provide further insight to

Figure 3.2: NVIDIA Nsight Systems.

kernel bottlenecks and makes suggestions for further optimizations. A screenshot of the visual profiler is provided in Figure 3.2. If programmers need the detailed performance metrics of an individual kernel and API debugging via a user interface and command line tool, they can use Nsight Compute.

3.4 Architectures

The NVIDIA GPU architecture is built around a scalable array of multithreaded SMs and employs a unique architecture called SIMT. A GPU provides hundreds, and sometimes thousands, of cores per device. A GPU core provides a much simpler control block than a CPU core and does not include advanced branch prediction or speculative execution. This trade-off allows the GPU to provide maximum throughput. A multiprocessor creates, manages, schedules, and executes threads in

groups of 32 parallel threads, called warps. Warps are handled independently by a scheduler. Warps within a block can be launched randomly and threads with a warp can execute out of lockstep. This is why block synchronization is important when dealing with shared memory. Execution context (program counters, registers, etc.) for each warp is processed by a multiprocessor and maintained on chip during the entire lifetime of the warp. A warp scheduler selects a warp that has threads ready to execute its next instruction. Therefore, to achieve maximum throughput, launching blocks with numerous threads can provide enough work to hide long latency memory reads. While the overview presented above is common across all architectures, low-level details vary. Those variations, that affect coding performed in this research, will be described in the following sections.

3.4.1 Kepler

Kepler architecture was designed using 28 nm technology with a primary focus on reducing energy consumption. This was achieved by using the primary GPU clock rather than the 2x shader clock. Running a larger number of cores at a lower clock rate allowed the removal of power-hungry clocking logic at the expense of area optimizations. Two Kepler cores used 90% of the energy required by the previous generation. NVIDIA realized that the compiler could be used to determine up front when instructions would be ready to issue. As a result, NVIDIA replaced several complex and power-expensive blocks with a simple hardware block that retrieves predetermined latency information, and uses it to mask out warps from eligibility at the inter-warp scheduler stage. This scheduling scheme has been carried forward on all

successive architectures. Kepler SMs schedule threads in groups of 32 parallel threads called warps. Each SM features four warp schedulers and eight instruction dispatch units, allowing four warps to be issued and executed concurrently. Two independent instructions per warp can be dispatched in each cycle and are very useful when improving instruction level parallelism (ILP). An example is shown in Figure 3.3.

A single Kepler SM contains 192 FP32 CUDA cores, 64 FP64 units, 32 special function units (SFUs), and 32 load/store (LD/ST) units, as shown in Figure 3.4. Each of the CUDA cores comes with fully pipelined floating-point and integer arithmetic logic units. The cores are also compliant with IEEE 754-2008, providing fuse-multiply-add (FMA) functionality. An FMA operation is performed in one step, with a single rounding, yielding a more accurate result.

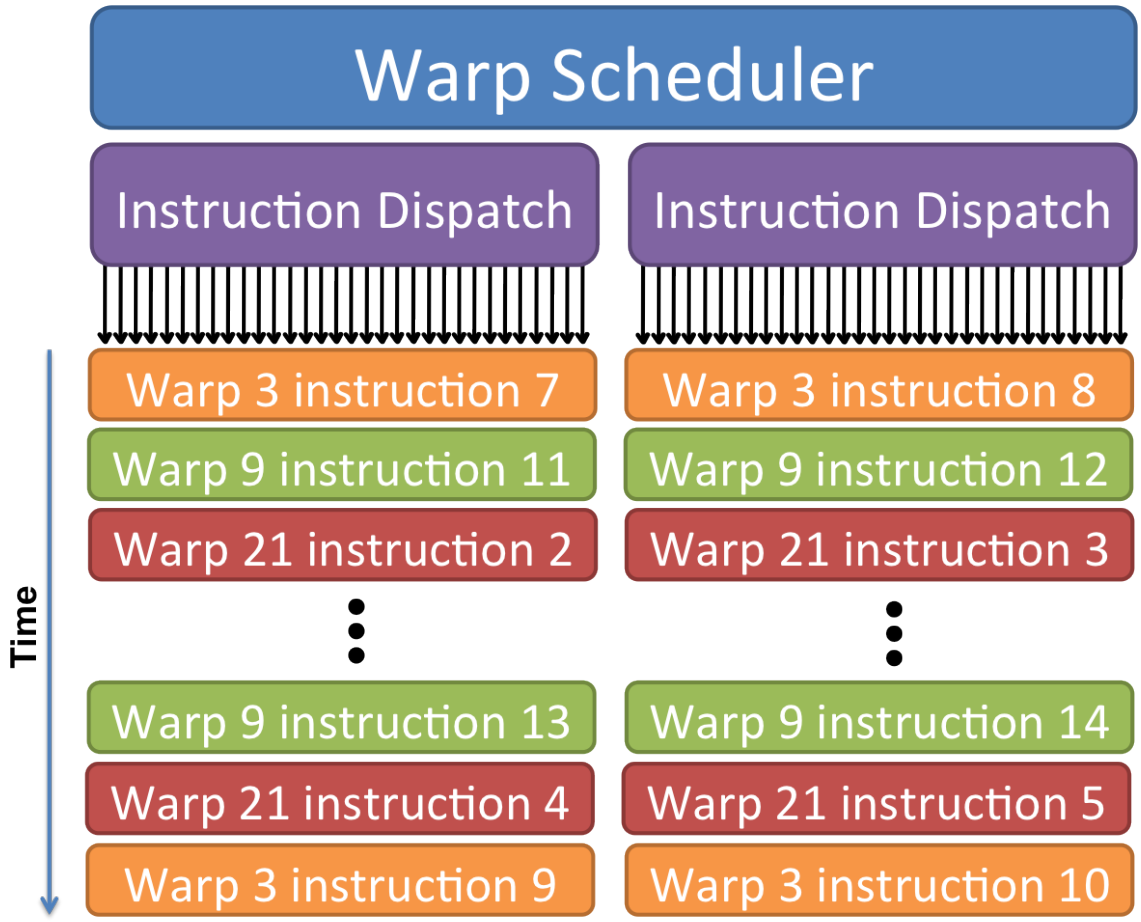


Figure 3.3: A single warp scheduler unit containing dual dispatch units.

Kepler compute capability 3.5 introduced 255 registers per threads, an increase from 63 in previous generations. This provided substantial performance improvements by removing register pressure and reducing spillage into local memory, which is located in global or device memory. The shuffle instruction was another important feature added to this series. This feature provided a way for programmers to share information between threads without using shared memory resources. It also offers a performance increase since load-and-store operations are executed in a single cycle.

Figure 3.4: A single Kepler SM: 192 FP32 CUDA cores, 64 FP64 units, 32 Special function units (SFU), and 32 load/store (LD/ST) units.

Kepler introduced configurable 64 KB shared memory. Since shared memory and L1 cache were packaged together, programmers could configure 16/48, 32/32, and 48/16 splits on a per kernel basis. While this provided much more flexibility,

it implicitly synchronized kernels when a new configuration was required. Two improvements to texture memory were also added. First, bindless textures eliminated previous limits to the number of texture objects that existed on previous architectures. Second, an additional 48 KB read-only data cache was provided that allow programmers to load data directly at the launch of a kernel and lasting the duration. This is beneficial because it takes both the load and working set footprint off the shared/L1 cache path. It can be accessed by providing hints to the compiler with tags `const __restrict`, meaning the memory is not aliased, or explicitly with the `ldg()` intrinsic [33]. This additional cache can be seen in Figure 3.5.

Lastly, dynamic parallelism and Hyper-Q added levels of parallelism that had not previously existed. With dynamic parallelism, a GPU was able to generate, synchronize, and control new work for itself, a role traditionally performed by the CPU. Hyper-Q increased the number of hardware managed work queues from 1 to 32. This provided more efficient concurrency when a GPU needed to handle work from multiple CPU cores simultaneously or from MPI processes. Kepler introduced many architectural changes from its predecessors that are still preserved today.

3.4.2 Maxwell

With the introduction of the Maxwell architecture, NVIDIA reduced the CUDA core count from 192 in the Kepler to 128 per SM. While still using 28 nm technology, NVIDIA was able to double the number of SMs used on some chips without doubling the size of the die. This new core configuration allows for better resource utilization on data transfers. The Maxwell SM retains the same number of instruction issue slots

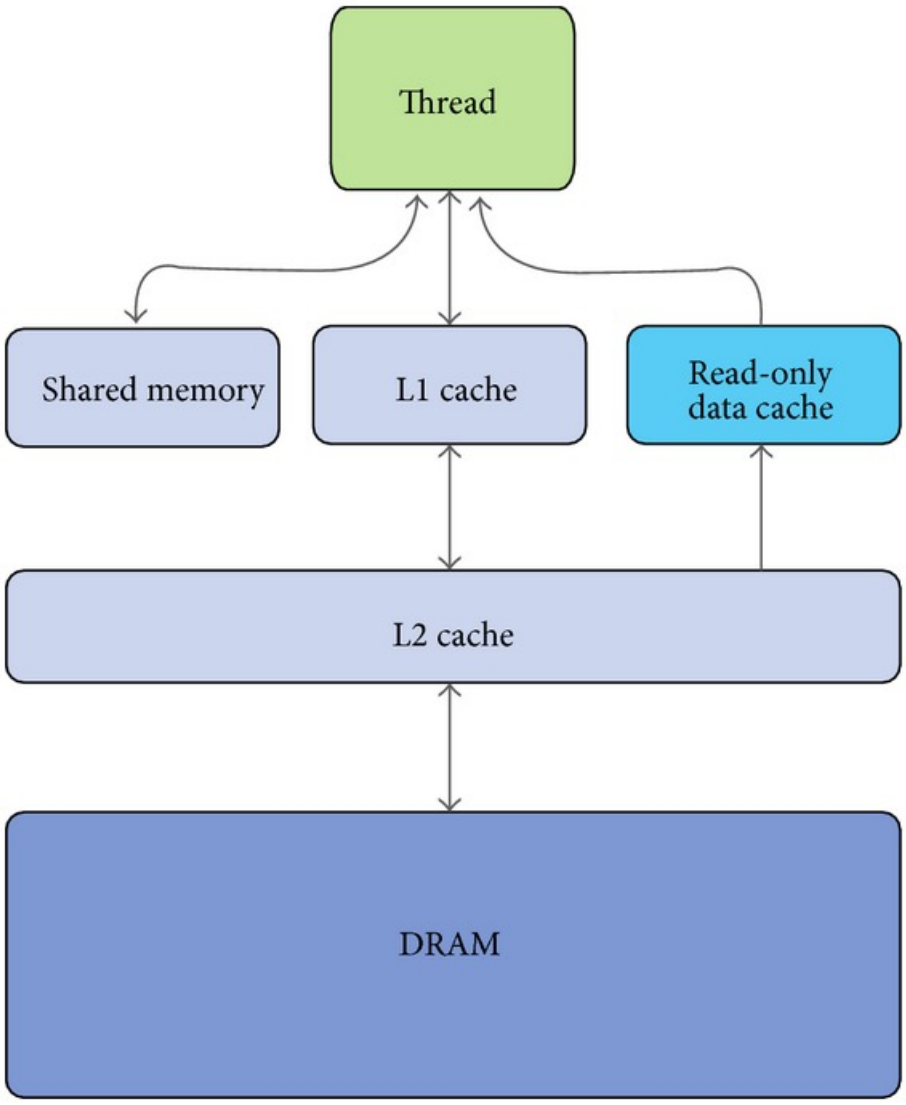


Figure 3.5: Kepler Memory Hierarchy.

per clock and reduces arithmetic latencies compared to the Kepler design. To balance efficiency and workload requirements of modern games, texture units were also reduced from sixteen to eight per SM. Higher clock rates enable the reduced number of texture units to maintain and/or exceed performance of the previous series. While the Maxwell SM contains four warp schedulers and eight instruction dispatch units

like its predecessor, it is evenly partitioned into four, 32-core processing blocks. Each processing block has dedicated scheduling and instruction resources. This is one of the greatest changes NVIDIA has made to reduce power consumption. Shared resources, though extremely useful when you have the workloads to fill them, do have drawbacks. They waste space and power if not fed, and there is additional scheduling overhead from having to coordinate the actions of the warp schedulers [34]. These changes can be seen in Figure 3.6.

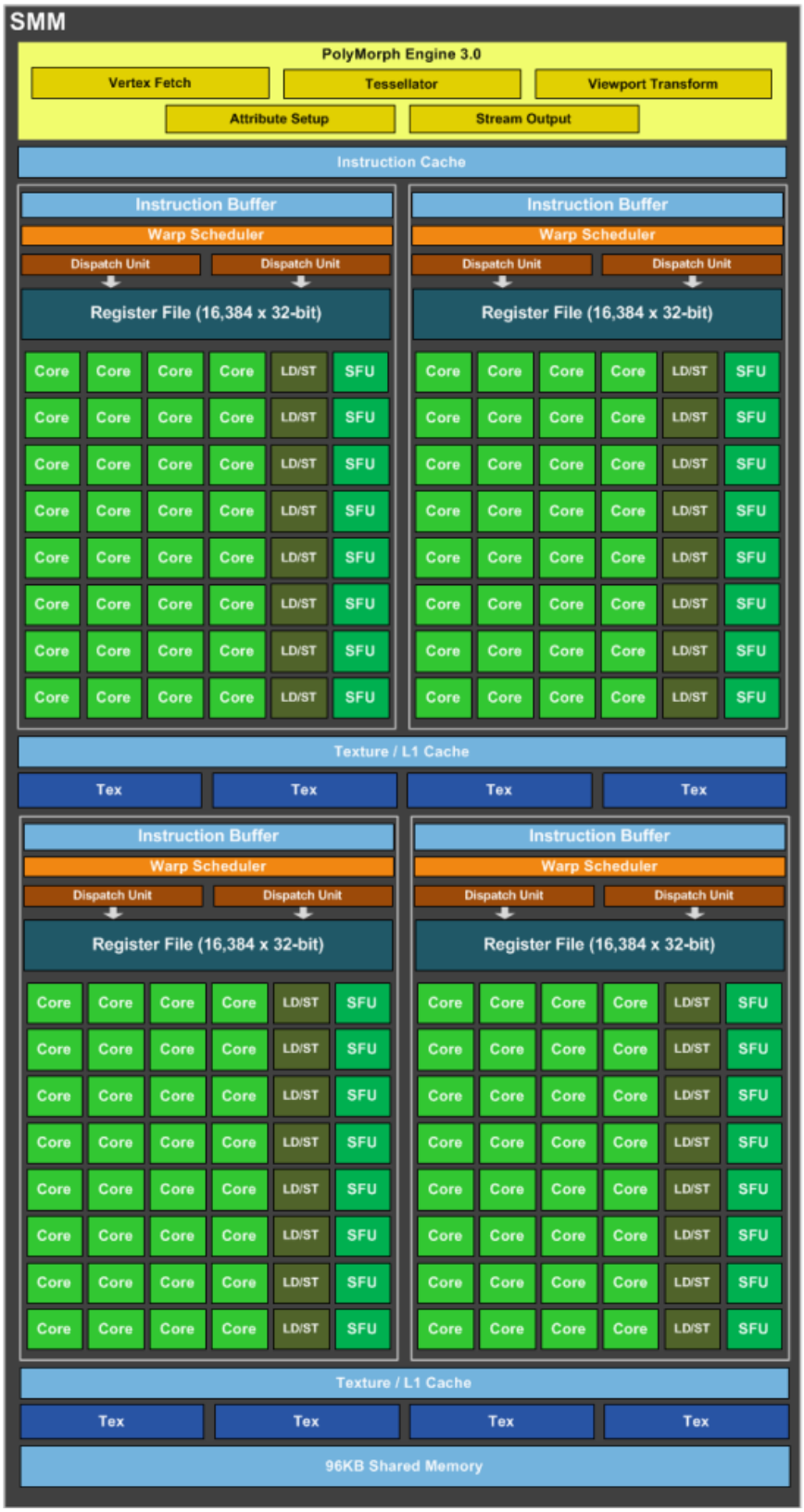


Figure 3.6: Single Maxwell SM.

This series was the first to provide dedicated shared memory on an SM and combines L1 and texture caches into a single unit. Shared memory per SM is either 64 KB or 96 KB depending on the chip. The increase in maximum number of active thread blocks per multiprocessor, from 16 to 32, was a significant improvement. This results in automatic occupancy improvements without any code modifications by the programmer. In addition, the throughput of many integer operations including multiply, logical operations, and shift were improved on Maxwell. Specialized integer instructions that can accelerate pointer arithmetic were added. Lastly, L2 cache was significantly increased to 2 MB, reducing the amount of traffic that needs to cross the memory bus. This reduced both the power spent on the memory bus and improved overall performance.

3.4.3 Pascal

The Pascal architecture was the first to use 16 nm FinFET technology. Even though smaller transistors were used, Pascal incorporated 64 (GP100) CUDA cores per SM, while Maxwell and Kepler used 128 and 192, respectively. The SM, shown in Figure 3.7, is partitioned into two processing blocks. It maintains the 32 single-precision CUDA cores, an instruction buffer, a warp scheduler, and two dispatch units per block used by Maxwell. With the reduction in the number of processing blocks, texture units were also reduced from eight to two per SM. The Pascal architecture is the worlds first GPU architecture to support high bandwidth memory 2 (HBM2), which offers three times the memory bandwidth of the Maxwell. Because HBM2 is stacked memory, it allowed NVIDIA to design a denser GPU compared to

previous versions. It provides a much simpler data path organization than the Kepler and improves scheduling and floating-point utilization over the Maxwell. The HBM2 memory requires less die area and reduced power consumption over previous architectures. A higher ratio of shared memory, registers, and warps per SM allows for more efficiently executed code [35].

Figure 3.7: Single Pascal SM.

This architecture also includes 16-bit precision operations with twice the throughput as 32-bit operations. FP16 plays an important role in deep learning (DL) applications by reducing memory usage and increasing arithmetic performance and data transfers. An important thing to note is that as SMs became smaller and more were

added to a chip, NVIDIA was able to increase the amount of L2 cache to 4 MB. This allowed for fewer requests to device memory, reducing overall board power and improving performance.

3.4.4 Volta

One of newest architectures is the Volta series. It is fabricated with new 12 nm FinFET NVIDIA (FFN) technology. Volta returns to four processing blocks per SM like the Maxwell, but the processing block has some significant differences. First, instead of having 32 FP32 cores, it has 16 FP32 cores and 16 INT32 cores, as shown in Figure 3.8. The addition of INT32 cores allows for simultaneous execution of FP32 and INT32 operations at full throughput, while also increasing instruction issue throughput. FMA math operations are reduced from six cycles on Pascal to four. The instruction buffer in each block is replaced with a new L0 instruction cache to provide higher efficiency and is private to every scheduler. It also mitigates any penalty associated with large instruction size [36]. In addition, the number of dispatch units is reduced to one, therefore it no longer has superscalar execution. This means that Volta is a pure thread level parallelism (TLP) design; max utilization comes from maximizing the number of threads active at any given time. The Volta is said to have a streamlined instruction set for simpler decoding and reduced instruction latencies [37].

The Volta architecture combined the L1 cache and shared memory, similar to Kepler. The combined capacity is 128 KB per SM and allows for a configurable, shared memory capacity of 96 KB. All 128 KB can be used as cache by kernels that

Figure 3.8: Single Volta SM.

do not use shared memory. A key reason to merge the L1 data cache with shared memory in Volta is to allow L1 cache operations to attain the benefits of shared memory performance. Shared memory provides high bandwidth and low latency, but the programmer needs to manage this memory explicitly. Volta narrows the gap

between applications that explicitly manage shared memory and those that access data in device memory directly. Prior NVIDIA GPUs performed load caching, while the Volta performs write caching. This further improves performance. Volta is the first architecture to implement independent thread scheduling. The individual CUDA cores within a 32-thread warp now have a limited degree of autonomy. Threads can be synchronized at a fine-grain level, which means that while the SIMT paradigm is still alive and well, it has greater overall efficiency. A visual aid is provided in Figure 3.9. Independent thread scheduling allows threads to diverge and reconverge at sub-warp granularity. In order to ensure proper warp synchronization, Volta should be paired with CUDA 9.X or greater and use Cooperative Groups.

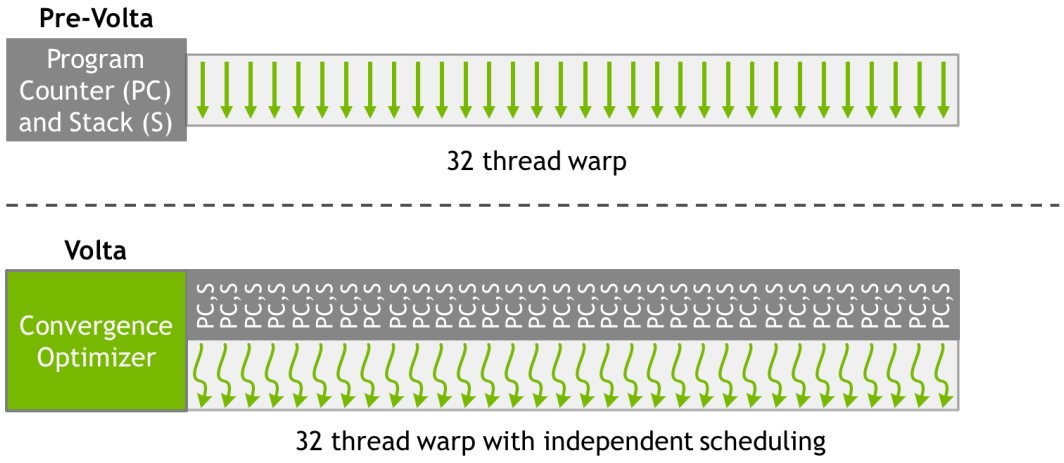


Figure 3.9: Independent thread scheduling comparison between Pre-Volta and Volta architectures.

One of the most significant additions to the Volta SM is the two mixed-precision Tensor Cores. They are highly utilized by the DL matrix arithmetic. Tensor

Cores are a new type of core geared specifically for tensor math operations. The significance of these cores is that by performing a massive matrix multiplication operation in one unit, NVIDIA can achieve a much higher number of FLOPS for this one operation. Unfortunately, Tensor Cores are relatively rigid and are not ideal for anything except tensor operations. They are only applicable to certain classes of compute tasks.

