Western Michigan University
ScholarWorks at WMU

Dissertations | Graduate College

6-2019

High-Performance Quasi-Monte Carlo Integration and Applications

Ahmed Hassan H. Almulihi
Western Michigan University, a.almulihi@gmail.com

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

 Part of the Computer Sciences Commons

Recommended Citation

Almulihi, Ahmed Hassan H., "High-Performance Quasi-Monte Carlo Integration and Applications" (2019). *Dissertations*. 3452.
https://scholarworks.wmich.edu/dissertations/3452

This Dissertation-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Dissertations by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.

HIGH-PERFORMANCE QUASI-MONTE CARLO INTEGRATION AND APPLICATIONS

by

Ahmed Hassan H. Almulihi

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Computer Science
Western Michigan University
June 2019

Doctoral Committee:

Dr. Elise de Doncker, Chair
Dr. John Kapenga
Dr. Joseph McKean

Copyright by
Ahmed Hassan H. Almulihi
2019

ACKNOWLEDGEMENTS

In the name of Allah, the most gracious, the most merciful. All thanks to Allah, the lord of the worlds, for his guidance to complete this work.

First, I would like to thank my supervisor, Prof. Elise de Doncker, for the patient guidance, motivation and advice she has provided throughout my time as her student. I have been very lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly. Besides my advisor, I would like to thank the other members of my thesis committee, Prof. John Kapenga and Prof. Joseph McKean, for their encouragement and insightful comments.

I would like to express my sincere gratitude to my scholarship sponsors, the Ministry of Education and Taif University, Saudi Arabia for their financial support during my Master's and Ph.D. studies. Also, I would like to thank the Department of Computer Science at Western Michigan University for the opportunities and the support.

I would also like to thank all of my friends who supported me during this journey, and encouraged me to accomplish my goals. A very special thanks to my family. Words cannot express how grateful I am to my mother and father for all of the sacrifices they made on my behalf. Their prayer for me was what kept me going.

Finally, I would like to express my sincere appreciation to my beloved wife, Ohud, who spent years with me away from home and was always my support in the moments of weakness and depression.

Ahmed Hassan H. Almulihi

HIGH-PERFORMANCE QUASI-MONTE CARLO INTEGRATION AND APPLICATIONS

Ahmed Hassan H. Almulihi, Ph.D.

Western Michigan University, 2019

While adaptive integration by region partitioning is generally effective in low dimensions, quasi-Monte Carlo methods can be used for integral approximations in moderate to high dimensions. Important application areas include high-energy physics, statistics, computational finance and stochastic geometry with applications in robotics, tessellations and imaging from medical data using tetrahedral meshes.

Lattice rule integration is a class of quasi-Monte Carlo methods, implemented by an equal-weight cubature formula and suited for fairly smooth functions. Successful methods to construct these rules are the component-by-component (CBC) algorithm by Sloan and Restsov (2001) and the fast algorithm for CBC by Nuyens and Cools (2006). As the ability to invoke a large number of function evaluations is an important factor in high-dimensional integration, we investigate the acceleration of the CBC construction for large rank-1 lattice rules using the CUDA (cuFFT) Fast Fourier Transform procedure.

A major part of this study is the development of high-performance lattice rule algorithms for approximating moderate- to high-dimensional integrals on GPUs. Lattice rules are incorporated with a periodizing transformation. We show that rank-1 lattice rules on GPUs (possibly with an integral transformation to alleviate the effects of boundary singularities) yield better accuracy and efficiency for various classes of integrals compared to classic Monte Carlo and adaptive methods. The computational power of GPU accelerators also leads to

significant improvements in efficiency and accuracy for integration based on embedded (composite) lattices.

These methods have been motivated as possible contributions to high-performance computing software such as the ParInt multivariate integration package developed at WMU. We further show an application in Bayesian analysis, leading to a class of problems where the integrand has a dominant peak in the integration domain. We demonstrate a black-box approach provided by the adaptive strategies in the ParInt package.