3.4.5 Turing

Turing, the latest architecture, is constructed using 12 nm FFN technology and consists of an SM with 16 FP32 cores, 16 INT32 cores, two Tensor Cores, one warp scheduler, and one dispatch unit, shown in Figure 3.10. Among significant similarities with the Volta, the Turing differs in that NVIDIA has added an additional datapath to further increase parallelism between the FP32 cores and the INT32 cores. With a second datapath, instructions like integer adds for addressing and fetching data can coincide with floating point math. Early studies have shown an additional 36% throughput on modern applications [38]. L1 cache on the Turing is also very similar to the Volta. For Turing, there is a combined 96 KB of L1 and shared memory in an unified memory block. Compute workloads can partition the L1/shared memory space with up to 64 KB as L1 and the remaining 32 KB as shared memory, or vice versa. For Volta, the shared memory can be configured up to 96 KB. Anything above 48 KB must be done dynamically.

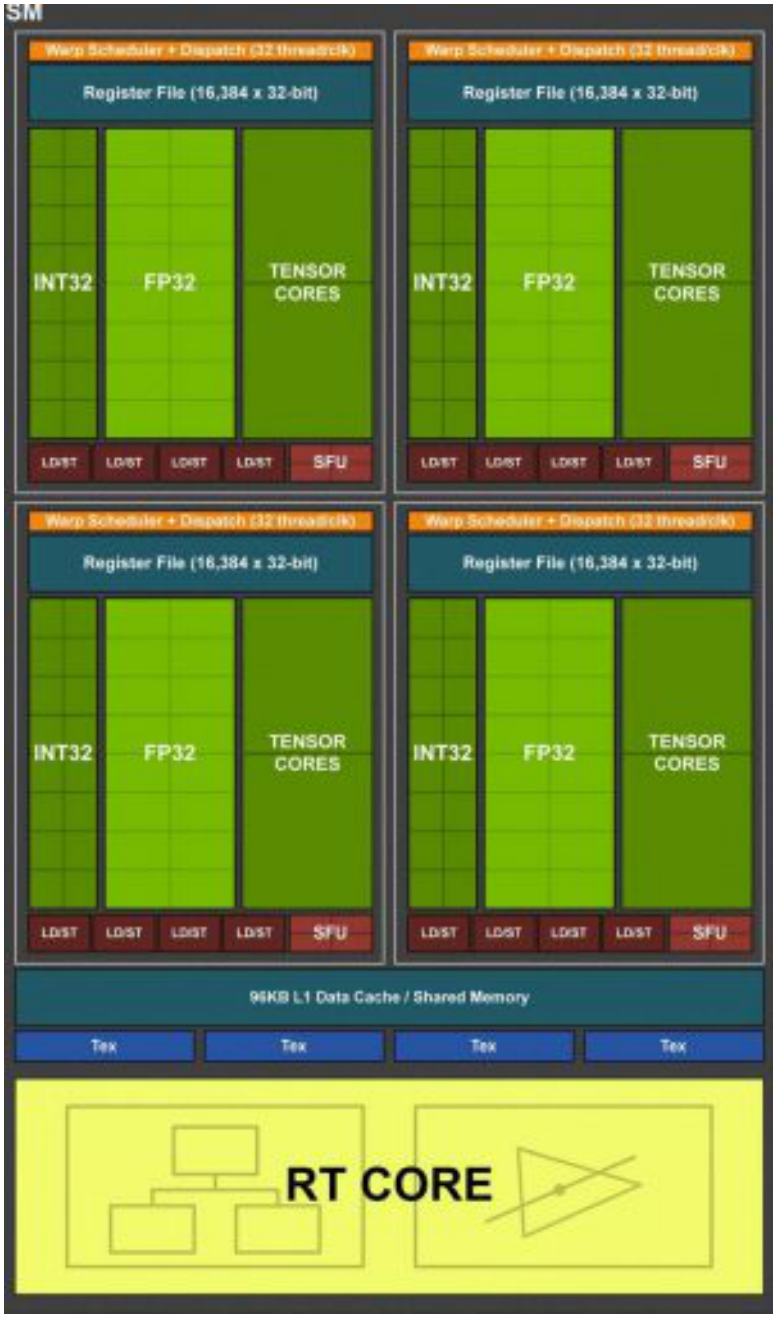


Figure 3.10: Single Turing SM.

Turing is the first GPU architecture to support GDDR6 memory, enabling as little as a 27% increase in memory bandwidth. Aside from these improvements and

some additional Tensor Core capabilities, there are not many noteworthy differences between the Volta and Turing compute units. Although it does not contribute to this research, the most significant advancement that Turing brings to the industry is the new ray-tracing cores (RT cores). Ray-tracing, in short, is a rendering process that emulates how light behaves in the real world and is primarily used in gaming graphics. These new cores enable the ability to make calculations in real-time.

3.4.6 Embedded Devices

While higher quality graphics for the gaming industry have driven advancements in GPU evolution, small fanless devices and autonomous cars have driven advancements in embedded systems, particularly the SOC. In 2013, NVIDIA introduced their first embedded GPU paired with an advanced RISC machines (ARM) CPU and memory. This has allowed programmers to provide high computing algorithms to devices previously using FPGAs. NVIDIA provides the Jetson embedded platforms, which include versions based off the Kepler (TK1), Maxwell (TX1), Pascal (TX2/TX2i), and Volta (Xavier) architectures. It packs this performance into a small, power-efficient form factor that is ideal for intelligent edge devices like robots, drones, smart cameras, and portable medical devices. Software can be installed using JetPack, which is a complete software development kit (SDK) that includes the board support package (BSP), libraries for deep learning, computer vision, GPU computing, and multimedia processing. Figure 3.11 shows a picture of a TX2 module (left) and a TX2 Developer Kit (right).

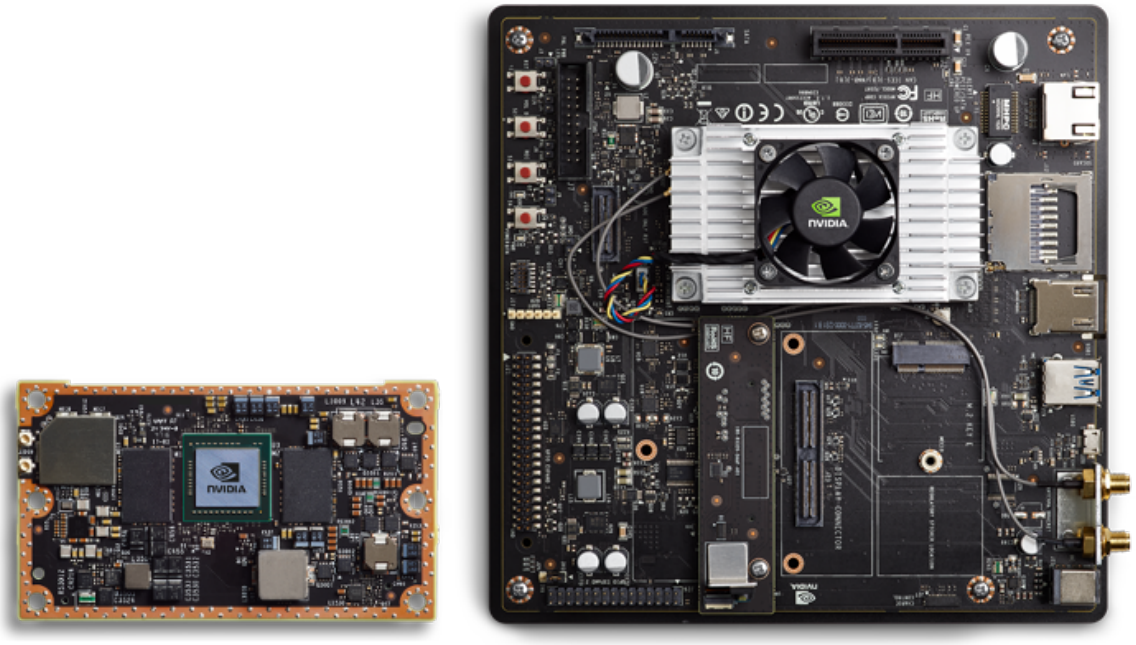


Figure 3.11: Side-by-side comparison of the Jetson TX2 module (left) and the Jetson TX2 Developer Kit (right).

CHAPTER 4

PARALLEL RESAMPLING TECHNIQUE

My hypothesis is that we can solve [the software crisis in parallel computing], but only if we work from the algorithm down to the hardware – not the traditional hardware first mentality.

—Tim Mattson, Intel

4.1 Algorithm Description

Systematic resampling is the algorithm most preferred due to its easy implementation and ability to minimize Monte Carlo variation [10, 18, 39]. Because of its similarities with stratified resampling the implementation presented below can easily be applied to both methods. Systematic and stratified resampling can be divided into three sections: 1) *prefix-sum* generation, 2) random number generation, 3) comparing prefix-sum and random numbers. Unfortunately, systematic and stratified, as well as some other traditional resampling algorithms, require collective operations across particles and weights due to data dependencies. This collective operation, also known as a prefix-sum, is the cumulative summation of particle weights. Major contributions have been made to parallelize prefix-sum algorithms on GPUs [40–43]. The parallel inclusive-prefix-sum has $\mathcal{O}(\log N)$ complexity. Therefore, it effectively

disappears as N continues to grow. Next, generating random numbers for these resampling algorithms can have a complexity of $\mathcal{O}(N)$ on a CPU. On GPUs random number generation can be distributed among threads bringing its complexity to $\mathcal{O}(1)$. This can provide a more pronounced improvement to stratified resampling which requires a different random number per particle. The third portion of systematic and stratified resampling, which requires iterating through the prefix-sum and comparing values to the uniform ordered random numbers, remains a serial process.

To efficiently utilize the GPU, work must be distributed to many threads. It is important to remember that data local to each thread is not visible by any other thread in a block. Taking a closer look, these resampling methods, they consist of a while-loop wrapped in a for-loop iterating over a *zero-based consecutive strictly monotonic* index through the prefix-sum, comparing elements to a random number [44]. This process can be split into two sub-processes; a *middle-out* approach where each process executes a while-loop on each element in the prefix-sum simultaneously. One sub-process increments through the prefix-sum until each and all threads have satisfied the comparator or until the end of the array is reached. The second sub-process then decrements through the prefix-sum until all threads satisfy the comparator or until the end of the array is reached. From henceforth, the kernel that increments will be referred to as *Up* and the the kernel that decrements will be referred to as *Down*. Intermediate comparator results are stored in a bit mask. This implementation produces a variable task-length problem for each thread similar to that of rejection resampling, as described in Section 2.4.6. Therefore, it has a maximum complexity of $\mathcal{O}(\ell)$, where ℓ is the number of strides from *c[thread]*, where c is the

prefix-sum array and *thread* is the thread index within a grid. A grid is a collection of all the threads executing the kernel. Algorithm 12 provides pseudocode of the parallel implementation. For brevity, stratified and systematic implementations have been combined, with the only difference being the random number generation. Notice the if-statement at line (8) during the Up kernel and line (20) in the Down kernel. It ensures that the while-loops do not access memory out of bounds.

The parallelized systematic and stratified are identical except for the creation of uniform random numbers. Both parallel methods should have similar timing because random number generation is performed by each thread.

Weights	0.06	0.01	0.05	0.09	0.08	0.05	0.09	0.06	0.09	0.08	0.04	0.01	0.02	0.09	0.09	0.09
Prefix-Sum	0.06	0.07	0.12	0.21	0.29	0.34	0.43	0.49	0.58	0.66	0.70	0.71	0.73	0.82	0.91	1.00
U[0,1)	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
u	0.01	0.08	0.14	0.20	0.26	0.33	0.39	0.45	0.51	0.58	0.64	0.70	0.76	0.83	0.89	0.95
Mask	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
							1st While Loop									
Mask	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0
Index	1	3	4	4	5	6	7	8	9	10	11	12	14	15	15	16
Mask	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Index	1	3	4	4	5	6	7	8	9	10	11	12	14	15	15	16
							2nd While Loop									
Mask	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	1
Index	1	3	4	4	5	6	7	8	9	9	10	11	14	15	15	16
Mask	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Final	1	3	4	4	5	6	7	8	9	9	10	11	14	15	15	16

Figure 4.1: A simplified example of the parallel resampling approach. The index for each particle (column) can be computed by individual threads in parallel.

Algorithm 12 Parallel Stratified (Systematic)

Require: $w \leftarrow$ Particle Weights
Ensure: $idx \leftarrow$ Resample Index
1: $c \leftarrow$ Inclusive-Prefix-Sum (w)
2: **for all** $t \leftarrow thread$ **do**
3: $N \leftarrow$ count (w)
4: $u_t \leftarrow (t + u_n(u_0))/N$ ▷ $u_n(u_0) \sim U[0,1)$; $n = 1, \ldots, N$
5: $m_t \leftarrow$ `true` ▷ Bit mask for thread t
6: $\ell_t \leftarrow 0$
7: **while** $m_t \neq$ `false` **do**
8: **if** $t > (N - \ell_t)$ **then**
9: $m_t \leftarrow$ `false`
10: **else**
11: $m_t \leftarrow c(t + \ell_t) < u_t$
12: **end if**
13: **if** $m_t =$ `true` **then**
14: $idx_t \leftarrow idx_t + 1$
15: **end if**
16: $\ell_t \leftarrow \ell_t + 1$
17: **end while** ▷ All mask threads must be FALSE to exit
18: $\ell_t \leftarrow 1$
19: **while** $m_t \neq$ `true` **do**
20: **if** $t < \ell_t$ **then**
21: $m_t \leftarrow$ `true`
22: **else**
23: $m_t \leftarrow c(t - \ell_t) < u_t$
24: **end if**
25: **if** $m_t =$ `false` **then**
26: $idx_t \leftarrow idx_t - 1$
27: **end if**
28: $\ell_t \leftarrow \ell_t + 1$
29: **end while** ▷ All mask threads must be TRUE to exit
30: **end for**

As an example, a simplified output of Algorithm 12 is provided in Figure 4.1. In the first while-loop, 4 out of 16 elements satisfy the comparator at line (11); therefore, only those elements are incremented. When all threads fail the while condition, the elements proceed to the next while-loop. In the second loop, only 3 elements fail the comparator at line (23) and must be decremented. Once all threads satisfy the

second while-loop, only the resampled index remains. Summing the total operations, what would have taken 23 operations using serial methods now can be completed in 4. Another advantage of this new method is that it allows the data to remain on the GPU without expensive copies to and from the CPU.

4.2 Optimizations To Parallel Implementation

While the naive parallel implementation presented in the previous section provides a speedup over the serial versions, both kernels can be modified in the following areas to better utilize GPU resources to achieve optimal performance:

1. Memory dependency stalls
2. Thread execution efficiency
3. Sub-optimal occupancy
4. Grid-stride looping
5. CUDA streams

In the following sections, limitations and the approach used to alleviate them will be described.

4.2.1 Memory dependency stalls

As mentioned in Section 3.2.1.1, global memory has the slowest access times of all memory spaces. To ensure optimal performance when loading data from global memory to on-chip resources, programmers must ensure all accesses are coalesced. A

coalesced access is when all threads in a warp access one cache line. The simplest way is for the k-th thread to access the k-th word in a cache line. For some architectures, accesses through L1 cache with 128-byte lines and L2 cache with 32-byte lines. Other architectures access both L1 and L2 caches with 32-byte lines. Warp sizes of 32 threads can help facilitate memory accesses that are aligned to cache lines. If sequential threads in a warp access memory that is sequential but not aligned with the cache lines then it is considered a misaligned memory access. Misaligned accesses require an additional cache line to be retrieved. While not a factor in this parallel implementation, it is important to mention the other extreme, strided accesses. Non-unit-stride global memory accesses occur when threads within a warp must load, or store, data that is greatly separated in global memory. If data is spread too far apart then L2 cache cannot be utilized. Global accesses then acts in a serial manner and can be detrimental to application performance. Examples of coalesced, misaligned and strided global memory access are provided in Figure 4.2.

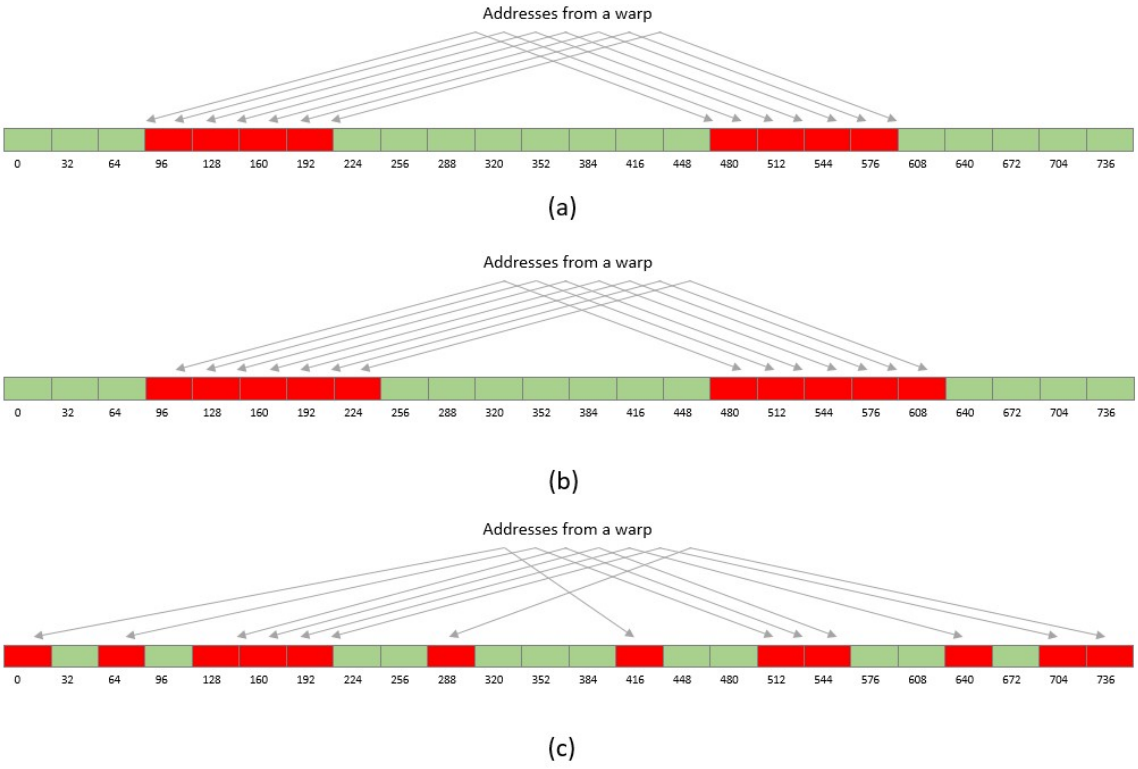


Figure 4.2: **(a)** Coalesced, **(b)** Misaligned, and **(c)** Strided

The naive implementation when profiled with NVIDIA Nsight Compute, as shown in Figure 4.3, shows that the kernels are more memory (Memory %) bound than compute bound (SM %). A suggested best practice is for CUDA kernels to have high utilization with both compute and memory resources. This imbalanced utilization is because there are not enough calculations being performed in each timestep to hide memory latency. There is one comparator and one arithmetic operation per single global memory load. Every timestep, each thread in a warp must retrieve the next value from the prefix-sum in global memory. Because these strides through memory

are strictly consecutive monotonic, periodic global accesses meet the requirement for a coalesced load. As warps traverse through a cache line, global accesses are misaligned.

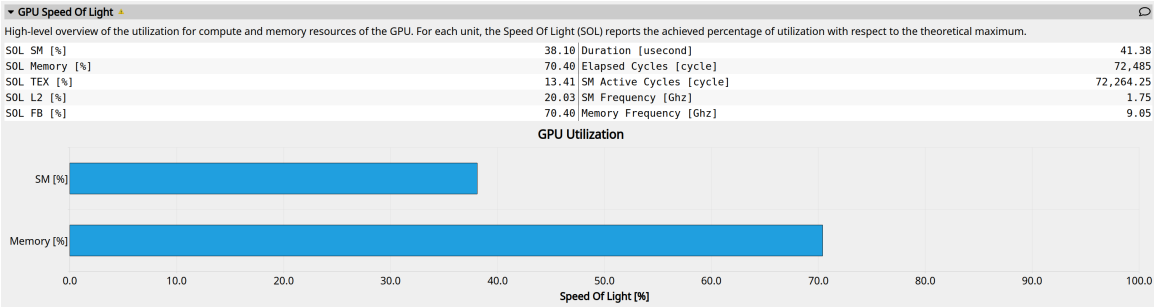


Figure 4.3: GPU Utilization: Naive (green).

For this parallel implementation, the stall with the greatest impact is the CPI Stall 'Long Scoreboard.' This occurs when a warp is waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture) operation. To reduce the number of cycles waiting on L1TEX, one must make sure data accesses are coalesced and optimal for the target architecture, improve cache hit rates by increasing data locality, and move frequently used data to shared memory. Table 4.1 shows that the number of warp cycles per instruction is roughly 3x less when utilizing shared memory.

4.2.1.1 Utilizing Shared Memory

One way to mitigate misaligned memory accesses and reduce the total number of loads from global memory is to take advantage of shared memory. Covered previously, in Section 3.2.1.3, shared memory is on-chip and has higher bandwidth and lower latency than global memory. This is assuming there are no bank con-

flicts between threads, which will be covered later in the section. Instead of threads traversing though global memory to retrieve the next iteration from the prefix-sum, chunks of data can be stored in a coalesced manner in shared memory. Threads can then traverse shared memory with little penalty to performance. As an example, a warp can load two consecutive coalesced accesses into shared memory, or a total of 64 elements. Then threads can step through shared memory 32 times, comparing the prefix-sum value to its random number. Now, the quantity of computation per global memory load has been increased by a factor of 16. This is because there are two coalesced loads, versus a combination of 32 coalesced and misaligned loads [44]. Figure 4.4 shows improved GPU utilization when using shared memory to reduce the number of global memory loads. The kernel using shared memory uses 55% more compute resources, while reducing memory utilization by 13%. This reduction in memory utilization is due to sub-optimal occupancy, which will be explained in detail later in this chapter.

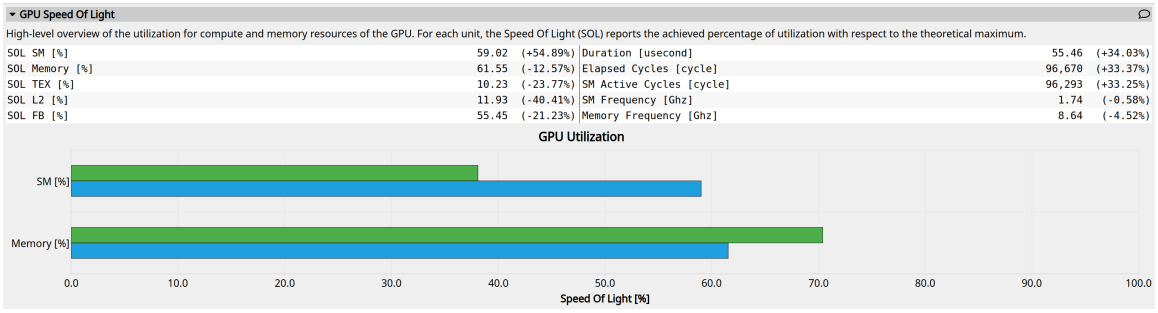


Figure 4.4: GPU Utilization: Naive (green) and Shared (blue).

It is important to note that care should be taken when utilizing shared memory on the SM. Because it is on-chip, shared memory has a much higher bandwidth and

lower latency than local or global memory. To achieve this bandwidth, memory is divided into equally sized memory banks that can be accessed simultaneously. However, if multiple memory requests access the same bank, the request will be serialized. The one exception is when multiple threads within a warp access the same shared memory location. The result is then broadcast to those threads. Devices with compute capability 3.x and higher have two banking mode options, successive 32-bit or 64-bit words. Because the calculations in this paper deal with single precision, 32-bit mode is chosen. The implementation presented does not introduce any bank conflicts because each warp is allocated a shared memory space mutually exclusive from the other and individual threads, in each warp, never access the same memory bank in a given cycle.

4.2.1.2 Cooperative Groups

In the naive implementation, each thread works independently of the adjacent thread, in a warp, traversing the prefix-sum in global memory, and does so until its condition is satisfied. At that point, the thread becomes inactive and waits for all other threads to finish before the block is ejected and new work is scheduled. With a shared memory implementation, threads in a warp must work together. Each warp traverses the prefix-sum in global memory reading two 32 strictly consecutive monotonic element chunks of data at a time. The chunks are loaded into shared memory. Then each thread steps through shared memory until it reaches the end of the prefix-sum, satisfies its condition statement, or traverses its 32 element section. If at least one thread has not reached the satisfied condition, two more chunks of

data must be loaded into shared memory. For all 64 values to be loaded into shared memory, all threads in a warp must remain active.

Table 4.1: Warp State Statistics: Instructions/Issues Per Cycle

Warp State Statistics		
Warp Cycles	Naive	Shared
Issued Instructions	27.05	9.14
Issue Active	29.56	11.41
Executed Instruction	27.09	9.12

For all threads in a warp to know if at least one thread requires more data from the prefix-sum, *intra-wrap* communication is required. This can be accomplished using *cooperative groups*, introduced in CUDA 9.0 [32]. Cooperative groups allow programmers the ability to define and synchronize groups of threads smaller than thread blocks and enable greater performance through design flexibility and *collective* group-wide primitives. Of the warp-level functions provided, *any()* and *all()* can be used to check the bit mask to determine if a thread in a warp has satisfied its condition. The *any()* function returns true if any thread in a warp satisfies the condition, and *all()* returns true if all threads satisfy the condition. Figure 4.5 shows how cooperative groups extend the CUDA programming model to allow for flexible, dynamic grouping of threads. It shows an example of a 64 thread block being partitioned into two 32 thread warp groups, and then being further partitioned into 4 thread groups.

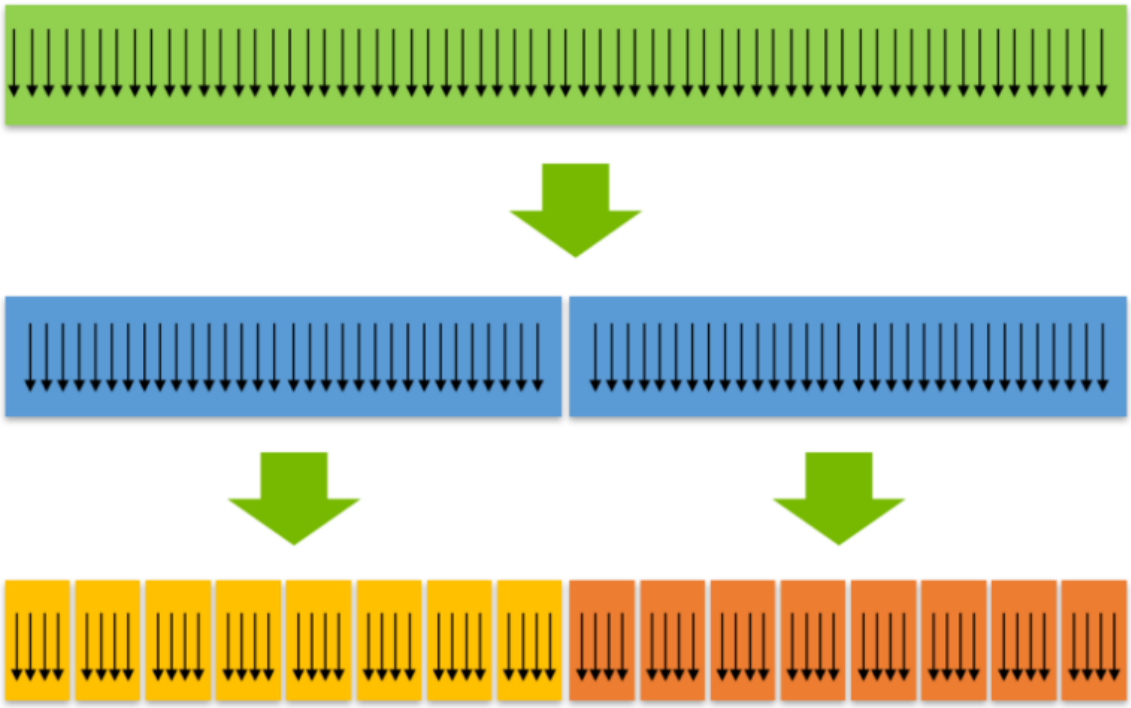


Figure 4.5: Cooperative groups allows for more flexible programming.

4.2.1.3 Software Prefetching

Now that the Up and Down kernels are efficiently using shared memory, profiling the code shows that the bottleneck lies with loading data from global memory to shared memory. The data path from global to shared memory is handled by the *LDG* and *STS* instructions. For compute capabilities 5.X and later, *LDG* loads data from global memory through L2 cache, and L1 cache depending on the architecture, to a register. Instruction *STS* then stores a value from a register into a shared memory bank. Therefore, loading data straight from global memory to shared memory requires two steps. These two steps combined with synchronization between loads and

reads can cause unnecessary delays. Synchronization is required when using shared memory to ensure that threads are not reading from and/or writing to shared memory while other threads are accessing the same memory banks. Without synchronization, race conditions can occur on data in shared memory. A race condition occurs when two or more threads try to modify data in the same shared or global memory location at the same time. Pseudocode is provided in Technique 1, which shows the basic steps required when utilizing shared memory.

Technique 1 No Prefetch

```
1: Initial kernel
2: for all Threads do
3:     while True do
4:         Global memory → shared memory
5:         Synchronize threads
6:         Perform math calculations on shared memory
7:     end while
8: end for
```

This workflow can be modified to avoid mathematical delays. Instead of loading data from global memory directly into shared memory each timestep, it can be broken up into multiple steps. First, load the initial set of data into shared memory during the first timestep. Next, preload the next set of data into registers and then perform math calculations on the previous set. Once calculations are finished, load the data from the registers into shared memory. Then synchronization can occur. This process will be performed each timestep and is known as *software prefetching*. It is a common technique to adapt a program to take advantage of hardware designs. Pseudocode for this modified workflow is provided in Technique 2.

Technique 2 Prefetching

```
Initial kernel
for all Threads do
    Global memory → shared memory
    while True do
        Global memory → registers (Preload next set)
        Perform math calculations on shared memory
        Registers → shared memory
        Synchronize threads
    end while
end for
```

Figure 4.6 shows that when using software prefetching, compute and memory utilizations increase by roughly 3% over using shared memory alone.

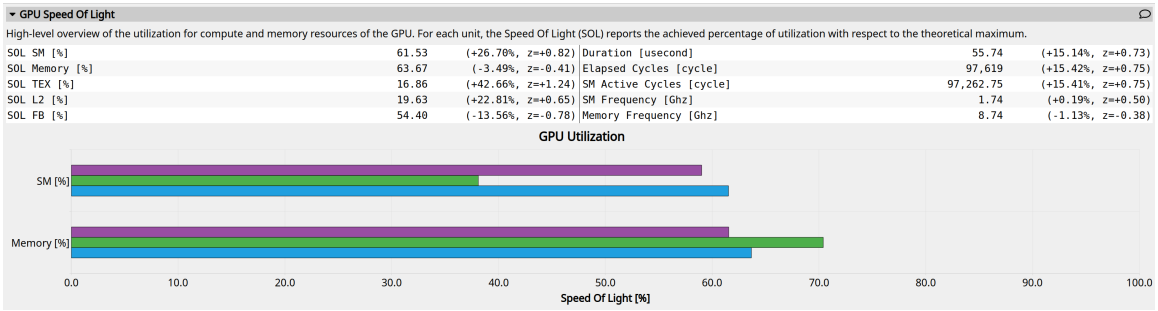


Figure 4.6: GPU Utilization: Naive (green), Shared (purple), and Prefetch (blue).

4.2.2 Thread Execution Efficiency

While utilizing shared memory reduces the overall quantity of global memory loads, it can introduce a new issue caused by a false dependency on shared memory. The issue is further exacerbated because of low thread execution efficiency related to the stochastic nature of the implementation. As mentioned previously, threads in a block are bundled into fixed-size warps. Maximum thread execution efficiency is achieved when 100% of a warp's threads are active. Less than 100% means threads are

inactive due to sub-optimal launch, early return, or predicated off due to control flow divergence; hence, a variable-task length problem. Due to the stochastic nature of the parallel implementation, its efficiency is lessened by early return of thread through control flow divergence. As threads satisfy the comparator in a given while-loop, they enter an inactive state. This is critical to GPU performance because a warp cannot exit until all threads have finished the last instruction, and a block can not exit a SM until all warps have finished.

In the worst case, where there are 1024 threads (32 warps with 32 threads) in a block, the maximum allowed, if only one thread is active then that means the remaining 31 warps are active and consuming resources, even though their threads may be inactive [44]. This decreases the number of eligible warps per scheduler. Eligible warps are a subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, one should either increase the number of active warps or reduce the time the active warps are stalled. The stochastic nature allows for the possibility of one thread executing while the rest of the block remains idle, limiting available resources. Due to each thread having a variable-task length caused by the while-loop, control flow divergence and early return are simply inherent to the implementation.

Additionally, variable-task length threads that require synchronization on shared memory can produce a false dependency. That is where there is no data dependence between each warp in a block, but they have to wait for each other to complete to transition to the next instruction because they are sharing shared memory. This

means that threads in a block cannot execute new instructions until all threads reach the barrier. The solution to false dependency is to reduce the number of threads in a block. In Figure 4.7, the basic execution modes when multiple warp access the same shared memory array are presented in diagram (a). Diagram (b) shows that an imbalanced workload between warps causes false dependency on shared memory. Diagram (c) shows how when imbalanced workloads are divided into smaller blocks, warps with smaller workloads are not required to wait.

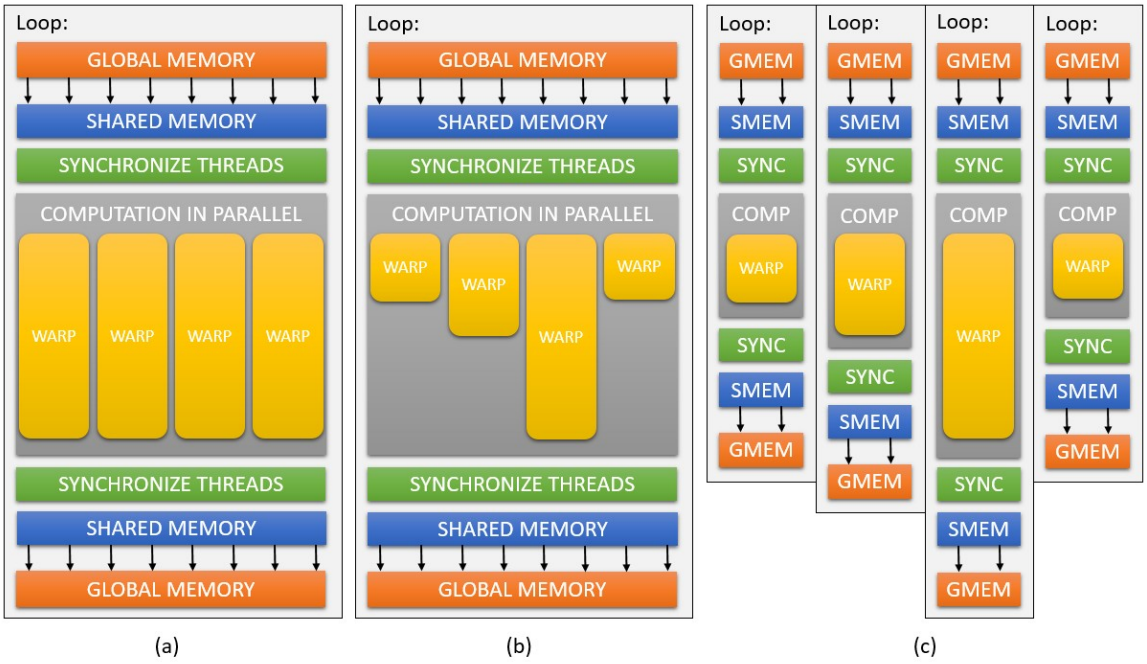


Figure 4.7: False Dependency on Shared Memory where **(a)** basic execution mode, **(b)** false dependency, and **(c)** less warps per block to remove false dependency.

Fixing the false dependency on share memory also mitigates low thread execution efficiency by reducing the possibility of a long running thread, preventing a large block from being retired from an SM. Making the block size equal to the number of

threads in a warp eliminates the scenario of one active thread in a warp in a block of multiple warps. Now if a block has only one active thread it only affects that individual warp and not many. This increases the ratio of eligible warps to active warps; therein boosting performance.

4.2.3 Sub-optimal Occupancy

Another important consideration when programming on a GPU is *occupancy*. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. It helps measure a kernels ability to utilize resources of an SM. The number of blocks that can execute concurrently on an SM is limited by multiple factors, such as the size of blocks, the quantity of warps and registers, and the amount of shared memory. A subset of the resources available on an NVIDIA GTX 1080, the GPU used in this research, is given in Table 4.2.

Table 4.2: NVIDIA GTX 1080 Hardware Specifications: CC 6.1.

Property	Value
Architecture	Pascal
Clock Rate	1860 MHz
CUDA Cores	2560
Global Memory	8GB
Shared Memory	48KB
Register Count	65,536
SM Count	20
Max Threads Per SM	2048
Max Blocks Per SM	32
Max Warps Per SM	64

If a block requires too much of any one resource, it limits the number of active blocks on that SM. As previously stated, in order to minimize the effect of low thread execution efficiency, one must increase the number of active warps or reduce the time the active warps are stalled. This is accomplished by setting the block size equal to that of a warp, or 32 threads as stated in Section 4.2.2. On an SM with compute capability 6.1, like the GTX 1080, the maximum number of active blocks is 32 and the maximum number of active warps is 64. That means there must be at least two active warps per active block to reach 100% occupancy. By limiting the number of threads per block to increase thread execution efficiency, the theoretical occupancy of the Up and Down kernels are limited to 50%, meaning 50% of the GPU is underutilized at any given time during kernel execution.

In order to increase occupancy, without removing any previously mentioned optimizations, warp-level functionality in cooperative groups can be used to increase the number of warps per block to two. This can be achieved by doubling the amount of shared memory so each warp works on independent data. Using warp synchronization instead of block synchronization eliminates false dependency on shared memory, while increasing block size. Using two warps per block does slightly increase the possibility of low thread execution efficiency, but there will be zero negative impact because the method is utilizing warps that would otherwise be inactive. Figure 4.8 shows that when the number of warps per block is doubled to two, compute and memory utilizations increase by roughly 7% over shared memory and prefetching.

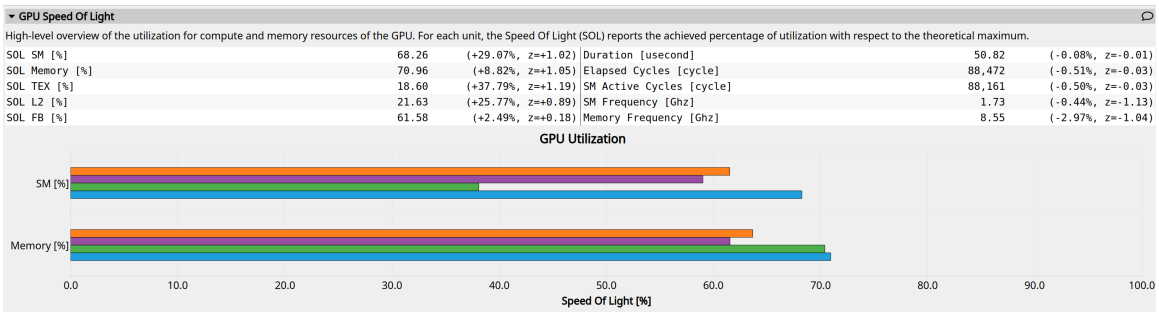


Figure 4.8: GPU Utilization: Naive (green), Shared (purple), Prefetch (orange), vs Two Warps per Block (Blue).

4.2.4 Grid-Stride Looping

When programming on GPUs, different implementation techniques can indirectly affect performance. There are two popular techniques to implement kernels using CUDA, *monolithic kernels* and *grid-stride loops*. A monolithic kernel utilizes a single large grid containing one thread per element and processes the entire array in one pass. Grid-stride loops deploy a small grid of threads and loops over the data one grid at a time. They let you decouple the size of your CUDA grid from the data size it is processing. In Figure 4.9, diagram (a) displays a monolithic kernel where the quantity of threads in a grid matches the data set. Diagram (b) shows the grid-stride looping technique covering the same data set with half the number of threads in a grid.

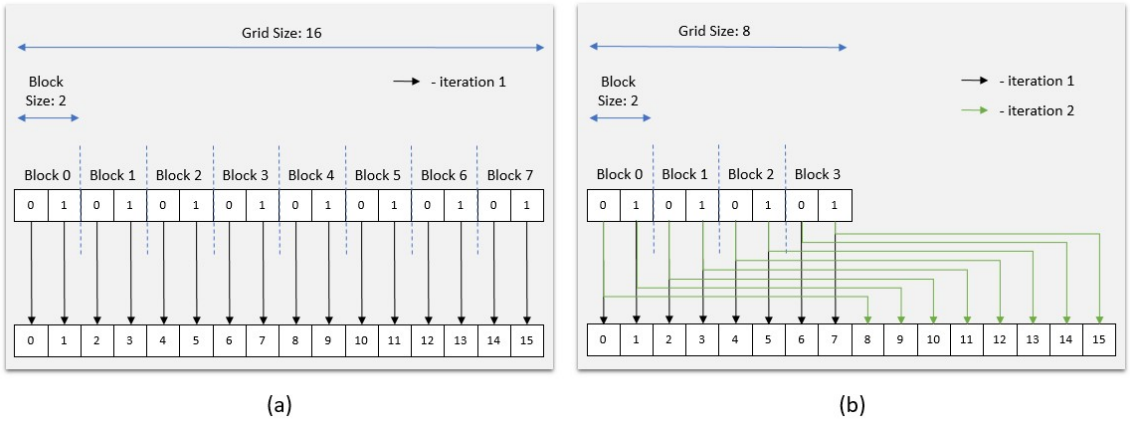


Figure 4.9: **(a)** Monolithic kernel and **(b)** Grid-stride looping.

Grid-stride loops also provide additional tuning capability by allowing configurable block-grid sizes per kernel per device. The goal is to achieve optimal persistence of a thread block on an SM. It is often the preferred technique because it reduces the overhead of launching, maintaining, and destroying additional blocks. It is also regarded as being more flexible, scalable, and portable. Additionally, it has the ability to handle the case when there is a mismatch between grid size and the size of the data set. Examples of both monolithic and grid-stride looping kernels, using SAXPY computation, are shown in Technique 3 and Technique 4, respectively.

Technique 3 Monolithic

Require: $n \leftarrow$ size of array
1: tid = blockIdx.x * blockDim.x + threadIdx.x
2: **if** $tid < n$ **then**
3: y[i] = a * x[i] + y[i]
4: **end if**

Choosing the optimal grid size for grid-stride looping is kernel dependent but it is suggested to be a multiple of the number of SMs. This methodology helps

Technique 4 Grid-Stride Loop

Require: $n \leftarrow$ size of array

```
1: tid = blockIdx.x * blockDim.x + threadIdx.x
2: gridSize = blockDim.x * gridDim.x
3: for i = tid; i<n; i += gridSize do
4:     y[i] = a * x[i] + y[i]
5: end for
```

reduce any tail effect caused by inactive SMs as a kernel finishes a task. This parallel implementation also requires the quantity of blocks to be a multiple of 32 in order to utilize all 32 active blocks of an SM. Therefore, the minimum number of blocks should be 32 times the number of SMs, or 640 on a GTX 1080. Using 64 threads in a block gives a grid size of 40,960. Particle sets with a quantity less than this grid size will under utilize the GPU. Table 4.3 provides a list of grid sizes used in this research.

Table 4.3: Grid-Stride Looping Sizes

Grid Stride Looping Sizes		
Number of Particles	Blocks per Grid	Threads per Grid
1024	16	1024
2048	32	2048
4096	64	4096
8192	128	8192
16384	256	16384
32768	512	32768
65536	640	40960
131072	640	40960
262144	1280	81920
524288	1280	81920
1048576	2560	163840

That multiplier's performance can vary depending on compute and memory utilization of the kernels themselves. Using grid-stride looping also allows data reuse. As an example, because the systematic resampling requires the same randomly generated number for each particle, the generation can be performed outside the grid-stride loop and used by all subsequent loops.