TABLE OF CONTENTS

Table of Contents—Continued

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

HPC	**H**igh **P**erformance **C**omputing
MPI	**M**essage **P**assing **I**nterface
CBC	**C**omponent **B**y **C**omponent
FFT	**F**ast **F**ourier **T**ransform
IFFT	**I**nverse **F**ast **F**ourier **T**ransform
MC	**M**onte **C**arlo
QMC	**Q**uasi **M**onte **C**arlo
ParInt	**Par**allel **Int**egration package
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
GPU	**G**raphical **P**rocessing **U**nit

NOTATION

n	number of points
d	number of dimensions
$\mathbf{z}$	lattice rules generating vector
$\mathbb{R}$	set of real numbers
$\mathbb{Z}$	set of integer numbers
$\{x\}$	fractional part of x ($\{x\} = x \mod 1$)
$\mathcal{H}$	Hilbert space of functions
$\mathcal{H}(K)$	reproducing kernel Hilbert space $\mathcal{H}$ with kernel K

CHAPTER 1

INTRODUCTION

1.1. Overview

The *curse of dimensionality* is a concept first introduced by Richard Bellman in 1957 to describe the increasing difficulty of a problem as the number of variables (dimension) increases [1]. Integration is one of the problems that are affected by the curse of dimensionality and may become *intractable* for higher integral dimensions. Computing high-dimensional integrals is still a major issue in the area of scientific computation [2], and is mostly unsolved without a priori information on the integrand behavior.

Multivariate integration problems arise in many scientific fields. Typical applications include the computation of the multivariate normal [3], stochastic geometry integrals arising in tesselations [4], Bayesian inference [5], and financial derivatives where, e.g., the Mortgage Backed Security (MBS) problem [6]–[8] leads to high-dimensional integrals. When the number of dimensions increases, the integral becomes harder to approximate and consumes more computation time. However, advances in computational power have allowed for more efficient approximations of higher-dimensional integrals using stochastic techniques [9].

1.2. Multivariate Integration

Our aim is to approximate multivariate integrals over the d-dimensional unit hypercube of the form

$$I(f) = \int_{[0,1]^d} f(\boldsymbol{x})\, d\boldsymbol{x} = \int_0^1 \dots \int_0^1 f(x_1, \dots, x_d)\; dx_1 \dots dx_d \tag{1.1}$$

Finding an exact integral value or a closed form for such an integral may be hard or impossible. The dimension d is often large in practical applications.

Approximating a multivariate integral using a product of classical (1D) quadrature rules:

$$Q(f) = \sum_{n_1=0}^{m} ... \sum_{n_d=0}^{m} q_{n_1}...q_{n_d}\ f(x_{n_1}, ..., x_{n_d})$$

with quadrature points $x_0, ..., x_m$ in $[0, 1]$ and with weights $q_0, ..., q_m \in \mathbb{R}$, is impractical in moderate dimensions even with the most powerful computers existing today.

1.3. Approximation Methods

1.3.1. Monte Carlo

The Monte Carlo integration method is similar to performing a number of random experiments to answer some "What if" questions. Physics researchers Ulam, Fermi, von Neumann, and Metropolis promoted the name "Monte Carlo" as a reference to a famous casino in Monaco where Ulam's uncle would borrow money to gamble [10]. Monte Carlo integration is one of the most popular tools to deal with higher dimensional integrals and with erratic integrand functions or regions. It uses random sampling to calculate an approximation to Eq (1.1), yielding:

$$Q_n(f) = \frac{1}{n} \sum_{j=0}^{n-1} f\left(\mathbf{x}_j\right) \tag{1.2}$$

where the $\mathbf{x}_j$ are uniform random in $[0, 1]^d$. Based on the Strong Law of Large Numbers [11], the probability of this approximation to converge is 1,

$$\lim_{n\to\infty} Q_n(f) \to I(f) \tag{1.3}$$

under general conditions. The integration error is defined as

$$e_n(f) = I(f) - Q_n(f) \tag{1.4}$$

The basic idea of Monte Carlo integration is to use a large number of random samples (points) and to evaluate the integrand at each point in order to calculate an average result. Monte Carlo may be the only feasible method if the integrand is badly behaved, or if the integration region is irregular, or if the problem dimension is high. However, Monte Carlo integration has a very slow convergence rate ($O(1/\sqrt{n})$ as n grows large), and thus requires a large number of points in order to reduce the error margin [12]. Each integrand evaluation may take a long time, leading to an extensive overall computation time.

Monte Carlo methods depend heavily on using random numbers, thus it is an issue that the numbers generated by computers are not quite random. They are generated as pseudo-random sequences because they are obtained using a deterministic approach, which can lead to repeatable and predictable sequences. Pseudo-random number generators are initialized by a single value called the *seed*. The seed is set by default in some systems, or it can be generated generated based on some factors such as the computer clock, but in most tools the user can set it [13]. It is important to use a reliable pseudo-random number generator in order to achieve a good approximation, especially if n is very large [12]. Another issue is the accumulation of roundoff error in large summations (see, e.g., [14]).