4.2.5 CUDA Streams

The parallel implementation presented in this paper consists of two kernels that are mutually exclusive. Fortunately, CUDA provides an additional level of concurrency in the form of *streams*. A stream is a sequence of operations that executes in issue-order on the GPU. Different streams may execute their commands, or kernels, concurrently or out of order with respect to each other. It is important to note that any kernel not explicitly designated to a specific stream is launched in the *default* stream, which is synchronous, or blocking. Running the two kernels in independent asynchronous, or non-blocking, streams will not increase individual kernel occupancy, but now the GPU has an opportunity to run blocks from both kernels concurrently as resources become available [44]. A visual aid showing algorithmic concurrency is provided in Figure 4.10.

There are factors that effect streams' ability to run concurrently. First, SMs must wait until enough resources are available to launch a kernel's warps. Figure 4.11 shows that even though the Up and Down kernel are in independent streams they don't execute with perfect concurrency. There is an initial time period where the Up kernel is utilizing all available resources on the GPU. While overlapping kernels may

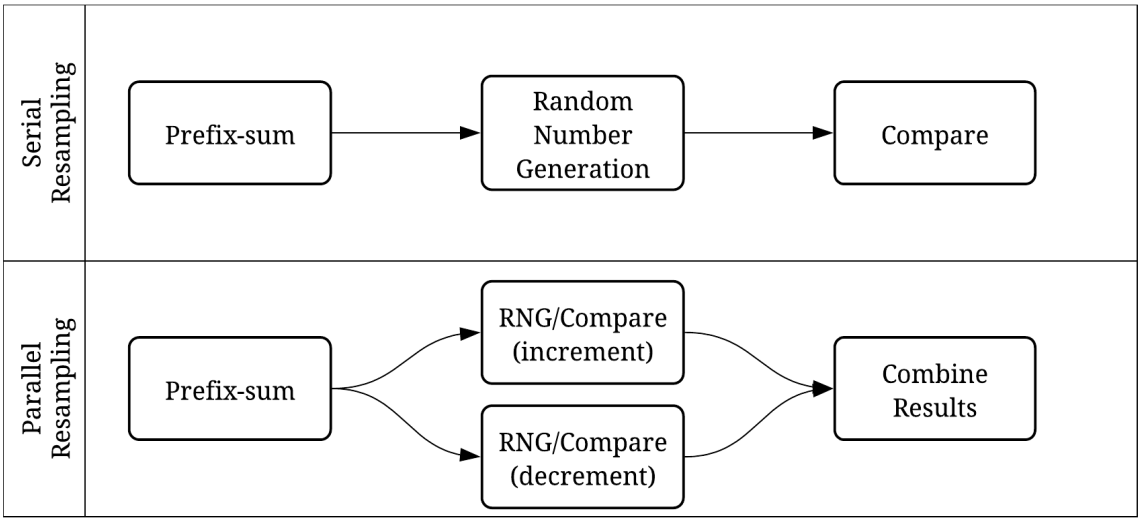


Figure 4.10: Algorithm flowchart comparing serial and parallel implementations.

slightly extend individual kernel execution time, it reduces overall execution of both kernels by allowing the hardware to utilize all available resources and removes any tail effect from the kernel launched first. This tail effect occurs as a kernel retires blocks from the SM and the SM becomes underutilized.

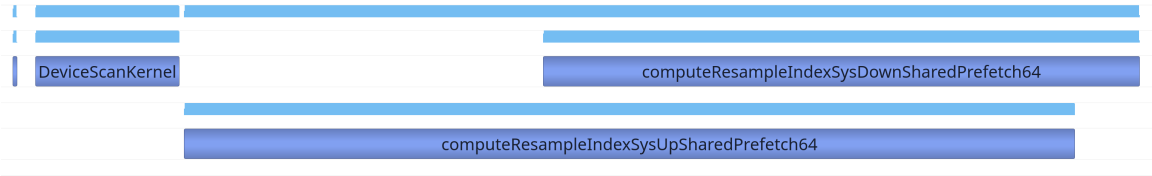


Figure 4.11: NVIDIA Nsight Compute Profiler showing Up and Down kernels running concurrently, processing 2^{20} particles.

Henceforth, the combination of shared memory, software prefetching, and utilizing two warps per block will be referred to as the improved implementation. Timing comparisons of the naive and improved implementations of Algorithm 12 will be presented in Section 6.1. Also, the source code is provided for the naive implementation

in Section A.1 and Section A.2 and the improved implementation in Section A.7 and Section A.8.

4.2.6 Algorithmic Efficiency

In computer science, algorithmic efficiency is the property of an algorithm which relates to the number of computational resources used by said algorithm. The algorithm is considered efficient if the resource consumption of concern falls at or below a particular threshold. In this section, the serial and parallel implementations will be compared across three metrics; time complexity (the amount it takes to finish the problem), work efficiency (how much work is performed by all threads in the algorithm), and space efficiency (the amount of memory allocation required). For simplicity, these comparisons will be analyzed on a parallel random-access machine (PRAM). Doing so neglects many real-life practical issues, such as memory access time, synchronization, and communication. It also removes hardware constraints like the number of processors. Removing such complexities allows for a more general understanding of the best and worst case scenarios from a mathematical perspective. Also, this section will be focused on resampling and will negate the prefix-sum and random number generation because those trends are well understood.

4.2.6.1 Time Complexity

Before proceeding, a baseline must be established. The baseline will be the serial algorithm, as stated earlier in Section 2.4.2 and Section 2.4.3, which has a time complexity of $T_s = \mathcal{O}(N)$, or linear time. The serial algorithm is highly determin-

istic but the parallel algorithm's performance is data dependent. Unlike the serial algorithm, the parallel can be broken down into best and worst case scenarios. Best case is when particle variance is at its lowest and worst case is when variance is at its highest. The time complexities for the naive parallel implementation are shown below in (4.1) and (4.2)

$$T_{nb} = \mathcal{O}(1). \tag{4.1}$$

$$T_{nw} = \mathcal{O}(N). \tag{4.2}$$

where T_{nb} is the naive best case time complexity and T_{nw} is the naive worst case. In the best case scenario, (4.1), all threads, in both kernels, perform one work cycle before returning to an inactive state, as shown in Figure 4.12. The worst case scenario will occur when all the elements in the prefix-sum have a value of zero except element $N-1$ which has a value of 1. This is the maximum variance capable in the problem. In the Down kernel, all threads will perform one cycle and return to an inactive state. While in the Up kernel, all threads traverse the entire array to the right, requiring the first thread to do so N times, as shown in Figure 4.13. To ensure that the first thread in the Down kernel does not read out of bounds memory, it is set to inactive by default. Therefore, the algorithmic speedup, $S = \frac{T_s}{T_p}$, can range from $S = \mathcal{O}(N) \rightarrow \mathcal{O}(1)$.

Next, the improved implementation has a slightly different time complexity due to the iterations through shared memory. In the shared memory implementations, a warp loads two 32 element chunks into a shared memory space. All threads must be active during this load. Therefore, threads cannot be retired immediately after

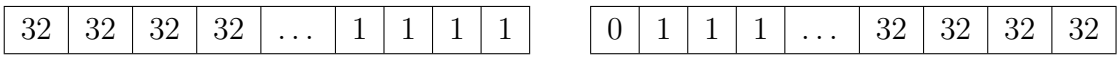
Up: Operations per thread Down: Operations per thread

Figure 4.12: Best case scenario for naive implementation

Up: Operations per thread Down: Operations per thread

Figure 4.13: Worst case scenario for naive implementation

satisfying the comparator. They must finish all iterations in the for loop before all bit masks are checked. If all threads have satisfied the comparator at the end of the for-loop the warp can be retired. This means a minimum of 32 iterations will be executed instead of only one for the best case, as shown in Figure 4.14. The exception to this is the last warp in the Up kernel and the first warp in the Down kernel. In these warps, threads load data from global memory directly into registers. This ensures threads in those warps don't traverse out of bounds.



Up: Operations per thread Down: Operations per thread

Figure 4.14: Best case scenario for improved implementation

The worst case time complexity in the improved Up kernel will match that of the naive implementation because additional work cannot be performed due to the boundary conditions, as shown in Figure 4.15. The Down kernel will be the same for both cases.

N	N-1	N-2	N-3	...	3	2	1

Up: Operations per thread

0	1	1	1	...	32	32	32	32

Down: Operations per thread

Figure 4.15: Worst case scenario for improved implementation

Requiring all threads to remain active and evaluate the for-loop even after the comparator has been satisfied is less detrimental to performance than applying branching to check the bit mask at loop iteration. The time complexities for the improved parallel implementation are shown below in (4.3) and (4.4).

$$T_{ib} = \mathcal{O}(32). \tag{4.3}$$

$$T_{iw} = \mathcal{O}(N). \tag{4.4}$$

Therefore, the improved implementation's algorithmic speedup, S = $\frac{T_s}{T_p}$, can range from $S = \mathcal{O}(N) \rightarrow \mathcal{O}(32)$.

4.2.6.2 Work Efficiency

An algorithm's work efficiency considers the total amount of work that is performed across all threads to arrive at the correct answer. Again the baseline will

be the serial implementation, which is $W_s = \mathcal{O}(N)$, because a single CPU thread will perform all the work. The work efficiency is $WE = \frac{W_s}{W_p}$, where W_p is the work performed by the parallel implementation. As with time complexity, work performed by the naive parallel kernels will have a best and worst case scenario, represented by W_{nb} (4.5) and W_{nw} (4.6), respectively.

$$W_{nb} = \underbrace{N}_{\text{Up}} + \underbrace{N-1}_{\text{Down}} \tag{4.5}$$

$$W_{nw} = \underbrace{\frac{(N-1)(N)}{2} + N}_{\text{Up}} + \underbrace{N-1}_{\text{Down}} \tag{4.6}$$

Using L'Hôpital's rule, the limits of the work efficiency can be evaluated as $N \to \infty$.

$$\begin{aligned}
WE_{nb} &= \frac{W_s}{W_{nb}} \\
\lim_{N\to\infty} &= \frac{W_s}{W_{nb}} \\
&= \frac{N}{N+N-1} \\
&= \frac{N}{2N-1} \\
&= \frac{1}{2-\frac{1}{N}} \\
&= \frac{\lim_{N\to\infty}(1)}{\lim_{N\to\infty}(2-\frac{1}{N})} \\
&= \frac{1}{2}
\end{aligned}$$

$$
\begin{aligned}
WE_{nw} &= \frac{W_s}{W_{nw}} \\
\lim_{N\to\infty} &= \frac{W_s}{W_{nw}} \\
&= \frac{N}{\frac{(N-1)(N)}{2} + N + N - 1} \\
&= \frac{2N}{N^2 + 3N + 2} \\
2 * \lim_{N\to\infty} &= \frac{N}{N^2 + 3N + 2} \\
&= \frac{\frac{1}{N}}{1 + \frac{3}{N} + \frac{2}{N^2}} \\
&= 2 * \lim_{N\to\infty} \left(\frac{\frac{1}{N}}{1 + \frac{3}{N} + \frac{2}{N^2}} \right) \\
&= 2 * \frac{\lim_{N\to\infty}(\frac{1}{N})}{\lim_{N\to\infty}(1 + \frac{3}{N} + \frac{2}{N^2})} \\
&= 2 * \frac{0}{1} \\
&= 0
\end{aligned}
$$

While the algorithm has the potential to significantly improve time complexity, it comes at the cost of work efficiency. Because there are two while-loops in the parallel implementation, best case takes double the amount of work as $N \to \infty$. For the worst case, the work per thread in the parallel implementation is effectively serialized. Therefore, it makes sense that as $N \to \infty$, there will be zero work efficiency.

For the improved implementation, the additional work per thread, due to shared memory utilization, must be taken into account. Work complexity will be computed in terms of warps where WS equals the warp size of 32, and F equals the number of iterations of the for loop, or 32. Because of software prefetching,

WS is multiplied by two. The best and worst case are presented in (4.7) and (4.8), respectively. Notice that in the (4.7) the WS is multiplied by a factor of 2 to account for software prefetching. In (4.8), we can reuse the time complexity equation to calculate the worst case for the Up kernel because the amount of work will be the same.

$$W_{ib} = \underbrace{\big((N - (WS * 2)) * F\big) + (WS * 2)}_{\text{Up}} + \underbrace{\big((N - (WS * 2)) * F\big) + ((WS * 2) - 1)}_{\text{Down}} \tag{4.7}$$

$$W_{iw} = \underbrace{\frac{(N-1)(N)}{2} + N}_{\text{Up}} + \underbrace{\big((N - WS) * F\big) + (WS - 1)}_{\text{Down}} \tag{4.8}$$

Using L'Hôpital's rule, the limits of the work efficiency can be evaluated as $N \to \infty$.

$$\begin{aligned}
WE_{ib} &= \frac{W_s}{W_{nb}} \\
\lim_{N\to\infty} &= \frac{W_s}{W_{ib}} \\
&= \frac{N}{(N-64) * 32 + 64 + (N-64) * 32 + 63} \\
&= \frac{N}{64N - 3969} \\
&= \frac{1}{64 - \frac{3969}{N}} \\
&= \frac{\lim_{N\to\infty}(1)}{\lim_{N\to\infty}(64 - \frac{3969}{N})} \\
&= \frac{1}{64}
\end{aligned}$$

$$
\begin{aligned}
WE_{iw} &= \frac{W_s}{W_{nw}} \\
\lim_{N\to\infty} &= \frac{W_s}{W_{iw}} \\
&= \frac{N}{\frac{(N-1)(N)}{2} + N + (N-64) * 32 + 63} \\
&= \frac{2N}{N^2 + 65N + 3970} \\
2 * \lim_{N\to\infty} &= \frac{N}{N^2 + 65N + 3970} \\
&= \frac{\frac{1}{N}}{1 + \frac{65}{N} + \frac{3970}{N^2}} \\
&= \frac{\lim_{N\to\infty}(\frac{1}{N})}{\lim_{N\to\infty}(1 + \frac{65}{N} + \frac{3970}{N^2})} \\
&= 2 * \frac{0}{1} \\
&= 0
\end{aligned}
$$

The best case requires 32 times more work than the naive implementation, while the worst case has the same efficiency.

4.2.6.3 Space Efficiency

For space efficiency, the parallel implementation requires memory allocation to store resampling indexes for both the Up and Down kernels in global memory. This means the parallel implementation requires twice the memory footprint. For the worst case that would be 2^{20} elements * 4 bytes, for single precision, requiring 4MB of additional storage, which is not a significant impact on todays devices.

CHAPTER 5

PSEUDORANDOM NUMBER GENERATORS

Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.

—John von Neumann

To ensure that Monte Carlo simulations provide accurate approximate numerical solutions, the RNGs that provide stochastic input to each trial must be indistinguishable from a true random number source. In reality, RNGs in simulations do not produce random numbers, but use deterministic algorithms to produce numbers that mimic randomness. This is a favored characteristic of RNGs because it allows repeatability. Repeatability in Monte Carlo simulations allow engineers to reproduce Monte Carlo variations that cause unwanted effects for further analysis. The main method used to produce samples is to first generate uniform random numbers, then apply a transform to convert the uniform numbers into samples of the desired distribution. RNGs can be classified into three groups:

1. True Random Number Generators

   - Truly stochastic and are primary using in cryptography and secure sockets layer (SSL) protocols. Random numbers are usually generated on hardware

devices through a physical process based on random noise signals, such as thermal noise, the photoelectric effect, beam splitter, or some other quantum phenomena.

2. Quasirandom Number Generators

   - Used to evenly fill an n-dimensional space with points, without clumping points; also known as low-discrepancy. Although they can we used in Monte Carlo simulations they were not considered for this research.

3. Pseudorandom Number Generators

   - Designed to model a true RNG but are actually computed using a deterministic algorithm. This characteristic of determinism makes them appealing in Monte Carlo simulation because of reproducibility.

This work focuses on pseudorandom number generators (PRNGs). For greater insight into algorithmic selection criteria and RNG constructions refer to [45]. This chapter will briefly examine implementation requirements of RNGs for parallel processing, describe the PRNGs offered by cuRAND and their implementation details, and demonstrate their performance in this research.

5.1 Background

Much work over the past few decades has gone into the development of random number generators on CPUs. These optimized PRNG implementations take advantage of large instruction sets and large amounts of fast memory available to

the core(s). Below are a few general requirements that PRNGs should satisfy for simulation usage.

1. Execution speed
    - Generators should be able to create random numbers in a timely manner.
2. Long period
    - Because deterministic generators eventually repeat, longer spans in between repetitions are highly desirable.
3. Repeatability
    - Random numbers should be reproducible between simulations runs and various systems.
4. Good statistical quality
    - As previously stated, the goal of PRNGs is to be indistinguishable from true random number generators. Therefore, it is important that signs of correlation or patterns are not shown.

Due to limited on-chip resources, as described in Chapter 3, these CPU implementations do not map well to GPU architecture in their naive form. In parallel environments utilizing multiple CPU cores and software such as OpenMP [46], a central monitor can be extended to create multiple streams that are considered statistically independent. However, creating a large number of streams for GPUs that can have thousands of cores can cause excessive overhead and create a bottleneck.

A naive method of implementing RNGs is to generate a large quantity of numbers on the host or device and store them in device memory to be utilized by threads. This method suffers from memory bandwidth. First, if the numbers were to be generated on the host they would need to be transferred to device memory. Second, once numbers are in device memory, in order to be utilized by a kernel, data must be transferred to on-chip resources, whether that be registers or shared memory.

The preferred method is to use multiple PRNGs, or copies of the same PRNGs, with substreams starting at different states. If these substreams are able to imitate individual independent streams, reproducibility can be achieved by distributing substreams among Monte Carlo runs. They can be created using multiple PRNGs or by taking a single sequence and splitting it into smaller sequences that are used by threads. It is advised that the starting points of the substreams be sufficiently spread out to ensure no overlap occurs. For a PRNG to be effective in Monte Carlo simulations on a GPU, additional requirements include [47]:

1. Stream division
    - The generator must be able to divide a stream into substreams that can be utilized on parallel nodes with no overlap.
2. Substream quality
    - Substreams must retain good statistical quality and appear to be an independent stream of random numbers.

The NVIDIA cuRAND library offers multiple ways to achieve the requirements listed. The next section will describe their implementation techniques.

5.1.1 Single PRNG - Multiple Substreams

The most common approach to create multiple streams is from a single PRNG split into equally spaced streams. According to the cuRAND Programming Guide, that approach is considered to have higher statistically quality [32]. Each experiment will be assigned a unique seed and each thread will be designated a unique sequence number. It is also advised that sequence numbers be assigned in a monotonically increasing way. A common practice is to assign the thread index to the sequence number. A code snippet is provided below.

```
1  __global__ void generateRandomNumbers ( int n, ull seed , float * devRanNum ) {
2
3    int t = threadIdx.x, b = blockIdx.x, g = gridDim.x;
4
5    for ( unsigned int tid = b * blockDim.x + t; tid < n; tid += blockDim.x * g ) {
6
7      curandStateXORWOW_t localState { };
8      curand_init ( seed , tid , 0, & localState );
9      devRanNum [ tid] = curand_uniform ( & localState );
10   }
11 }
```

In the sample code, the variable *localState* is created to hold a local on-chip copy of an XORWOW state. (XORWOW is a type of PRNG that will be described in further detail in the section.) Next, the state is initialized using `curand init()` with its parameters listed in Table 5.1.

The same seed is passed to all threads from the host, where a seed can easily be created using the `clock()` command. The initialization function also accepts the sequence number, in this case *tid*, which is the thread index created by grid-stride

Table 5.1: `curand_init()` Parameters

`curand_init()`		
Type	**Parameter**	**Description**
unsigned long long int	*seed*	Arbitrary bits to use as a seed
unsigned long long int	*sequence*	Subsequence to start at
unsigned long long int	*offset*	Absolute offset into subsequence
curandStateXORWOR_t	*state*	Pointer to state to initialize

looping. Finally, a uniform random number is generated with `curand_uniform()` and stored back in global memory. It should be noted though that using `clock()` to generate the seed will not allow repeatability between runs.

With this implementation in cuRAND, each thread is provided with an independent substream equally spaced apart. If the given seeds are the same across all threads, the starting state for each thread is calculated as such:

$$state = 2^{67} * sequence + offset \tag{5.1}$$

With XORWOW as an example, the main sequence has a period of 2^{190}. Therefore, monotonically increasing the sequence id with a zero-based thread index provides a total of 2^{123} independent streams. Then each thread will have a range of 2^{67} states before transitioning into the next substream.

5.1.2 Single PRNG - Multiple Seeds

If jumping through the sequence is too taxing on performance, an alternative method is to initialize each thread with a different seed and set the sequence number to a constant zero for all threads. One must ensure that the PRNG has a relatively large period compared to the number of values generated. Applying different seeds to each thread initializes the thread to some random position within the period. A code snippet is provided below.

```
1  __global__ void generateRandomNumbers ( int n, ull seed , float * devRanNum ) {
2
3    int t = threadIdx.x, b = blockIdx.x, g = gridDim.x;
4
5    for ( unsigned int tid = b * blockDim.x + t; tid < n; tid += blockDim.x * g ) {
6
7      curandStateXORWOW_t localState { };
8      curand_init ( seed + tid , 0, 0, & localState );
9      devRanNum [ tid] = curand_uniform ( & localState );
10   }
11 }
```

Notice in this snippet, the thread index is added to the seed to allow each thread to have a different seed, while the sequence number is set to zero. Unfortunately, this method does not guarantee uncorrelated random numbers. The limitations of the method come from the fact that interactions between the seeding algorithm and the mathematical properties of the PRNG can be erratic. This means that it is possible for multiple threads to begin at the same location. Although possible, this phenomena is highly improbable. Given a period p, with s number of streams, each with a length ℓ, the probability P that there will be zero collisions can be given as

$$P = \left(1 - \frac{s * \ell}{p}\right)^{s-1} \tag{5.2}$$

This gives

$$\ln P = (s-1) * \ln(1-x) \tag{5.3}$$

$$= \frac{s * \ell}{p} \tag{5.4}$$

$$\tag{5.5}$$

When x is small,

$$P \approx -(s-1)x \tag{5.6}$$

Therefore,

$$1 - P \approx -\ln P \tag{5.7}$$

$$\approx (s-1)x \tag{5.8}$$

$$\approx \frac{s^2 * \ell}{p} \tag{5.9}$$

In the case of XORWOW, the main sequence has a period of 2^{190}. The highest number of threads utilized for this research is 2^{20} because of stability issues with

prefix-sum in combination with single precision. This instability will be described in greater detail in Chapter 6. Letting s = $\ell = 2^{20}$ the probability is

$$1 - P \approx \frac{(2^{20})^2 * 2^{20})}{2^{190}} \tag{5.10}$$

$$\approx 7.3468e^{-40} \tag{5.11}$$

If initialization is performed each time step, where $\ell = 1$, probability can be reduced even further giving

$$1 - P \approx \frac{(2^{20})^2 * 1)}{2^{190}} \tag{5.12}$$

$$\approx 7.0065e^{-46} \tag{5.13}$$

This approach will also prevent the possibility of recursive collisions.

5.1.3 Statistical Experiments

To ensure that numbers generated by RNGs meet statistics requirements, they can be analyzed by a series of battery tests. For many years, the test suite of choice was DIEHARD [48], but as technology advanced and higher precision become readily available this suite was replaced with TestU01 [49]. TestU01 is used to compare statistical properties of pseudorandom sequences to those expected from a true random process. For clarity, it is important to state that even if a PRNG passes all tests,

that does not mean the generator is flawless on a given system running a particular simulation [50]. For cryptograpic application developers, the preferred reference suite is NIST battery test [51]. A subset of the results from these battery tests is provided in Section 4 of the cuRAND Programming Guide [32].

5.2 Random Number Generator Implications

As mentioned in Section 3.3.2, NVIDIA's cuRAND library facilities the simple and efficient generation of high-quality pseudorandom and quasirandom numbers. While a pseudorandom sequence is generated by a deterministic algorithm, it satisfies most of the statistical properties of a truly random sequence and is well suited for this research. Although cuRAND comes with five PRNGs, MT19937 will not be used. In this section, the remaining four PRNGs will be tested to determine how they affect the timing of the resampling step and the quality of the root mean squared error (RMSE).

MT19937 is 32-bit version for Mersenne Twister PRNG based on a period length of the Mersenne prime 2^{19937}-1. It was proposed for generating uniform pseudorandom numbers and was developed in 1997 by Makoto Matsumoto and Takuji Nishimura [52]. In cuRAND it is only available through the host API. Although some host API calls are highly optimized, they required expensive stores to global memory when the call finishes its task and again by kernels loading the data from global memory into on-chip registers. This process causes significant negative impact to performance and is therefore not used.

The MTGP32 generator provided in cuRAND is an adaptation of the Mersenne Twister optimized for GPUs developed by Saito and Matsumoto [53]. It requires input parameters that allow many threads to compute the recursion in parallel. cuRAND provides a 200 parameter set that has been pre-generated for the 32-bit generator with period 2^{11214}. It allows thread-safe generation and state update for up to 256 concurrent threads (within a single block) for each of the 200 sequences [32]. Because neither the naive nor improved parallel implementations exceed 256 threads per block this will not be a limitation. However, by using these pre-generated states, the maximum number of blocks in a grid can be 200. MTGP32's performance is significantly impacted by the host API calls `curandMakeMTGP32Constants` and `curandMakeMTGP32KernelState`. These host functions help set up parameters for different sequences in global memory and set up the initial state. The function `curandMakeMTGP32Constants` takes the pre-generated parameter set and reorganizes the format before copying it to global memory. The next function, `curandMakeMTGP32KernelState`, initials the required number of states and copies them to global memory. Neither function has alternative stream functionality, therefore forcing the calls to run serially in the default stream. In order to ensure that the resampling kernels do not try to use the PRNG data before the memory is finished copying, `cudaDeviceSynchronize` must be called to block all kernel calls until copying is finished.

The last three PRNGs, XORWOW, MRG32k3a, and Philox_4x32_10, are relatively simple to implement because they do not require any host API calls and can be called directly from a kernel. XORWOW is a member of the xor-shift fam-

ily, a subset of linear-feedback shift registers (LFSR), and was discovered by George Marsaglia [54]. It is one of the fastest non-cryptographically-secure random number generators and works by generating the next number in series by taking the *exclusive or* of the number, hence the name. MRG32k3a is a 32-bit combined multiple recursive generator with two components of order 3. It has good multidimensional uniformity, or a long period, and performs well in statistical tests up to at least 45 dimensions [55]. Philox_4x32_10 is a non-cryptographic bijection that uses multiplication instructions that compute the high and low halves of the product of word-sized operands and, as the name suggests, utilizes four 32-bit unsigned int values [56].

Table 5.2: RMSE comparison between PRNGs at 2^{16} particles.

PRNGs	RMSE for States			
	x_1	x_2	x_3	x_4
XORWOW	0.2058	0.1768	0.1661	0.1539
MRG32k3a	0.2062	0.1771	0.1662	0.1540
Philox_4x32_10	0.2059	0.1768	0.1661	0.1539
MTGP32	0.2061	0.1770	0.1662	0.1540

Table 5.2 shows the quality results for 100 MCs over 2500 samples for the improved parallel systematic resampling utilizing the multiple seed, single sequence id technique. It is clear that the results for each PRNG are nearly identical. Although their qualities are similar, the timing and implementations for the PRNGs vary greatly. Table 5.3 will be used to better understand these differences. It is split into two subtables, one consisting of statistics when calculating the starting state of each time step and the other when the starting state is only initialized once during ap-

plication startup. For calculating repeatedly, the entire resampling process, including state initialization and random number generation, is encompassed in the **Timing** column. When initializing at application startup, the amount of time required for initialization is in the column titled **Setup** and the time it takes to execute the resampling process, with only random number generation, is in the **Timing** column. Also provided is the number of registers required for the kernel incrementing up the while-loop and the kernel decrementing down the while loop, in columns **Up** and **Down**, respectively.

XORWOW and Philox_4x32_10 provide the fastest execution times for both systematic and stratified resampling algorithms and have similar implementation techniques. Both PRNGs are lightweight enough that state initialization and random number generation can be performed inside the kernels each time step. Implementing MRG32k3a in this fashion causes the worst execution time of all the PRNGs because the state initialization and random number generation of MRG32k3a are much slower than the previous two techniques. This is due to the amount of double precision registers required during state initialization. They consume all available registers in a block and then require additional local memory as described in Section 3.2.1.2, severely impacting performance. This issue is addressed in the cuRAND Programming Guide [32] under Section 3.5, Device API - Performance Notes. NVIDIA suggests saving and restoring the random generator state instead of recalculating the starting state repeatedly. By separating the initialization and random number generator, kernels using MRG32k3a were able to run at a speedup of 8x, although it remains the slowest overall. It is important to note that applying this best practice to XORWOW and

Philox_4x32_10 implementations degrades performance because loading the random generator state from global memory is slower than state initialization.

Table 5.3: Timing and registers statistics across PRNGs at 2^{16} particles.

Calculating Starting State Repeatedly								
PRNG	Systematic				Stratified			
	Setup (us)	Timing (us)	Registers		Setup (us)	Timing (us)	Registers	
			Up	Down			Up	Down
XORWOW		29	21	23		31	21	22
Philox_4x32_10		30	23	23		34	25	24
MRG32k3a		997	80	80		997	80	80
MTGP32		82	20	22		459	26	26

Calculating Starting State Once								
PRNG	Systematic				Stratified			
	Setup (us)	Timing (us)	Registers		Setup (us)	Timing (us)	Registers	
			Up	Down			Up	Down
XORWOW	25	87	23	24	25	88	23	24
Philox_4x32_10	54	100	37	36	54	100	37	36
MRG32k3a	600	128	30	28	600	128	30	28
MTGP32	50	32	20	22	250	68	26	26

MTGP32 has by far the most complicated implementation in order to be utilized efficiently in the parallel systematic and stratified resampling techniques. This is partly due to the required host API calls and partly due to the grid size limitation discussed earlier in the section. While the host API calls required for initialization are twice as fast as the MRG32k3a initialization, they do not come with alternative stream capability. This means they must be executed in the default stream. Because

the random generator states are inputs to both the parallel implementation kernels, which are not in the default stream, the only way to ensure the states are not used before they are created is to run `cudaDeviceSynchronize()` after initialization. Unfortunately, this API call blocks all future API calls and kernel launches until the device has completed all preceding requested tasks and requires more overhead than other synchronization calls. A better approach, similar to MRG32k3a's, is to perform initialization once at application startup. Next, as mentioned earlier, using the pre-generated parameters provided by NVIDIA, MTGP32 can only utilize 200 blocks in a grid. This inhibits the improved parallel implementation from launching a large number of blocks to hide its thread execution inefficiency as described in Section 4.2.2. In the kernel, device API call `curand_mtgp32_specific` is being used to generate uniform random numbers. Its constraints are listed below:

1. At most 256 threads may call this function for a given state.

2. Within a block, for a given state, if n threads are calling the function, the indices must run from 0...n-1. The indices do not have to match the thread numbers, and may be distributed among the threads as required by the calling program. At a given point in the code, all of the indices from 0...n-1, or none of them, must be used.

3. A given state may not be used by more than one block.

4. A given block may generate randoms using multiple states.

Item three from the list above further limits the parallel implementation because both the **Up** and **Down** kernels are running in different streams concurrently and it is possible that a given block from each kernel could, in theory, access the same state. To ensure this does not occur, each kernel must run with only 100 blocks and in one kernel the input parameter to `curand_mtgp32_specific` must be offset by 100. Finally, notice in Table 5.3 that when using MTGP32, the timing for systematic and stratified resampling differ. Due to second item in the above list, systematic resampling cannot be implemented in the conventional manner. For the systematic implementation, prior to launching the resampling kernels, a kernel must be launched with one thread and one block to produce a single uniform random number. This number is then stored in global memory and loaded by the following kernels. In turn, systematic is faster than stratified when utilizing MTGP32. This speedup is more pronounced when the starting state is calculated repeatedly because of the additional number of states being initialized and copied.

Section 3.5 of the cuRAND Programming Guide also suggests that one way to speed the random generator is to use different seeds for each thread and a constant sequence number of 0. This was used for the XORWOW, Philox_4x32_10, and MRG32k3a implementations in this research. It is known that this method provides less guarantees about the mathematical properties of the generated sequences and a possible side effect is that some threads may have highly correlated outputs. It is noted in the cuRAND Programming Guide that this is rare and this research did not see any adverse effects with this implementation technique [32]. Because the quanti-

ties of particles in this research can be considered low in respect to the sequence sizes of the PRNG, quality of the results is not affected by using this enhancement.

CHAPTER 6

TESTING AND ANALYSIS

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

—Brian Kernighan, Princeton

6.1 Experimentation Setup

To examine the performance of the parallel approach introduced in this paper, it is compared to alternatives using a general nonlinear, non-Gaussian four state-space model from research provided by Schön [7] and can be found under Software and Data sets - Rao-Blackwelled particle filter at `http://user.it.uu.se/~thosc112/research/`. For simplicity, a BPF was used to run 100 Monte Carlos of 2500 samples with 2^{16} and 2^{20} particles, with resampling performed every time step. Particle set sizes do not exceed 2^{20} to eliminate possible numerical instabilities produced during the prefix-sum, as pointed out by Murray [28] while using single-precision on GPUs. Double-precision on consumer-level cards and embedded systems is significantly slower.

These tests were performed using a desktop computer running Kubuntu 18.04 with an NVIDIA GTX 1080 and an Intel i7-5960X. Specifications are provided in Table 4.2 and Table 6.1, respectively. Serial versions of the systematic and stratified are run on the CPU with optimization level -O3. Code was written and tested in C++ GNU 7.3.0 and CUDA 10.0. The *--use_fast_math* flag was set at compile time to improve the use of any special functions by forcing the use of intrinsics. Intrinsic functions are faster as they map to fewer native instructions. A XORWOW generator was used from cuRAND [32] to generate all random numbers on the device. To ensure both while-loops of the parallelized systematic and stratified methods had the same random numbers, a seed was generated on the CPU and sent to the GPU as a parameter to each kernel [44].

Table 6.1: Intel i7-5960X Hardware Specifications.

Property	Value
Architecture	Haswell
Clock Rate	3.0 GHz
Overclock Rate	4.4 GHz
Cores	8
Threads	16
Cache	20 MB
Lithography	22nm

The prefix-sum at line (1) in Algorithm 12 can be implemented in parallel using the CUB library [57]. It is an NVIDIA library containing collective kernel-level primitives designed around reusable software components such as warps and blocks for the SIMT paradigm. This library not only provides the benefit of simplified

coding, but is optimized by NVIDIA to use the latest hardware features and provides sustainability when porting this method to different NVIDIA GPU architectures.

To assess and compare the quality of the serial and parallel resampling algorithms, the RMSE [58] is used in the following form

$$RMSE_{fo} = \sqrt{\sum_{i=1}^{M}\sum_{n=1}^{S}(f_n^i - o_n^i)^2/(M * S)} \tag{6.1}$$

where M is the number of Monte Carlo runs, S is the number of samples in each run, f is the expected results, or truth data, and o is the observed filter estimates.

Due to the stochastic nature of the resampling process, execution time is determined by averaging the execution time of all the Monte Carlos. Also, the first executed Monte Carlo was considered a warm-up run and the timing of that run was discarded. This warm-up run ensures the GPU is in the correct performance state, in this case the highest performance.

6.2 Simulation Results

Table 6.2 shows timing and RMSE for all systematic and stratified implementations. The naive GPU (N-GPU) implementations of Algorithm 12 is 8x and 16.42x faster than the serial CPU versions of systematic and stratified, respectively. The improved GPU (I-GPU) implementations are 16.89x and 33.43x faster. Stratified execution times are similar to the systematic on the GPU because the random number generation can be performed on each thread in parallel. All three produced nearly

identical RMSE. This is expected because the GPU implementations produce the exact resampling index as serial methods. Numerical differences can be attributed to rounding errors in CPU and GPU arithmetic. Timing results include the prefix-sum computation, random number generation, and resampling index search.

Table 6.2: RMSE comparison of Systematic/Stratified implementation at 2^{16} particles.

Type	Speed (μs)	RMSE for States			
		x_1	x_2	x_3	x_4
Systematic (CPU)	456	0.2061	0.1770	0.1662	0.1540
Stratified (CPU)	936	0.2061	0.1770	0.1662	0.1540
Systematic (N-GPU)	57	0.2060	0.1769	0.1661	0.1540
Stratified (N-GPU)	57	0.2060	0.1769	0.1661	0.1540
Systematic (I-GPU)	27	0.2060	0.1769	0.1661	0.1540
Stratified (I-GPU)	28	0.2060	0.1769	0.1661	0.1540

The additional speedup of the improved implementations over the naive is directly correlated to the decreased number of global memory loads. Load quantities for 2^{20} particles for over 1000 samples are presented in Table 6.3 and can be found using the NVIDIA Compute mentioned in Section 3.3.5. The improved implementation reduces the number of global memory loads over 10x by utilizing shared memory.

Table 6.3: 32-bit global memory loads of naive and improved.

2^{20} Particles	Naive	Improved
while_loop_increment	154,365,478	14,996,374
while_loop_decrement	152,593,046	13,611,845

As mentioned earlier, the historic bottleneck of particle filtering lies within the resampling step. This is evident in Figure 6.1, which shows that when running the serial systematic resampling on the CPU it consumes roughly 93% of workload performed in a time step. While the naive parallel implementation is able to shift this workload to 70%, it is the improved parallel implementation, with a workload of 32%, that transfers the majority of the computation performed from resampling to the prediction and update steps of particle filtering.

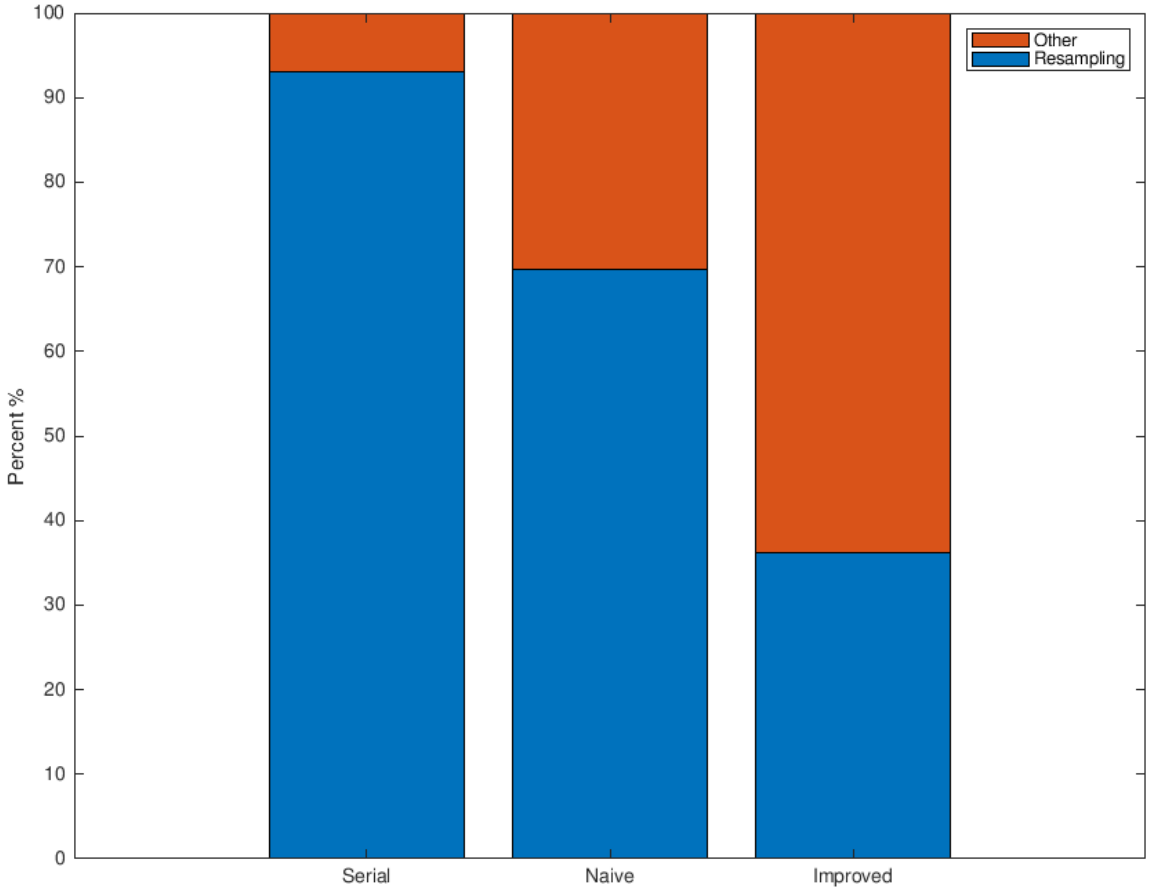


Figure 6.1: Workload percentages of systematic resampling implementations.

Table 6.4 shows speed and RMSE comparison between Metropolis, C1, C2; Rejection; and the parallelized systematic and stratified methods for a particle set of

2^{20}. First, when comparing to Table 6.2, it can be seen that the performance differences between the naive and improved parallel implementation are more pronouced as the number of particles increase. Table 6.4 shows that the improved parallelized implementations of the systematic and stratified methods provide at least a 13.13x speed over Metropolis, and as B increases, that speed up becomes more pronounced. As expected, systematic and stratified techniques provide the best quality, with Metropolis in second. Although C1 provides the fastest execution time of all Metropolis implementations, it has the worst quality because it focuses on local selections, and the expected number of repetitions of the particles becomes different than that in Metropolis [30]. C2 provides a balance between speed and quality; speed because it is coalesced and quality because it is more similar to Metropolis. The parallelized systematic and stratified methods are slightly faster than C2 with $B \geq 32$, but nearly double the execution time with $B=16$. With C2 producing quality only slightly worse for this particular test problem, it seems to be the best trade-off between speed and quality. It is evident that while rejection resampling is faster than Metropolis and C2, at $B \geq 64$, it has the worst quality out of all the methods for this data set.

Variance of the particle weight array will play a significant roll in the performance of these resampling methods. In Metropolis resampling, for a given B, as the variance increases, the execution time will remain constant while the quality will worsen. Conversely, execution time of the systematic and stratified parallel implementations will most certainly increase as threads are required to traverse further through the particle weight array to satisfy the while-loop condition.

Table 6.4: RMSE comparison of resampling techniques at 2^{20} particles.

Type	Cutoff (B)	Speed (μs)	RMSE for States			
			x_1	x_2	x_3	x_4
Systematic (N)		1511	0.2060	0.1769	0.1661	0.1539
Stratified (N)		1516	0.2060	0.1769	0.1661	0.1539
Systematic (I)		375	0.2059	0.1769	0.1661	0.1539
Stratified (I)		390	0.2059	0.1769	0.1661	0.1539
Metropolis	16	5122	0.2062	0.1772	0.1663	0.1540
	32	10158	0.2061	0.1771	0.1662	0.1540
	64	20226	0.2059	0.1769	0.1662	0.1540
Rejection		631	0.3352	0.2613	0.2123	0.1807
Metropolis (C1)	16	135	0.2192	0.1865	0.1725	0.1551
	32	259	0.2207	0.1871	0.1728	0.1551
	64	518	0.2216	0.1874	0.1730	0.1552
Metropolis (C2)	16	245	0.2062	0.1771	0.1663	0.1540
	32	452	0.2061	0.1771	0.1662	0.1540
	64	881	0.2061	0.1770	0.1662	0.1540

Figure 6.2 compares execution times of all methods over a sweep of particle set sizes from 2^{10} to 2^{20}. For small particle sets, the CPU will perform better than all GPU methods. Execution times on a GPU will plateau for small particle sets because overhead is a larger contributor than computational workload. It can be seen that C1 and C2 are faster alternatives to the improved systematic and stratified parallel versions when B is small, and might be an appealing choice if the impact to quality is not severe.