Monte Carlo integration is a powerful method and it is easy to implement. With $d = 300$ or more in Eq (1.2), applying Monte Carlo sampling can still give consistent results in some cases [15]. On the downside, Monte Carlo can be extremely slow and computationally intensive. The integration with this method has an error of $O(1/\sqrt{n})$ independent of the dimension of the integral, so that we need a vary large number of points in order to achieve

satisfactory results. Furthermore, the standard error bound in Monte Carlo integration is proportional to $\sigma/\sqrt{n}$ [16]. Reducing σ can help the computational performance, as is the goal of variance reduction techniques such as: Antithetic Variates, Stratification, Importance Sampling and Common Random Numbers [17].

1.3.2. Quasi-Monte Carlo

Instead of random points in Monte Carlo, quasi-Monte Carlo techniques use deterministic point sequences that are evenly spread over the integration region. Quasi-random sequences are designed to provide better uniformity than a random sequence, and hence faster convergence for the associated integration formulas [18].

Lattice rule integration is a class of quasi-Monte Carlo methods suited for fairly smooth functions in higher dimensions [19] [20]. An *integration lattice* in $\mathbb{R}^d$ is a discrete subset of $\mathbb{R}^d$ that is closed under addition and subtraction and which contains the integer vectors of $\mathbb{Z}^d$ as a subset [21]. An integration lattice can be constructed from a generator vector that must be carefully chosen based on the number of dimensions and the number of points. Not unlike Monte Carlo, integration with lattice rules may need a large number of points in order to reach a target accuracy; however the rate of convergence is faster under certain conditions. Fig. 1.1 shows a set of random points used by Monte Carlo (where the coordinates are drawn from a uniform random distribution) and the deterministic points of a 2-dimensional integration lattice.

A lattice rule is defined on the unit hypercube. A *rank-1* lattice rule is generated using a generator vector $\mathbf{z}$ [22] [21], which allows for an approximation of the integral in Eq. (1.1) of the form,

$$Q_n(f) = \frac{1}{n} \sum_{j=0}^{n-1} f\left(\left\{\frac{j\mathbf{z}}{n}\right\}\right) \tag{1.5}$$

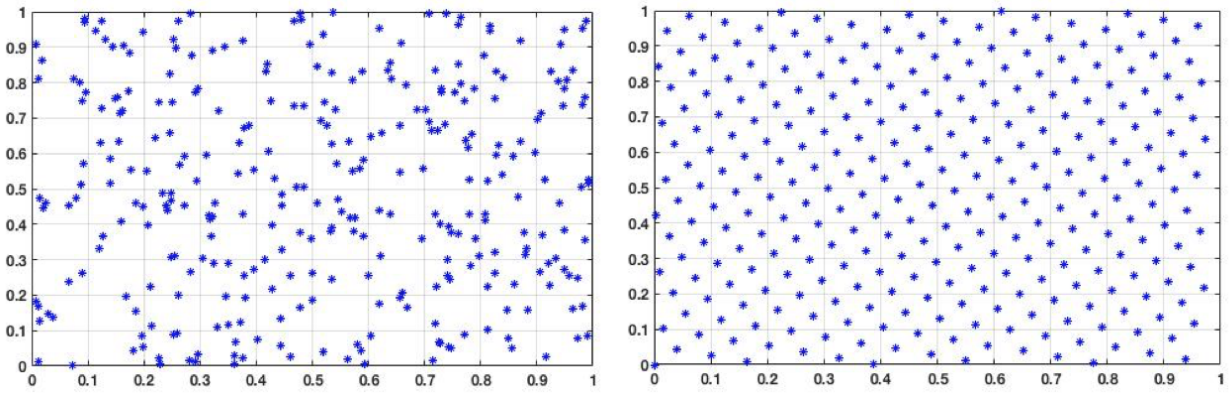


Figure 1.1: 300 random points (left) and 300 points of a 2-dimensional lattice constructed from generator vector $\mathbf{z}^t = (1, 129)$ (right).