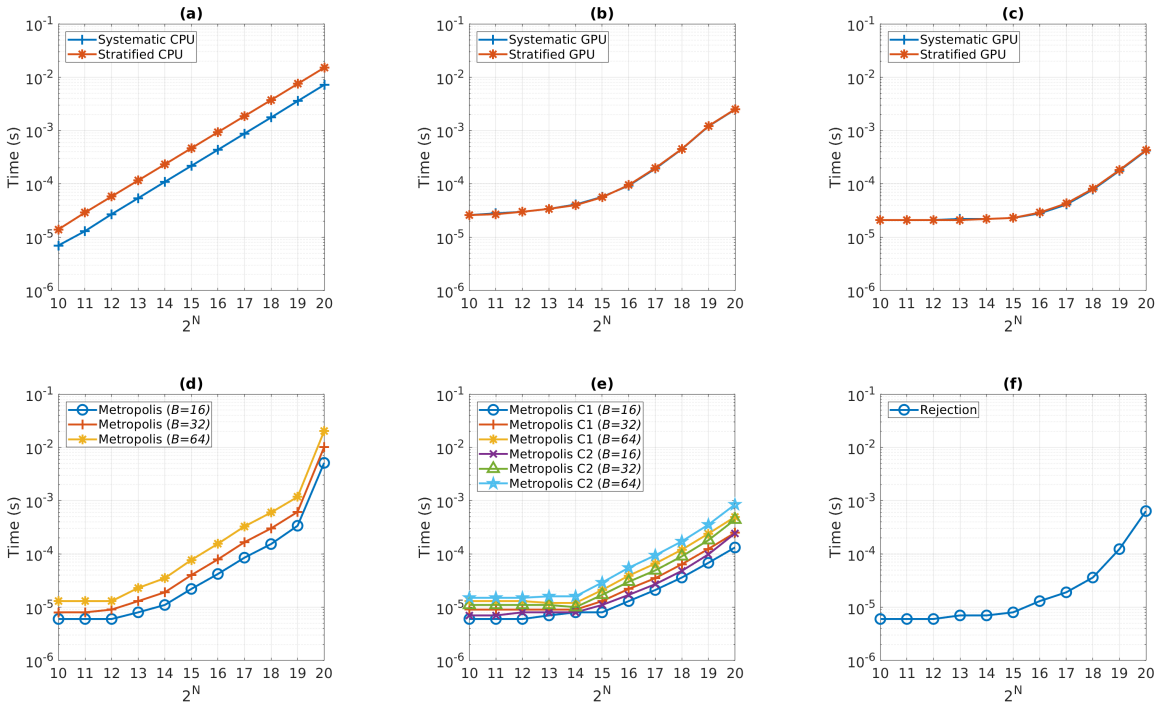


Figure 6.2: Execution times of the resampling algorithms in single-precision floating point arithmetic versus increasing number of particles (2^N): (**a**) serial systematic and stratified on CPU with -O3, (**b**) naive parallel Systematic and Stratified on GPU, (**c**) improved parallel systematic and stratified on GPU, (**d**) Metropolis on GPU, (**e**) Metropolis: C1 and C2 on GPU, (**f**) Rejection resampling on GPU.

6.3 Timing Across Multiple Variances

The previous section calculated timing performance based off a given benchmark problem. This section will display timing results of the parallel implementation across a sweep of variances in the particle weights. One of the characteristics of the parallel implementation is that as variance in particle weights increases so does execution time. Recall from Section 4.2.6 that the worst case scenario is when variance is at is maximum. Although possible, the purpose of resampling is to prevent that from happening and maintain the lowest variance possible. To get a better understanding

of how the parallel implementations should perform in more practical scenarios they are tested against the framework provided by Murray in [28].

Weights are simulated for N number of particles using the following equation:

$$w^i = \frac{1}{\sqrt{2\pi}} \exp(-\frac{1}{2}(x^i - y^i)^2) \tag{6.2}$$

where y is the observation, x^i is a set of weights generated with $\mathcal{N}(0, 1)$ and $i = 1, \ldots, N$. As y increases, $0, \frac{1}{2}, 1, 1\frac{1}{2}, \ldots, 4$; so does the relative variance. Table 6.5 and Table 6.6 show the timing and quantity of ℓ results for 2^{16} number of particles for naive and improved systematic resampling, respectively. The results are an average across 10,000 Monte Carlos. Notice that there is a direct relationship between the maximum number of ℓ to the maximum time taken to perform resampling. Also, these tables align with our work efficiency equations; as the variance increases, the work efficiency for both decreases, more so for the improved. More tables can be found in Section B.1 and Section B.2 for particle numbers ranging from 2^{10} to 2^{20} by a power of two.

Table 6.5: Naive: Timing/Quantity(ℓ) at 2^{16} Particles - Multiple Variances

Naive Method: Particles (65536)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	29	63	37	1	205	15	15	6,943,641	1	238	15	15	8,082,892
0.5	29	67	39	0	246	17	17	7,303,210	0	240	18	17	8,414,897
1.0	31	89	46	0	313	24	23	10,113,603	1	359	24	23	11,833,766
1.5	35	103	53	0	484	33	32	14,231,042	0	405	32	31	13,953,656
2.0	37	122	63	2	568	44	43	18,821,345	0	571	45	43	19,466,809
2.5	44	150	77	0	802	59	58	25,833,590	3	851	60	58	26,540,884
3.0	52	186	96	2	1,004	80	78	30,832,213	2	1,016	81	79	33,749,941
3.5	62	238	123	3	1,341	110	107	42,457,655	2	1,458	113	110	41,926,586
4.0	78	314	163	1	1,846	158	153	57,355,685	1	1,906	155	152	58,468,995

Table 6.6: Improved: Timing/Quantity(ℓ) at 2^{16} Particles - Multiple Variances

Naive Method: Particles (65536)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	28	46	31	0	226	15	15	8,150,984	0	198	15	15	8,050,415
0.5	29	45	31	0	259	17	17	8,150,281	0	269	18	17	8,304,139
1.0	29	52	32	0	318	24	23	10,706,964	1	327	24	23	11,979,178
1.5	29	53	33	0	457	33	32	14,345,207	2	419	33	32	14,776,303
2.0	29	56	34	0	580	45	43	21,142,322	0	588	45	43	22,306,243
2.5	30	58	36	1	808	60	58	23,713,926	3	854	60	58	31,948,670
3.0	31	68	39	1	1,130	81	79	34,374,626	3	1,204	80	78	40,136,019
3.5	32	78	42	1	1,612	113	110	51,111,637	0	1,548	113	109	50,980,780
4.0	34	91	48	4	2,146	158	154	66,755,247	2	2,145	158	154	71,576,139

6.4 Timing of Worst Case Scenario

Algorithm efficiency for work is described in Section 4.2.6. While that provides efficiency as $N \to \infty$ on a PRAM, this section will dive into the work efficiency of the Up and Down kernels on a machine from an implementation point-of-view. In

order to calculate a more accurate worst case scenario, hardware must be taken into account. First, this work efficiency takes into account the number of GPU cores that are used during computation. Second, it considers one cycle as one trip through the entire while-loop and will calculate the total number of cycles across all threads, where there is one thread per elements. Work efficiency does not depend on whether the implementation used monolithic or grid-stride looping kernels. Equation 6.3 and Equation 6.4 represent work efficiency for the naive and improved implementation, respectively.

$$WE_{np_w} = \frac{N}{\left[\left[\left(\frac{(N-1)*N}{2}\right) + N\right] + (N-1)\right] \Big/ \text{cores}} \tag{6.3}$$

$$WE_{ip_w} = \frac{N}{\left[\frac{(N-1)*N}{2} + N + (N - (WS*2))*F + (WS*2) - 1\right] \Big/ \text{cores}} \tag{6.4}$$

Table 6.7 and Table 6.8 provides the tabular form of Equation 6.3 and Equation 6.4 as the number of particles grow from 2^{10} to 2^{20} by a power of two. It shows the execution time on a GTX 1080, with 2560 CUDA cores, along with the maximum number of cycles, ℓ, for a given thread and the total number of cycles for all threads for each kernel. It is clear that the Down kernel provides little impact to the work efficiency because it executes the minimum number of cycles allowed. While the Down kernel in the improved implementation required more cycles over the naive because of the hard-coded for-loop, overall the improved is between 5-21x faster due to more efficient accesses to global memory.

Table 6.7: Naive: Work Efficiency of Worst Case Scenario: Systematic

Naive: Work Efficiency of Worst Case Scenario						
Particle #	Time μs	Kernel A		Kernel B		Work Efficiency
		Max	Total	Max	Total	
1,024	166	1,024	524,800	1	1,023	7.607×10^{-7}
2,048	300	2,048	2,098,176	1	2,047	3.809×10^{-7}
4,096	581	4,096	8,390,656	1	4,095	1.906×10^{-7}
8,192	1,147	8,192	33,558,528	1	8,191	9.533×10^{-8}
16,384	2,302	16,384	134,225,920	1	16,383	4.768×10^{-8}
32,768	4,700	32,768	536,887,296	1	32,767	2.384×10^{-8}
65,536	11,673	65,536	2,147,516,416	1	65,535	1.192×10^{-8}
13,1072	38,572	131,072	8,590,000,128	1	131,071	5.960×10^{-9}
262,144	149,927	262,144	34,359,869,440	1	262,143	2.980×10^{-9}
524,288	602,399	524,288	137,439,215,616	1	524,287	1.490×10^{-9}
1,048,576	2,533,634	1,048,576	549,756,338,176	1	1,048,575	7.451×10^{-10}

Table 6.8: Improved: Work Efficiency of Worst Case Scenario: Systematic

Improved: Work Efficiency of Worst Case Scenario						
Particle #	Time μs	Kernel A		Kernel B		Work Efficiency
		Max	Total	Max	Total	
1,024	33	1,024	524,800	32	30,783	7.200×10^{-7}
2,048	38	2,048	2,098,176	32	63,551	3.701×10^{-7}
4,096	49	4,096	8,390,656	32	129,087	1.878×10^{-7}
8,192	72	8,192	33,558,528	32	260,159	9.462×10^{-8}
16,384	121	16,384	134,225,920	32	522,303	4.750×10^{-8}
32,768	229	32,768	536,887,296	32	1,046,591	2.380×10^{-8}
65,536	641	65,536	2,147,516,416	32	2,095,167	1.191×10^{-8}
131,072	1,903	131,072	8,590,000,128	32	4,192,319	5.958×10^{-9}
262,144	7,132	262,144	34,359,869,440	32	8,386,623	2.980×10^{-9}
524,288	29,801	524,288	137,439,215,616	32	16,775,231	1.490×10^{-9}
1,048,576	131,764	1,048,576	549,756,338,176	32	33,552,447	7.450×10^{-10}

CHAPTER 7

CONCLUSION

7.1 Summary

Particle filters are known to provide much better accuracy in nonlinear, non-Gaussian problems, such as state estimation. Unfortunately, they are not nearly as common in practice as techniques like the Kalman filter and the EKF due to their high computational nature. This pitfall is caused by two underlying characteristics. First, as the number of states increase, the number of particles should increase exponentially. Second, and the primary focus of this research, is that traditional resampling techniques are considered serial by nature causing a performance bottleneck. GPUs provide a compute rich environment for mathematical problems that are embarrassingly parallel. These problems usually fit within the programming paradigm of SIMD and can take advantage of the high quantities and relatively simple cores provided by GPUs. Two out of the three steps in particle filtering have been implemented on GPU hardware in recent history. By finding a way to implement the last step, resampling, particle filtering becomes a viable alternative to nonlinear, non-Gaussian problems.

This dissertation shows that while some traditional resampling methods seem inherently serial, calculation can be distributed to parallel environments, such as GPUs, given the right conditions. In the case of systematic and stratified resampling, all inputs are known when the resampling index is determined and has indices that are zero-based, consecutive, strictly monotonically increasing. Therefore, the while condition can be implemented to all threads independently. While this parallel programming technique is shown to have poor work efficiency, it can provide improvements in speedups over traditional serial CPU implementation and produce an identical resampling index.

This research provides two ways to implement a parallel approach to systematic and stratified resampling. While the naive method is roughly 8x and 16x faster than the serial implementation, respectively, it has limitations that keep it from achieving optimal performance. An improved method is proposed that efficiently utilizes memory transitions to minimize global memory accesses. Improvements to thread execution efficiency, memory dependency stalls, and sub-optimal occupancy are then provided to decrease execution times of the systematic and stratified by 16.89x and 33.43x, respectively, over the serial method. With this parallel approach, all steps of the particle filter can be implemented in a parallel fashion. This shifts the historic workload particle filtering from resampling to the prediction and update steps.

For a thorough comparison, the performance was compared to a popular GPU method known as Metropolis resampling. Three versions of the Metropolis are presented; the original version and two coalesced versions that performed more efficient

memory reads to global memory. The parallel systematic and stratified methods are significantly faster then Metropolis on large particle sets. However, they are slightly slower than the coalesced versions of Metropolis when B is small. Therefore, the specific method that is chosen will be a trade-off between performance and accuracy.

Given that particle filtering is a Monte Carlo technique it requires the generation of random numbers. How these numbers are generated can significantly affect accuracy and speed. A list of requirements was given to ensure that random number generators implemented in the GPU environment provide good statistical quality and are repeatable between runs and on different systems. NVIDIA's cuRAND library provides a handful of PRNGs that can be implemented to guarantee statistical quality or forfeit that guarantee for improved speed. It was shown that the GPU parallel implementations of systematic and stratified resampling are not sensitive to random numbers generated with less guarantee that statistical correlation are non-existent.

7.2 Future Work

7.2.1 Multi-core Implementation

There is no reason to assume parallel techniques presented in this dissertation can not be extended to multi-core CPU environments. In that scenario, multiple CPU cores can be used to retrieve data from memory in chunks and put into large caches provided by the CPU architectures. Utilizing these large caches may eliminate the need for an improved implementation and will not be limited by warp size. This

would approach will provide performance improvements for applications that don't have access to GPU hardware.

7.2.2 Custom Prefix-Sum Kernel

It can be shown with profiling tools that there are gaps between the prefix-sum kernels, launched under the CUB library, and the novel parallel implementation presented in this work. This is due to kernel launch overhead, and on a GTX1080, when processing 2^{20}, there is not enough work to hide this latency. It might be possible to write a custom prefix-sum kernel that can also combine random number generation and the resampling index search. In doing so, a Kahan summation algorithm [59], also known as compensated summation, could be used to reduce the numerical error in the prefix-sum on single precision numbers.

7.2.3 Particle Filter Implementation

The parallel implementation provided in this paper can be integrated into particle filters that currently utilize the GPU for the updating and prediction steps. The combination of all three step on the GPU can provide online calculations while using unbiased resampling. These three steps in the particle filter workflow are repeatedly executed through the run-time of most applications. CUDA streams require that the work is resubmitted with every iteration, which consumes both time and CPU resources. CUDA Graphs is a new model for submitting work using CUDA, which was released with CUDA 10.0 [32]. A graph consists of a series of operations, such as memory copies and kernel launches, connected by dependencies and defined sepa-

rately from its execution. Graphs enable a define-once-run-repeatedly execution flow. By utilizing Graphs, the overhead of launching multiple kernels can be significantly reduced.

APPENDICES

APPENDIX A

A.1 Systematic/Stratified (Naive): Up Kernel

```
template<typename T>
__global__ void __launch_bounds__(kTRI) computeResampleIndexSysUp( int const n,
    unsigned long long const seed, int * __restrict__ devResampleIndexUp, T const *
    __restrict__ devCumulativeSum ) {

#if RESAMPLE == 0 // Systematic
  curandStateXORWOW_t localState { };
  curand_init( seed, 0, 0, &localState );
  T const distro = curand_uniform( &localState );
#endif

  for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x
      * gridDim.x ) {

    auto const tidf { static_cast<T>( tid ) };
    auto mask { true };
    auto l { 0 };
    auto idx { 0 };

#if RESAMPLE == 1 // Stratified
    curandStateXORWOW_t localState { };
    curand_init ( seed + tid, 0, 0, &localState );
    T const distro = curand_uniform(&localState);
#endif

    // Distribution will be the same for each Monte Carlo
    T const draw = ( distro + tidf ) / n;
    while ( mask && ( tid < n - 1 ) ) {
      // The last thread will always be false because devCumulativeSum(end) = 1.0f
      mask = devCumulativeSum[tid + 1] < draw;
      if ( mask )
        idx++;
      l++;
    }
    devResampleIndexUp[tid] = idx;
  }
}
```

A.2 Systematic/Stratified (Naive): Down Kernel

```
1  template<typename T>
2  __global__ void __launch_bounds__(kTRI) computeResampleIndexSysDown( int const n,
       unsigned long long const seed, int * __restrict__ devResampleIndexDown, T const *
        __restrict__ devCumulativeSum ) {
3
4  #if RESAMPLE == 0 // Systematic
5    curandStateXORWOW_t localState { };
6    curand_init( seed, 0, 0, &localState );
7    T const distro = curand_uniform( &localState );
8  #endif
9
10   for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x
         * gridDim.x ) {
11
12     auto const tidf { static_cast<T>( tid ) };
13     auto mask { false };
14     auto l { 1 };
15     auto idx { 0 };
16
17 #if RESAMPLE == 1 // Stratified
18     curandStateXORWOW_t localState { };
19     curand_init ( seed + tid, 0, 0, &localState );
20     T const distro = curand_uniform(&localState);
21 #endif
22
23     // Distribution will be the same for each Monte Carlo
24     T const draw = ( distro + tidf ) / n;
25
26     while ( !mask && ( tid >= l ) ) {
27       mask = devCumulativeSum[tid - l] < draw;
28       if ( !mask )
29         idx--;
30       l++;
31     }
32     devResampleIndexDown[tid] = idx;
33   }
34 }
```

A.3 Systematic/Stratified (Shared Memory): Up Kernel

```
template<typename T>
__global__ void __launch_bounds__(kTRI) computeResampleIndexSysUpShared( int const n,
     unsigned long long const seed, int * __restrict__ devResampleIndexUp, T const *
    __restrict__ devCumulativeSum ) {

  auto const tile32 = cg::tiled_partition<kWarpSize>( cg::this_thread_block( ) );
  auto const t = threadIdx.x;

  __shared__ T sData[kTRI * 2]; // Will hold 64 elements

#if RESAMPLE == 0 // Systematic
  curandStateXORWOW_t localState { };
  curand_init( seed, 0, 0, &localState );
  T const distro = curand_uniform( &localState );
#endif

  for ( int tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x *
       gridDim.x ) {

    auto const tidf { static_cast<T>( tid ) };
    auto mask { true };
    auto l { 0 };
    auto idx { 0 };

#if RESAMPLE == 1 // Stratified
    curandStateXORWOW_t localState { };
    curand_init ( seed + tid, 0, 0, &localState );
    T const distro = curand_uniform(&localState);
#endif

    // Distribution will be the same for each Monte Carlo
    T const draw = ( distro + tidf ) / n;

    while ( tile32.any( mask ) ) {

      if ( tid < n - kTRI - l ) {
        sData[t] = devCumulativeSum[tid + l];
        sData[t + kTRI] = devCumulativeSum[tid + kTRI + l];
        tile32.sync( );

        for ( auto i = 0; i < kTRI; i++ ) {
          mask = sData[t + i] < draw;
          if ( mask )
            idx++;
        }
        l += kTRI;
      } else {
        while ( mask && tid < ( n - l ) ) {
          // Last thread will always be false because devCumulativeSum(end) = 1.0f
          mask = devCumulativeSum[tid + l] < draw;
          if ( mask )
            idx++;
          l++;
        }
      }
      tile32.sync( );
    }
    devResampleIndexUp[tid] = idx;
  }
}
```

A.4 Systematic/Stratified (Shared Memory): Down Kernel

```
template<typename T>
__global__ void __launch_bounds__(kTRI) computeResampleIndexSysDownShared( int const
    n, unsigned long long const seed, int * __restrict__ devResampleIndexDown, T
    const * __restrict__ devCumulativeSum ) {

  auto const tile32 = cg::tiled_partition<kWarpSize>( cg::this_thread_block( ) );
  auto const t = threadIdx.x;

  __shared__ T sData[kTRI * 2];

#if RESAMPLE == 0 // Systematic
  curandStateXORWOW_t localState { };
  curand_init( seed, 0, 0, &localState );
  T const distro = curand_uniform( &localState );
#endif

  for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x
      * gridDim.x ) {

    auto const tidf { static_cast<T>( tid ) };
    auto mask { false };
    auto l { 0 };
    auto idx { 0 };

#if RESAMPLE == 1 // Stratified
    curandStateXORWOW_t localState { };
    curand_init ( seed + tid, 0, 0, &localState );
    T const distro = curand_uniform(&localState);
#endif

    // Distribution will be the same for each Monte Carlo
    T const draw = ( distro + tidf ) / n;

    while ( !tile32.all( mask ) ) {

      if ( tid >= kTRI + l ) {
        sData[t] = devCumulativeSum[tid - kTRI - l];
        sData[t + kTRI] = devCumulativeSum[tid - l];
        tile32.sync( );

        for ( auto i = 1; i < kTRI + 1; i++ ) {
          mask = sData[t + kTRI - i] < draw;
          if ( !mask )
            idx--;
        }
        l += kTRI;
      } else {
        while ( !mask ) {
          if ( tid > l ) mask = devCumulativeSum[tid - ( l + 1 )] < draw;
          else mask = true;
          if ( !mask )
            idx--;
          l++;
        }
      }
      tile32.sync( );
    }
    devResampleIndexDown[tid] = idx;
  }
}
```

A.5 Systematic/Stratified (SM/Prefetch) Up Kernel

```
template<typename T>
__global__ void __launch_bounds__(kTRI) computeResampleIndexSysUpShared( int const n,
     unsigned long long const seed, int * __restrict__ devResampleIndexUp, T const *
    __restrict__ devCumulativeSum ) {

  auto const tile32 = cg::tiled_partition<kWarpSize>( cg::this_thread_block( ) );
  auto const t = threadIdx.x;

  __shared__ T sData[kTRI * 2]; // Will hold 64 elements

#if RESAMPLE == 0 // Systematic
  curandStateXORWOW_t localState { };
  curand_init( seed, 0, 0, &localState );
  T const distro = curand_uniform( &localState );
#endif

  for ( int tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x *
       gridDim.x ) {

    auto const tidf { static_cast<T>( tid ) };
    auto mask { true };
    auto l { 0 };
    auto idx { 0 };

#if RESAMPLE == 1 // Stratified
    curandStateXORWOW_t localState { };
    curand_init ( seed + tid, 0, 0, &localState );
    T const distro = curand_uniform(&localState);
#endif

    // Distribution will be the same for each Monte Carlo
    T const draw = ( distro + tidf ) / n;

    while ( tile32.any( mask ) ) {

      if ( tid < n - kTRI - l ) {
        sData[t] = devCumulativeSum[tid + l];
        sData[t + kTRI] = devCumulativeSum[tid + kTRI + l];
        tile32.sync( );

        for ( auto i = 0; i < kTRI; i++ ) {
          mask = sData[t + i] < draw;
          if ( mask )
            idx++;
        }
        l += kTRI;
      } else {
        while ( mask && tid < ( n - l ) ) {
          // Last thread will always be false because devCumulativeSum(end) = 1.0f
          mask = devCumulativeSum[tid + l] < draw;
          if ( mask )
            idx++;
          l++;
        }
      }
      tile32.sync( );
    }
    devResampleIndexUp[tid] = idx;
  }
}
```

A.6 Systematic/Stratified (SM/Prefetch): Down Kernel

```
template<typename T>
__global__ void __launch_bounds__(kTRI) computeResampleIndexSysDownShared( int const
    n, unsigned long long const seed, int * __restrict__ devResampleIndexDown, T
    const * __restrict__ devCumulativeSum ) {

  auto const tile32 = cg::tiled_partition<kWarpSize>( cg::this_thread_block( ) );
  auto const t = threadIdx.x;

  __shared__ T sData[kTRI * 2];

#if RESAMPLE == 0 // Systematic
  curandStateXORWOW_t localState { };
  curand_init( seed, 0, 0, &localState );
  T const distro = curand_uniform( &localState );
#endif

  for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x
      * gridDim.x ) {

    auto const tidf { static_cast<T>( tid ) };
    auto mask { false };
    auto l { 0 };
    auto idx { 0 };

#if RESAMPLE == 1 // Stratified
    curandStateXORWOW_t localState { };
    curand_init ( seed + tid, 0, 0, &localState );
    T const distro = curand_uniform(&localState);
#endif

    // Distribution will be the same for each Monte Carlo
    T const draw = ( distro + tidf ) / n;

    while ( !tile32.all( mask ) ) {

      if ( tid >= kTRI + l ) {
        sData[t] = devCumulativeSum[tid - kTRI - l];
        sData[t + kTRI] = devCumulativeSum[tid - l];
        tile32.sync( );

        for ( auto i = 1; i < kTRI + 1; i++ ) {
          mask = sData[t + kTRI - i] < draw;
          if ( !mask )
            idx--;
        }
        l += kTRI;
      } else {
        while ( !mask ) {
          if ( tid > l )
            mask = devCumulativeSum[tid - ( l + 1 )] < draw;
          else
            mask = true;
          if ( !mask )
            idx--;
          l++;
        }
      }
      tile32.sync( );
    }
    devResampleIndexDown[tid] = idx;
  }
}
```

A.7 Systematic/Stratified (SM/Prefetch/2 Warps) Up Kernel

```
template<typename T>
__global__ void __launch_bounds__(64) computeResampleIndexSysUpSharedPrefetch64(int const n, unsigned long long const seed, int * __restrict__ devResampleIndexUp, T * __restrict__ devCumulativeSum) {

  auto const tile32 = cg::tiled_partition < kWarpSize > ( cg::this_thread_block() );
  uint const t = tile32.thread_rank();

  __shared__ T sWarp0[kTRI * 2];
  __shared__ T sWarp1[kTRI * 2];

#if RESAMPLE == 0
  curandStateXORWOW_t localState { };
  curand_init( seed, 0, 0, &localState );
  T const distro = curand_uniform( &localState );
#endif

  for ( int tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x * gridDim.x ) {

#if RESAMPLE == 1
    curandStateXORWOW_t localState { };
    curand_init( seed + tid, 0, 0, &localState );
    T const distro = curand_uniform( &localState );
#endif

    T const tidf { static_cast<T>( tid ) };
    int l { 0 };
    int idx { 0 };
    T a { };
    T b { };

    if ( threadIdx.x < kWarpSize ) {  // If warp 0

      bool mask { true };

      if ( tid < n - kTRI - 1 ) {
        sWarp0[t] = devCumulativeSum[tid + 1];
        sWarp0[t + kTRI] = devCumulativeSum[tid + kTRI + 1];
      }

      // Distribution will be the same for each Monte Carlo
      T const draw = ( distro + tidf ) / n;

      tile32.sync();

      while ( tile32.any( mask ) ) {
        if ( tid < n - ( kTRI + 32 ) - 1 ) {

          a = devCumulativeSum[tid + 32 + 1];
          b = devCumulativeSum[tid + kTRI + 32 + 1];

          #pragma unroll kTRI
          for ( int i = 0; i < kTRI; i++ ) {
            mask = sWarp0[t + i] < draw;
            if ( mask )
              idx++;
          }
          l += kTRI;
          sWarp0[t] = a;
          sWarp0[t + kTRI] = b;
          tile32.sync();

        } else {
          while ( mask && tid < ( n - 1 ) ) {
```

```
63              mask = devCumulativeSum[tid + l] < draw; // The last thread will always
                    be false because devCumulativeSum(end) = 1.0f
64              if ( mask )
65                idx++;
66              l++;
67            }
68          }
69
70          tile32.sync();
71        }
72        devResampleIndexUp[tid] = idx;
73
74      } else {  // If warp 1
75
76        bool mask { true };
77
78        if ( tid < n - kTRI - l ) {
79          sWarp1[t] = devCumulativeSum[tid + l];
80          sWarp1[t + kTRI] = devCumulativeSum[tid + kTRI + l];
81        }
82
83        // Distribution will be the same for each Monte Carlo
84        T const draw = ( distro + tidf ) / n;
85
86        tile32.sync();
87
88        while ( tile32.any( mask ) ) {
89          if ( tid < n - ( kTRI + 32 ) - l ) {
90
91            a = devCumulativeSum[tid + 32 + l];
92            b = devCumulativeSum[tid + kTRI + 32 + l];
93
94            #pragma unroll kTRI
95            for ( int i = 0; i < kTRI; i++ ) {
96              mask = sWarp1[t + i] < draw;
97              if ( mask )
98                idx++;
99            }
100           l += kTRI;
101           sWarp1[t] = a;
102           sWarp1[t + kTRI] = b;
103           tile32.sync();
104
105         } else {
106           while ( mask && tid < ( n - l ) ) {
107             mask = devCumulativeSum[tid + l] < draw; // The last thread will always
                    be false because devCumulativeSum(end) = 1.0f
108             if ( mask )
109               idx++;
110             l++;
111           }
112         }
113
114         tile32.sync();
115       }
116       devResampleIndexUp[tid] = idx;
117     }
118   }
119 }
```

A.8 Systematic/Stratified (SM/Prefetch/2 Warps): Down Kernel

```
1  template<typename T>
2  __global__ void __launch_bounds__(64) computeResampleIndexSysDownSharedPrefetch64(int
       const n, unsigned long long const seed, int * __restrict__ devResampleIndexDown,
       T * __restrict__ devCumulativeSum) {
3
4    auto const tile32 = cg::tiled_partition < kWarpSize > ( cg::this_thread_block() );
5    uint const t = tile32.thread_rank();
6
7    __shared__ T sWarp0[kTRI * 2];
8    __shared__ T sWarp1[kTRI * 2];
9
10 #if RESAMPLE == 0
11   curandStateXORWOW_t localState { };
12   curand_init( seed, 0, 0, &localState );
13   T const distro = curand_uniform( &localState );
14 #endif
15
16   for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x
         * gridDim.x ) {
17
18 #if RESAMPLE == 1
19     curandStateXORWOW_t localState { };
20     curand_init( seed + tid, 0, 0, &localState );
21     T const distro = curand_uniform( &localState );
22 #endif
23
24     T const tidf { static_cast<T>( tid ) };
25     int l { 0 };
26     int idx { 0 };
27     T a { };
28     T b { };
29
30     if ( threadIdx.x < kWarpSize ) {  // If warp 0
31
32       bool mask { false };
33
34       if ( tid >= kTRI + l ) {
35         sWarp0[t] = devCumulativeSum[tid - kTRI - l];
36         sWarp0[t + kTRI] = devCumulativeSum[tid - l];
37       }
38
39       // Distribution will be the same for each Monte Carlo
40       T const draw = ( distro + tidf ) / n;
41
42       tile32.sync();
43
44       while ( !tile32.all( mask ) ) {
45
46         if ( tid >= kTRI + 32 + l ) {
47           a = devCumulativeSum[tid - ( kTRI + 32 ) - l];
48           b = devCumulativeSum[tid - 32 - l];
49
50           #pragma unroll
51           for ( int i = 1; i < kTRI + 1; i++ ) {
52             mask = sWarp0[t + kTRI - i] < draw;
53             if ( !mask )
54               idx--;
55           }
56           l += kTRI;
57           sWarp0[t] = a;
58           sWarp0[t + kTRI] = b;
59           tile32.sync();
60
61         } else {  // If warp 1
62
```

```
63            while ( !mask ) {
64              if ( tid > l )
65                mask = devCumulativeSum[tid - ( l + 1 )] < draw; // tid:5 -> dCS[4] ->
                      dCS[3] ->  dCS[2] -> dCS[1] -> dCS[0] ->
66              else
67                mask = true;
68              if ( !mask )
69                idx--;
70              l++;
71            }
72          }
73
74          tile32.sync();
75        }
76        devResampleIndexDown[tid] = idx;
77
78      } else {
79
80        bool mask { false };
81
82        if ( tid >= kTRI + l ) {
83          sWarp1[t] = devCumulativeSum[tid - kTRI - l];
84          sWarp1[t + kTRI] = devCumulativeSum[tid - l];
85        }
86
87        // Distribution will be the same for each Monte Carlo
88        T const draw = ( distro + tidf ) / n;
89
90        tile32.sync();
91
92        while ( !tile32.all( mask ) ) {
93
94          if ( tid >= kTRI + 32 + l ) {
95            a = devCumulativeSum[tid - ( kTRI + 32 ) - l];
96            b = devCumulativeSum[tid - 32 - l];
97
98            #pragma unroll
99            for ( int i = 1; i < kTRI + 1; i++ ) {
100             mask = sWarp1[t + kTRI - i] < draw;
101             if ( !mask )
102               idx--;
103           }
104           l += kTRI;
105           sWarp1[t] = a;
106           sWarp1[t + kTRI] = b;
107           tile32.sync();
108
109         } else {
110
111           while ( !mask ) {
112             if ( tid > l )
113               mask = devCumulativeSum[tid - ( l + 1 )] < draw; // tid:5 -> dCS[4] ->
                      dCS[3] ->  dCS[2] -> dCS[1] -> dCS[0] ->
114             else
115               mask = true;
116             if ( !mask )
117               idx--;
118             l++;
119           }
120         }
121
122         tile32.sync();
123       }
124       devResampleIndexDown[tid] = idx;
125     }
126   }
127 }
```

A.9 Metropolis (Naive)

```
template<typename T>
__global__ void __launch_bounds__(kTMR) computeResampleIndexMetropolis( int const n,
    unsigned long long const seed, int * __restrict__ devResampleIndexDown, T const *
     __restrict__ devParticleWeight ) {

  for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x
      * gridDim.x ) {

    uint idx { tid };
    uint key { };
    T den { devParticleWeight[idx] };
    T num { };
    T randNum { };

    curandStateXORWOW_t localState { };
    curand_init( seed + tid, 0, 0, &localState );

    for ( auto i = 0; i < CUTOFF; i++ ) {

      randNum = curand_uniform( &localState );
      key = static_cast<uint>( curand_uniform( &localState ) * ( n - 1 ) );
      num = devParticleWeight[key];

      if ( randNum <= ( num / den ) ) {
        den = num;
        idx = key;
      }
    }
    devResampleIndexDown[tid] = idx;
  }
}
```

A.10 Rejection

```
template<typename T>
__global__ void __launch_bounds__(kTMR) computeResampleIndexMetropolisRejection( int const n, unsigned long long const seed, int * __restrict__ devResampleIndexDown, T const * __restrict__ devParticleWeight ) {

  for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x * gridDim.x ) {

    uint idx {tid};
    uint key {};
    T den {devParticleWeight[idx]};
    T randNum {};

    curandStateXORWOW_t localState {};
    curand_init( seed + tid, 0, 0, &localState );

    randNum = curand_uniform( &localState );

    while (randNum > (den / 1.0f)) {
      // Subtract warp size so the last warp load out-of-bounds
      key = static_cast<uint>( curand_uniform( &localState ) * (n - 1) );
      den = devParticleWeight[key];
      randNum = curand_uniform( &localState );
    }

    devResampleIndexDown[tid] = key;
  }
}
```

A.11 Metropolis (Coalesced 1)

```
template<typename T>
__global__ void __launch_bounds__(kTMR) computeResampleIndexMetropolisC1( int const n
    , unsigned long long int seed, int * __restrict__ devResampleIndexDown, T const *
     __restrict__ devParticleWeight ) {

  for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x
      * gridDim.x ) {

    uint idx {tid};
    uint key {};
    uint warp {};
    T den {devParticleWeight[tid]};
    T num {};
    T randNum {};

    curandStateXORWOW_t localState {}, randState {};
    // same random number for 0-based warp entire grid
    curand_init( static_cast<unsigned long long int>( tid >> 5 ) + seed, 0, 0, &
        localState );
    curand_init( static_cast<unsigned long long int>( tid ) + seed, 0, 0, &randState
        );

    // Calculate s(warp) using warp index. All threads in warp have same value
    int SS = kWarpSize; // Size of segment
    int SC = n / SS; // The number of segments
    int DC = SS;
    // Random number [0 -> number of warps]
    warp = static_cast<uint>( curand_uniform( &localState ) * (SC - 1) );

    for ( auto i = 0; i < CUTOFF; i++ ) {

      randNum = curand_uniform( &randState );
      key = static_cast<uint>( curand_uniform( &randState ) * (DC - 1) );
      key = warp * DC + key;
      num = devParticleWeight[key];

      if ( randNum <= ( num / den ) ) {
        den = num;
        idx = key;
      }
    }
    devResampleIndexDown[tid] = idx;
  }
}
```

A.12 Metropolis (Coalesced 2)

```
template<typename T>
__global__ void __launch_bounds__(kTMR) computeResampleIndexMetropolisC2( int const n, unsigned long long int seed, int * __restrict__ devResampleIndexDown, T const * __restrict__ devParticleWeight ) {

  for ( auto tid = blockIdx.x * blockDim.x + threadIdx.x; tid < n; tid += blockDim.x * gridDim.x ) {

    uint idx {tid};
    uint key {};
    uint warp {};
    T den {devParticleWeight[tid]};
    T num {};
    T randNum {};

    curandStateXORWOW_t localState {}, randState {};
    // same random number for 0-based warp entire grid
    curand_init( static_cast<unsigned long long int>( tid >> 5 ) + seed, 0, 0, &localState );
    curand_init( static_cast<unsigned long long int>( tid ) + seed, 0, 0, &randState );

    // Calculate s(warp) using warp index. All threads in warp have same value
    int SS = kWarpSize; // Size of segment
    int SC = n / SS; // The number of segments
    int DC = SS;

    for ( auto i = 0; i < CUTOFF; i++ ) {

      // Random number [0 -> number of warps]
      warp = static_cast<uint>( curand_uniform( &localState ) * (SC - 1) );

      randNum = curand_uniform( &randState );
      key = static_cast<uint>( curand_uniform( &randState ) * (DC - 1) );
      key = warp * DC + key;
      num = devParticleWeight[key];

      if ( randNum <= ( num / den ) ) {
        den = num;
        idx = key;
      }
    }
    devResampleIndexDown[tid] = idx;
  }
}
```

APPENDIX B

B.1 Variance Effect on Naive Implementation: Systematic

Naive Method: Particles (1024)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	15	34	18	0	23	1	1	14,043	0	26	1	1	14,070
0.5	17	38	19	0	33	2	2	16,227	0	35	2	2	16,911
1.0	17	36	19	0	41	3	2	22,571	0	41	3	2	20,586
1.5	18	38	21	0	63	4	3	30,782	0	56	4	4	26,522
2.0	18	41	22	0	75	5	5	39,909	0	76	5	5	40,570
2.5	19	40	23	0	107	7	7	49,693	0	106	7	7	61,801
3.0	20	47	25	0	144	10	9	71,655	0	134	10	9	69,107
3.5	21	52	28	0	207	14	13	109,871	0	196	13	13	99,342
4.0	23	59	32	0	251	19	18	130,568	0	278	19	19	148,264

Naive Method: Particles (2048)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	17	38	20	0	39	2	2	49,184	0	39	2	2	35,032
0.5	18	35	20	0	43	3	3	49,394	0	47	3	3	47,514
1.0	18	34	21	0	64	4	4	59,094	0	69	4	4	66,854
1.5	18	44	22	0	79	5	5	78,058	0	85	5	5	96,870
2.0	19	42	24	0	108	7	7	104,665	0	116	8	7	132,122
2.5	20	48	26	0	151	10	10	138,159	0	156	10	10	142,988
3.0	21	54	29	0	197	14	14	202,271	0	185	14	14	204,311
3.5	22	61	32	0	270	19	19	271,382	0	278	20	19	307,554
4.0	25	73	38	0	392	28	27	380,445	0	381	27	26	411,612

Naive Method: Particles (4096)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	18	36	21	0	55	3	3	107,074	0	53	3	3	132,312
0.5	19	37	21	0	60	4	4	119,335	0	63	4	4	120,096
1.0	19	43	23	0	78	6	5	162,737	0	85	6	5	180,631
1.5	20	43	24	0	113	8	7	207,280	0	116	8	8	228,076
2.0	20	44	26	0	182	11	10	344,192	0	164	11	10	335,918
2.5	22	52	29	0	192	15	14	415,344	0	205	15	14	444,249
3.0	23	62	33	0	306	20	19	571,957	0	271	20	19	673,142
3.5	26	74	39	0	371	28	27	779,765	1	384	27	27	763,462
4.0	29	87	47	0	488	39	38	1,024,128	0	514	39	38	1,094,475

Naive Method: Particles (8192)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	19	38	23	0	77	5	5	277,984	0	79	5	5	351,184
0.5	19	45	23	0	81	6	6	315,669	0	84	6	6	360,824
1.0	20	43	24	0	113	8	8	496,037	0	110	8	8	475,319
1.5	21	49	27	0	159	11	11	624,667	0	167	11	11	790,147
2.0	23	53	30	0	237	15	15	824,227	0	212	15	15	1,042,645
2.5	24	62	34	0	286	21	20	1,174,082	1	335	21	20	1,460,934
3.0	26	75	40	1	394	28	28	1,658,350	0	382	29	28	1,646,390
3.5	30	92	48	0	512	39	38	2,185,470	1	546	39	38	2,232,463
4.0	34	110	60	0	650	54	53	3,006,540	0	676	56	54	2,986,704

Naive Method: Particles (16384)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	20	41	25	0	112	7	7	872,572	0	115	7	7	995,799
0.5	21	46	25	0	115	9	8	979,746	0	120	9	8	1,032,100
1.0	22	55	28	0	170	12	11	1,393,929	0	178	11	11	1,419,709
1.5	23	59	31	0	230	16	16	2,037,999	0	224	16	16	2,031,805
2.0	25	67	36	0	336	22	21	2,747,368	0	315	22	21	2,826,546
2.5	27	78	42	1	414	30	29	3,375,862	0	436	30	29	3,189,756
3.0	30	96	50	0	547	40	39	4,248,205	1	582	40	39	4,456,862
3.5	34	120	61	4	685	57	55	6,847,309	1	737	55	53	6,659,793
4.0	41	152	78	1	937	78	76	9,243,982	1	982	79	77	8,501,424

Naive Method: Particles (32768)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	25	51	30	0	171	11	10	3,249,074	0	162	11	10	3,320,119
0.5	25	55	32	0	194	12	12	3,061,915	0	168	12	12	2,890,829
1.0	27	58	35	0	232	17	16	3,426,888	0	252	17	16	3,878,895
1.5	28	78	40	0	315	23	22	5,081,087	0	313	23	22	5,363,604
2.0	31	90	46	0	464	31	30	6,954,286	0	427	31	30	6,704,842
2.5	34	108	55	0	591	41	40	9,565,324	1	614	43	42	9,486,122
3.0	38	129	67	0	753	58	56	13,027,754	1	784	56	55	14,937,875
3.5	41	161	83	3	974	79	77	16,514,483	0	1,007	79	77	18,499,407
4.0	53	210	109	1	1,301	111	108	24,557,152	3	1,354	113	109	23,565,627

Naive Method: Particles (65536)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	29	63	37	1	205	15	15	6,943,641	1	238	15	15	8,082,892
0.5	29	67	39	0	246	17	17	7,303,210	0	240	18	17	8,414,897
1.0	31	89	46	0	313	24	23	10,113,603	1	359	24	23	11,833,766
1.5	35	103	53	0	484	33	32	14,231,042	0	405	32	31	13,953,656
2.0	37	122	63	2	568	44	43	18,821,345	0	571	45	43	19,466,809
2.5	44	150	77	0	802	59	58	25,833,590	3	851	60	58	26,540,884
3.0	52	186	96	2	1,004	80	78	30,832,213	2	1,016	81	79	33,749,941
3.5	62	238	123	3	1,341	110	107	42,457,655	2	1,458	113	110	41,926,586
4.0	78	314	163	1	1,846	158	153	57,355,685	1	1,906	155	152	58,468,995

Naive Method: Particles (131072)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	37	109	57	1	324	22	21	18,081,327	1	304	22	21	19,158,912
0.5	40	118	61	1	332	25	24	19,587,893	0	333	25	25	21,119,801
1.0	44	140	73	0	471	33	32	24,595,041	1	425	34	33	27,027,164
1.5	49	172	90	4	577	45	44	31,271,098	2	575	46	45	33,703,505
2.0	59	216	113	1	754	61	60	41,483,177	2	818	63	61	43,426,888
2.5	70	272	143	5	1,016	82	81	55,275,763	2	1,010	84	82	56,202,818
3.0	84	350	184	0	1,228	112	111	70,117,789	5	1,314	112	111	74,527,430
3.5	106	466	244	7	2,007	153	153	95,976,428	4	1,929	155	153	102,206,876
4.0	138	636	333	19	2,771	218	216	133,185,435	12	2,487	216	214	137,853,663