Here n is the number of points, $\mathbf{z}$ is an integer generator vector, and $\{x\}$ denotes the fractional part of x, so that, $\{j\mathbf{z}/n\} = \dfrac{j\mathbf{z}}{n} \pmod 1$, component-wise (see [23]).

The vector $\mathbf{z} = (z_1, z_2, ..., z_d)^t$ is of length d (the dimension of the integral), and its components have no factor in common with n [21]. A popular type of lattice rule integration has been the Korobov method (1959) [20], where $\mathbf{z}$ is obtained as

$$\mathbf{z} = (1, a, a^2 \bmod n, a^3 \bmod n, ..., a^{d-1} \bmod n)^t \tag{1.6}$$

and a is an integer satisfying $1 \leq a \leq n-1$ and $\gcd(a, n) = 1$.

The theory of good lattice points relies on properties of periodic functions, and supplies error bounds for functions that are periodic (with period 1) in all variables. The method is extended to sufficiently smooth non-periodic functions by periodizing transformations.

On the downside, integration by rank-1 lattice rules does not provide a direct way of error estimation [21]. Cranley and Patterson [24] proposed a method of error estimation based on

the *randomly shifted lattice rules* of the form

$$Q(\mathbf{z}, n, \mathbf{c})f = \frac{1}{n}\sum_{j=0}^{n-1} f\left(\left\{\frac{j\mathbf{z}}{n} + \mathbf{c}\right\}\right), \tag{1.7}$$

where $\mathbf{c}$ is a uniform random vector. Shifting the lattice rule points is also useful for making the integration region boundaries clear from any integration points [21]. To obtain an error estimate, one can apply multiple randomly shifted lattice rules using different values of $\mathbf{c}$, and compute the mean of the results, with a mean squared error estimate.

1.3.3. Adaptive Integration

In the adaptive integration method for multivariate integrals, the integration domain is subdivided repeatedly into smaller subregions until the process reaches a target accuracy. This procedure can only be used in fairly low dimensions (say, less than 10 or 12), as the effort to partition a multivariate region around hot spots increases exponentially with the dimension [25].

Generally, an adaptive method has four main components [21]: quadrature/cubature rules for approximating the integral over subregions of the integration domain; an error estimation method; an error acceptance standard; and a subdivision method. There are a number of adaptive integration packages that apply different subdivision strategies. We will describe the ParInt package, which uses parallel adaptive integration, in Chapter 2.

1.4. Related Work

The development of computational power over the past century has led to numerous research efforts to address the curse of dimensionality. Many efforts have been made to improve the slow converge rate of the Monte Carlo method [18]. The use of parallel computing

to improve Monte Carlo performance received a lot of attention and achieved noticeable results. For example, Kanzaki [26] used a graphics processing unit (GPU) to accelerate the performance of VEGAS and BASES, which are Monte Carlo integration programs. The experiment achieved a speedup of 50 compared to the sequential programs. Another implementation of Monte Carlo on GPU was done by de Doncker and Assaf in [4] and by de Doncker, Kapenga and Assaf in [27] with more significant speedups. Szałkowski and Stpiczyǹski [28] implemented Monte Carlo integration using distributed memory computers and clusters. The authors showed results of various parallel methods including MPI (Message Passing Interface), OpenMP and CUDA. They concluded that Monte Carlo integration can be implemented efficiently on distributed memory computers. An implementation of parallel Monte Carlo using an Intel Xeon Phi accelerator was presented by Shareef and de Doncker [29].

There have been many tools and software packages to perform numerical integration. One such package is CUBPACK by Cools et al. [30] [31], which is an adaptive integration package that is written in FORTRAN 90 for multivariate integration over hyper-rectangles and simplex regions. Another software is CUBA by Hahn [32] [33], which includes four algorithms based on Monte Carlo/quasi-Monte Carlo integration (Vegas, Suave and Divonne), and Cuhre based on cubature rules for adaptive integration. The package is written in C and offer interfaces for Fortran, C/C++ and Mathematica.

Iterated or repeated integration with (one-dimensional) adaptive QuadPack programs has been proven effective for applications to Feynman loop integrals with severe singularities in high-energy physics [34]. However, a drawback of iterated integration is the expense of the method for problems of dimensions, say, ≥ 6.

In a sequential setting, multivariate adaptive integration is considered suitable for moderate dimensions (say, up to 10) while possibly dealing with mildly irregular integrand behavior,

and higher dimensions for well-behaved functions. These limits can be increased somewhat in a parallel environment, for allowing finer subdivisions with higher numbers of integrand evaluations.