Naive Method: Particles (262144)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	68	234	123	1	388	31	30	41,245,277	2	439	31	30	47,759,427
0.5	75	254	134	1	460	35	34	46,147,641	2	476	36	35	51,980,329
1.0	83	314	165	1	616	47	46	59,685,478	1	626	48	46	65,232,014
1.5	103	396	208	5	752	63	62	78,794,271	3	883	65	64	85,620,122
2.0	119	506	266	4	1,019	85	85	103,314,077	5	1,075	89	87	112,588,449
2.5	148	652	342	2	1,436	115	115	135,867,093	3	1,488	118	116	145,994,087
3.0	191	852	448	2	1,903	153	154	179,608,368	7	1,956	162	160	190,235,873
3.5	243	1,133	594	3	2,432	211	214	238,394,287	15	2,525	217	217	260,093,891
4.0	314	1,579	828	7	3,398	302	303	338,115,562	32	3,618	303	303	353,416,695

Naive Method: Particles (524288)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	151	564	297	1	516	43	42	110,525,507	2	543	44	43	115,987,187
0.5	169	624	328	1	564	49	49	120,015,322	4	618	50	49	130,875,053
1.0	191	774	407	2	777	65	65	156,715,967	6	833	66	65	169,289,565
1.5	244	1,003	527	2	1,059	90	89	209,222,607	3	1,095	92	90	221,528,586
2.0	298	1,278	671	2	1,403	119	120	271,433,471	5	1,416	122	121	291,893,875
2.5	352	1,685	882	12	1,989	161	162	361,773,948	2	2,012	164	164	381,489,617
3.0	464	2,285	1,192	5	2,656	220	220	493,794,500	9	2,684	224	223	513,559,826
3.5	624	3,079	1,606	6	3,425	295	300	654,488,089	11	3,737	303	304	689,749,528
4.0	862	4,351	2,275	16	4,925	421	426	925,360,238	27	5,273	427	429	927,631,163

Naive Method: Particles (1048576)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	333	1,382	728	3	752	59	59	282,041,563	2	779	62	61	300,948,892
0.5	390	1,544	815	4	802	69	69	317,463,747	8	887	70	69	340,610,191
1.0	444	1,959	1,031	3	1,121	90	91	409,166,815	3	1,153	94	93	436,208,347
1.5	573	2,596	1,367	6	1,504	124	125	548,666,195	4	1,499	127	127	574,364,383
2.0	701	3,453	1,810	9	2,055	170	171	736,055,246	8	2,111	170	171	771,123,305
2.5	845	4,612	2,410	18	2,651	227	229	986,002,857	9	2,610	230	231	1,007,757,094
3.0	1,248	6,282	3,266	36	3,544	307	311	1,336,726,314	24	3,630	313	314	1,407,651,485
3.5	1,429	8,525	4,442	28	5,043	420	428	1,772,503,159	13	5,250	421	430	1,926,741,172
4.0	2,145	12,166	6,329	24	6,725	591	602	2,474,845,005	36	7,119	601	608	2,543,237,914

B.2 Variance Effect on Improved Implementation: Systematic

Improved Method: Particles (1024)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	15	35	18	0	25	1	1	12,563	0	27	1	1	13,134
0.5	16	37	19	0	33	2	2	16,774	0	32	2	2	15,317
1.0	16	32	19	0	39	3	2	20,005	0	42	3	2	20,348
1.5	17	39	19	0	55	4	3	28,159	0	71	4	4	35,557
2.0	17	39	20	0	74	5	5	42,590	0	95	5	5	52,305
2.5	18	38	20	0	101	7	7	61,504	0	98	7	7	53,317
3.0	18	40	21	0	140	10	9	73,362	0	157	10	9	73,247
3.5	18	38	21	0	195	14	13	117,141	0	193	13	13	108,052
4.0	18	42	22	0	259	19	18	138,088	0	277	19	19	163,689

Improved Method: Particles (2048)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	19	33	20	0	41	2	2	42,815	0	38	2	2	39,149
0.5	18	39	20	0	46	3	3	43,361	0	43	3	3	46,795
1.0	19	35	20	0	63	4	4	60,939	0	60	4	4	60,332
1.5	19	39	20	0	79	5	5	87,084	0	80	5	5	78,369
2.0	19	39	20	0	108	7	7	132,678	0	119	8	7	112,770
2.5	19	41	21	0	150	10	10	149,502	0	153	10	10	160,773
3.0	19	37	21	0	213	14	14	218,423	0	194	14	14	218,482
3.5	19	39	21	0	290	19	19	280,047	0	270	20	19	259,107
4.0	19	42	22	0	381	27	26	412,884	0	397	27	27	373,677

Improved Method: Particles (4096)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	18	38	20	0	50	3	3	113,900	0	60	3	3	116,768
0.5	19	37	20	0	61	4	4	133,412	0	69	4	4	132,703
1.0	19	36	20	0	101	6	5	170,570	0	90	6	5	175,084
1.5	19	42	20	0	125	8	7	222,683	0	126	8	8	243,488
2.0	19	42	20	0	166	11	10	340,773	0	144	11	10	335,000
2.5	20	41	21	0	224	15	14	400,845	0	205	15	14	409,429
3.0	19	42	21	0	307	20	19	669,296	0	287	20	19	584,844
3.5	19	39	22	0	422	28	27	917,253	0	410	28	27	734,733
4.0	20	42	23	0	578	39	38	1,065,531	0	549	39	38	1,068,697

Improved Method: Particles (8192)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	18	37	21	0	84	5	5	344,715	0	77	5	5	324,354
0.5	19	41	21	0	86	6	6	333,311	0	86	6	6	347,942
1.0	19	36	21	0	157	8	8	600,883	0	130	8	8	470,094
1.5	19	39	21	0	181	11	11	652,084	0	145	11	11	624,514
2.0	19	38	21	0	206	15	15	896,732	0	211	15	15	887,658
2.5	19	41	21	0	299	21	20	1,162,607	0	329	21	20	1,313,808
3.0	19	40	22	0	370	28	28	1,661,196	0	412	28	28	1,817,993
3.5	20	40	22	1	508	39	38	2,084,392	0	533	40	38	2,249,305
4.0	19	41	23	0	851	55	54	3,593,413	0	734	55	54	2,905,647

Improved Method: Particles (16384)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	19	39	21	0	106	7	7	891,737	0	104	7	7	987,104
0.5	20	39	21	0	122	8	8	1,052,954	0	120	9	8	1,196,296
1.0	20	41	21	0	165	11	11	1,555,590	0	178	12	11	1,393,749
1.5	20	38	22	0	237	16	15	1,933,794	0	252	16	16	2,011,901
2.0	20	43	22	0	292	22	21	2,436,601	0	300	22	21	2,673,895
2.5	20	39	22	0	486	30	29	4,149,539	0	409	30	29	3,095,610
3.0	20	42	23	1	539	40	39	4,638,941	1	578	41	40	4,850,497
3.5	21	46	23	1	742	55	54	6,651,402	0	787	57	54	5,978,023
4.0	21	46	25	2	1,024	79	77	8,705,031	2	1,036	78	76	9,317,496

Improved Method: Particles (32768)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	25	40	28	0	155	11	10	2,638,124	0	152	11	10	2,519,122
0.5	25	47	28	1	184	12	12	3,130,646	0	189	12	12	2,946,591
1.0	26	45	28	0	237	16	16	3,723,693	0	261	17	16	4,211,337
1.5	25	44	28	0	324	23	22	5,128,370	2	360	23	22	5,404,120
2.0	25	48	29	0	410	31	30	6,848,322	1	440	31	30	6,695,924
2.5	27	48	29	0	613	42	41	10,065,913	0	632	42	41	10,617,933
3.0	26	59	31	1	774	57	55	13,628,503	4	755	58	56	11,274,322
3.5	27	52	32	3	1,069	79	76	17,619,535	0	1,113	79	77	16,584,409
4.0	28	66	34	4	1,393	111	108	25,722,686	1	1,654	112	109	27,514,266

Improved Method: Particles (65536)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	28	46	31	0	226	15	15	8,150,984	0	198	15	15	8,050,415
0.5	29	45	31	0	259	17	17	8,150,281	0	269	18	17	8,304,139
1.0	29	52	32	0	318	24	23	10,706,964	1	327	24	23	11,979,178
1.5	29	53	33	0	457	33	32	14,345,207	2	419	33	32	14,776,303
2.0	29	56	34	0	580	45	43	21,142,322	0	588	45	43	22,306,243
2.5	30	58	36	1	808	60	58	23,713,926	3	854	60	58	31,948,670
3.0	31	68	39	1	1,130	81	79	34,374,626	3	1,204	80	78	40,136,019
3.5	32	78	42	1	1,612	113	110	51,111,637	0	1,548	113	109	50,980,780
4.0	34	91	48	4	2,146	158	154	66,755,247	2	2,145	158	154	71,576,139

Improved Method: Particles (131072)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	34	56	38	1	380	22	21	21,963,816	1	339	22	21	22,221,643
0.5	35	61	39	1	367	25	24	24,911,429	0	380	26	25	25,755,573
1.0	36	69	42	1	468	34	33	31,635,471	2	476	34	32	32,616,537
1.5	36	74	44	1	636	48	46	45,410,093	1	682	45	44	41,653,922
2.0	38	84	48	1	840	63	61	53,581,806	3	881	61	60	56,371,927
2.5	39	102	54	1	1,202	85	82	72,299,319	1	1,237	85	82	79,914,542
3.0	41	117	61	5	1,494	114	111	90,327,681	2	1,575	117	112	95,736,260
3.5	45	138	72	6	1,974	157	154	112,457,104	5	2,181	159	154	117,309,507
4.0	51	168	87	2	2,793	219	215	142,294,960	9	2,725	222	217	149,501,469

Improved Method: Particles (262144)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	54	100	64	1	487	31	30	54,661,972	2	443	31	30	56,442,058
0.5	55	113	66	0	569	36	35	72,185,577	1	514	36	35	61,899,448
1.0	56	137	71	2	723	47	46	90,874,494	2	720	48	47	89,912,398
1.5	59	151	78	4	866	64	63	115,178,934	3	869	66	64	114,733,041
2.0	60	170	89	5	1,204	88	86	136,434,231	1	1,193	90	87	140,947,540
2.5	65	196	103	9	1,524	119	116	164,015,182	5	1,486	120	117	168,717,616
3.0	70	232	122	0	1,909	158	156	204,113,329	5	2,063	161	158	209,426,451
3.5	78	286	151	11	2,781	217	216	262,368,565	2	2,745	221	219	267,810,047
4.0	92	368	193	5	3,682	306	304	354,103,625	9	3,597	305	302	357,253,717

Improved Method: Particles (524288)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	100	245	127	3	708	43	42	143,166,004	1	603	44	43	154,473,073
0.5	102	250	133	1	717	50	49	177,216,998	2	796	51	50	199,052,546
1.0	106	282	147	2	848	67	65	218,050,262	2	900	68	66	230,633,428
1.5	111	324	170	1	1,151	92	90	261,423,106	6	1,132	93	90	278,678,148
2.0	118	378	199	9	1,518	125	122	322,215,238	4	1,567	124	122	335,569,563
2.5	132	450	237	8	2,073	166	165	404,824,551	2	2,078	165	164	417,707,759
3.0	146	558	293	11	2,834	223	222	525,997,783	32	2,757	226	223	537,764,341
3.5	162	700	368	14	3,933	303	303	690,597,965	26	3,644	307	307	703,134,001
4.0	185	930	486	40	4,837	435	433	955,715,405	27	5,250	423	425	954,586,655

Improved Method: Particles (1048576)													
y	Time (μs)			Up Kernel					Down Kernel				
	Min	Max	Mean	Min	Max	Mean	Std	Total (ℓ)	Min	Max	Mean	Std	Total (ℓ)
0.0	191	514	269	2	830	62	61	403,061,082	5	797	63	61	420,034,686
0.5	194	541	285	3	1,087	71	70	436,427,120	4	956	73	71	444,394,797
1.0	208	620	327	7	1,199	96	93	527,585,803	3	1,234	96	93	541,446,521
1.5	224	733	388	7	1,633	132	129	663,910,867	13	1,624	127	125	672,092,184
2.0	230	888	469	6	2,155	173	171	842,951,465	4	2,081	175	173	843,262,381
2.5	279	1,099	578	5	2,806	234	231	1,082,761,879	7	2,750	233	231	1,077,060,689
3.0	331	1,406	736	7	3,631	313	312	1,416,104,867	7	3,674	314	314	1,423,866,332
3.5	373	1,870	979	10	5,104	421	424	1,883,189,492	35	5,171	443	438	1,918,900,556
4.0	475	2,532	1,326	13	7,211	607	609	2,645,548,143	46	6,720	591	598	2,677,374,746

REFERENCES

[1] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME Journal of Basic Engineering*, no. 82 (Series D), pp. 35–45, 1960.

[2] S. Särkkä, T. Tamminen, A. Vehtari, and J. Lampinen, "Probabilistic methods in multiple target tracking," *Technical Report*, 2004.

[3] P. Kaniewski, "Structures, models and algorithms in integrated positioning and navigation systems," *WAT, Warszawa*, 2010.

[4] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-gaussian bayesian state estimation," *IEEE Proceedings F - Radar and Signal Processing*, vol. 140, pp. 107–113, April 1993.

[5] G. Kitagawa, "Monte carlo filter and smoother for non-gaussian nonlinear state space models," *Journal of Computational and Graphical Statistics*, vol. 5, no. 1, pp. 1–25, 1996.

[6] J. Carpenter, P. Clifford, and P. Fearnhead, "Improved particle filter for nonlinear problems," *IEEE Proceedings - Radar, Sonar and Navigation*, vol. 146, pp. 2–7, Feb 1999.

[7] T. Schön, F. Gustafsson, and P. J. Nordlund, "Marginalized particle filters for mixed linear/nonlinear state-space models," *IEEE Transactions on Signal Processing*, vol. 53, pp. 2279–2289, July 2005.

[8] G. Hendeby, R. Karlsson, F. Gustafsson, and N. Gordon, "Performance issues in non-gaussian filtering problems," in *2006 IEEE Nonlinear Statistical Signal Processing Workshop*, pp. 65–68, Sept 2006.

[9] K. Nummiaro, E. Koller-Meier, and L. V. Gool, "An adaptive color-based particle filter," *Image and Vision Computing*, vol. 21, no. 1, pp. 99 – 110, 2003.

[10] A. Doucet and A. M. Johansen, "A tutorial on particle filtering and smoothing: Fifteen years later," 2008.

[11] T. Flury and N. Shephard, "Bayesian inference based only on simulated likelihood: Particle filter analysis of dynamic economic models," *Econometric Theory*, vol. 27, no. 5, p. 933956, 2011.

[12] D. Weikersdorfer and J. Conradt, "Event-based particle filtering for robot self-localization," in *2012 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 866–870, Dec 2012.

[13] J. Li and J. Zhang, "Research on the algorithm of multi-autonomous underwater vehicles navigation and localization based on the extended kalman filter," in *2016 IEEE International Conference on Mechatronics and Automation*, pp. 2455–2460, Aug 2016.

[14] A. Doucet, N. J. Gordon, and V. Krishnamurthy, "Particle filters for state estimation of jump markov linear systems," *IEEE Transactions on Signal Processing*, vol. 49, pp. 613–624, Mar 2001.

[15] C. Musso, N. Oudjane, and F. Le Gland, "Improving regularized particle filters," in *Sequential Monte Carlo methods in practice* (A. Doucet, N. de Freitas, and N. Gordon, eds.), Statistics for Engineering and Information Science, pp. 247–271, Springer, 2001.

[16] B. W. Silverman, *Density estimation for statistics and data analysis / B.W. Silverman.* Chapman and Hall London ; New York, 1986.

[17] L. Devroye, *A Course in Density Estimation Density Estimation.* Birkhauser, 1987.

[18] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-gaussian bayesian tracking," *IEEE Transactions on Signal Processing*, vol. 50, pp. 174–188, Feb 2002.

[19] A. Kong, J. Liu, and W. Hung Wong, "Sequential imputations and bayesian missing data problems," *JASA. Journal of the American Statistical Association*, vol. 89, 03 1994.

[20] V. Zaritskii, V. Svetnik, and L. Šimelevič, "Monte-carlo technique in problems of optimal information processing," *Avtomatika i Telemekhanika*, no. 12, pp. 95–103, 1975.

[21] A. Doucet, S. Godsill, and C. Andrieu, "On sequential monte carlo sampling methods for bayesian filtering," *Statistics and computing*, vol. 10, no. 3, pp. 197–208, 2000.

[22] P. E. Jacob, "Sequential bayesian inference for implicit hidden markov models and current limitations," *ESAIM: Proceedings and Surveys*, vol. 51, pp. 24–48, 2015.

[23] J. Zuo, "Dynamic resampling for alleviating sample impoverishment of particle filter," *IET Radar, Sonar Navigation*, vol. 7, pp. 968–977, December 2013.

[24] W. R. Gilks and C. Berzuini, "Following a moving target-monte carlo inference for dynamic bayesian models," *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, vol. 63, no. 1, pp. 127–146, 2001.

[25] T. Fetzer, F. Ebner, F. Deinzer, and M. Grzegorzek, "Recovering from sample impoverishment in context of indoor localisation," in *2017 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pp. 1–8, Sept 2017.

[26] T. Li, S. Sun, T. P. Sattar, and J. M. Corchado, "Review: Fight sample degeneracy and impoverishment in particle filters: A review of intelligent approaches," *Expert Syst. Appl.*, vol. 41, pp. 3944–3954, June 2014.

[27] E. R. Beadle and P. M. Djurić, "A fast-weighted bayesian bootstrap filter for nonlinear model state estimation," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 33, pp. 338–343, Jan 1997.

[28] L. M. Murray, A. Lee, and P. E. Jacob, "Parallel resampling in the particle filter," *Journal of Computational and Graphical Statistics*, vol. 25, no. 3, pp. 789–805, 2016.

[29] T. Li, M. Bolić, and P. M. Djurić, "Resampling methods for particle filtering: Classification, implementation, and strategies," *IEEE Signal Processing Magazine*, vol. 32, pp. 70–86, May 2015.

[30] Ö. Dülger, H. Oğuztüzün, and M. Demirekler, "Memory coalescing implementation of metropolis resampling on graphics processing unit," *Journal of Signal Processing Systems*, vol. 90, no. 3, pp. 433–447, 2018.

[31] D.-H. Kim, "Evaluation of the performance of gpu global memory coalescing," *Journal of Multidisciplinary Engineering Science and Technology*, vol. 4, no. 4, p. 5, 2017.

[32] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2019. Available at https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, version 10.1.105.

[33] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210*, NVIDIA Corporation, 2014. Available at https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[34] *NVIDIA GeForce GTX 980*, NVIDIA Corporation, 2014. Available at https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF.

[35] *NVIDIA Tesla P100*, NVIDIA Corporation, 2016. Available at https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[36] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *CoRR*, vol. abs/1804.06826, 2018.

[37] *NVIDIA TESLA V100 GPU ARCHITECTURE*, NVIDIA Corporation, 2017. Available at https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[38] *NVIDIA TURING GPU ARCHITECTURE*, NVIDIA Corporation, 2018. Available at https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[39] B. G. Sileshi, C. Ferrer, and J. Oliver, "Particle filters and resampling techniques: Importance in computational complexity analysis," in *2013 Conference on Design and Architectures for Signal and Image Processing*, pp. 319–325, Oct 2013.

[40] M. A. Chao, C. Y. Chu, C. H. Chao, and A. Y. Wu, "Efficient parallelized particle filter design on cuda," in *2010 IEEE Workshop On Signal Processing Systems*, pp. 299–304, Oct 2010.

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

[42] P. Gong, Y. O. Basciftci, and F. Ozguner, "A parallel resampling algorithm for particle filtering on shared-memory architectures," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pp. 1477–1483, May 2012.

[43] S. Maskell, B. Alun-Jones, and M. Macleod, "A single instruction multiple data particle filter," in *2006 IEEE Nonlinear Statistical Signal Processing Workshop*, pp. 51–54, Sept 2006.

[44] M. A. Nicely and B. E. Wells, "Improved parallel resampling methods for particle filtering," *IEEE Access*, vol. 7, pp. 47593–47604, 2019.

[45] P. L'Ecuyer, "Random number generation with multiple streams for sequential and parallel computing," in *2015 Winter Simulation Conference (WSC)*, pp. 31–44, Dec 2015.

[46] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, pp. 46–55, Jan 1998.

[47] P. LEcuyer, D. Munger, B. Oreshkin, and R. Simard, "Random numbers for parallel computers: Requirements and methods, with emphasis on gpus," *Mathematics and Computers in Simulation*, vol. 135, pp. 3 – 17, 2017. Special Issue: 9th IMACS Seminar on Monte Carlo Methods.

[48] G. Marsaglia, "The marsaglia random number cdrom: Including the diehard battery of tests of randomness," 1995. Computer file.

[49] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, pp. 22:1–22:40, aug 2007.

[50] M. Manssen, M. Weigel, and A. K. Hartmann, "Random number generators for massively parallel simulations on gpu," *The European Physical Journal Special Topics*, vol. 210, pp. 53–71, Aug 2012.

[51] L. E. Bassham, III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," tech. rep., Gaithersburg, MD, United States, 2010.

[52] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.

[53] M. Saito and M. Matsumoto, "Variants of mersenne twister suitable for graphic processors," *ACM Trans. Math. Softw.*, vol. 39, pp. 12:1–12:20, Feb. 2013.

[54] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software, Articles*, vol. 8, no. 14, pp. 1–6, 2003.

[55] P. L'Ecuyer, "Good parameters and implementations for combined multiple recursive random number generators," 1998.

[56] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: As easy as 1, 2, 3," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 16:1–16:12, ACM, 2011.

[57] D. Merrill, "Cuda unbound." Available at http://nvlabs.github.io/cub/, version 1.8.0.

[58] A. G. Barnston, "Correspondence among the correlation, rmse, and heidke forecast verification measures; refinement of the heidke score," *Weather and Forecasting*, vol. 7, no. 4, pp. 699–709, 1992.

[59] W. Kahan, "Pracniques: Further remarks on reducing truncation errors," *Commun. ACM*, vol. 8, p. 40, Jan 1965.