Quasi-Monte Carlo rules can be applied for higher dimensions and smooth integrands. Solved by a parallel quasi-Monte Carlo method in ParInt [6], excellent speedups are reported for a 360-dimensional mortgage backed security problem from [8], [35]).

There are a number of algorithms to construct lattice rules. A recent algorithm is the component-by-component construction of good lattice rules (CBC) introduced by Sloan and Reztsov in (2002). The algorithm minimizes the worst-case error with respect to a reproducing kernel Hilbert space [36]. The CBC algorithm was a breakthrough in constructing lattice rules efficiently even for large numbers of points and dimensions [22]. Nuyens and Cools [23] used Fast Fourier Transform (FFT) operations to improve the CBC algorithm time from $O(dn^2)$ to $O(dn\ \log(n))$ where d is the dimension and n is the number of points. This allows the algorithm to deal with much larger n in far less time than the original algorithm. Whereas the space (memory) complexity of the CBC algorithm is $O(n)$ (linear in n), the fast CBC algorithm space complexity is still $O(n)$ but in fact doubles the memory requirement. The construction of good lattice rules will be discussed further in Chapter 3.

There are a limited number of software tools available to construct lattice rules. These include:

- **Fast component-by-component construction**: The Matlab code by Nuyens [37], which generates rank-1 lattice rules and embedded sequence rules with product and order-dependent weights.

- **Stochastic Simulation in Java (SSJ)**: A Java-based framework for Stochastic Simulation [38]. Although this library is not intended for integration, it is capable of

producing deterministic, low-discrepancy sequences that can be used for quasi-Monte Carlo integration [39].

- **RandQMC**: A package written in C that contains various quasi-Monte Carlo methods for multidimensional integration [40].

- **Lattice Builder**: a software library written in C++ by L'Ecuyer and Munger [41], which includes various construction algorithms for good rank-1 lattice rules. It supports exhaustive and random searches, as well as CBC and random CBC constructions, for various measures of uniformity of the points.

1.5. Motivation

Many Monte Carlo and quasi-Monte Carlo algorithms were developed before the age of parallel processing and high performance computing. There has been a considerable amount of work done in parallelizing the existing algorithms, which was intuitive especially since most of these algorithms are embarrassingly parallel [42]. However, there are some stochastic techniques that cannot be easily adapted to parallel computing [43]. In the course of this research, we found that the component-by-component (CBC) construction for rank-1 lattice rules received little to no attention with respect to parallelization. The fast CBC algorithm is implemented in the Matlab code by Nuyens [37]. However, even the fast CBC algorithm, with a time complexity of $O(dn\ \log(n))$, can be slow if n is very large. The use of high performance computing in constructing and applying lattice rules for high-dimensional integration can yield more accurate and efficient approximations. This can be beneficial in many scientific fields such high-energy physics, statistics, and modeling in behavioral psychology.

1.6. Thesis Statement

Quasi-Monte Carlo methods can be a better alternative to Monte Carlo methods in dealing with the curse of dimensionality under certain conditions. Furthermore, the ability to use a large number of points can be an important factor in obtaining accurate approximations to high-dimensional integrals. Therefore, the aim of this work is to design and optimize parallel scalable algorithms based on CBC methods to construct and apply good lattice rules.

1.7. Contributions

- A parallel implementation of lattice rule integration on Graphics Processing Units (GPUs).
- Quality and cost comparisons between GPU lattice rule integration and other parallel multivariate integration methods.
- The design and implementation of a parallel fast CBC construction algorithm.
- Implementations of this algorithm on multi- and many-core systems.
- Demonstrations that the algorithm can be incorporated efficiently for a direct ("on the fly") generation of rules in integration software and for real-world applications.
- A parallel construction and application of embedded lattice rules.
- A parallel adaptive integration of high-dimensional integrals in Bayesian statistics.

CHAPTER 2

ParInt: ADAPTIVE TASK PARTITIONING

2.1. Overview

In this chapter, we show our application and analysis of a parallel adaptive integration method for the computation of Bayesian integrals [44]. Approximating multivariate integrals is an important problem in Bayesian inference and statistics in general. Especially in higher dimensions, these problems can be very hard to solve in a timely manner [45] [9] [5].

Using the notations of Evans and Swartz [45], a number of integrals need to be calculated where the integrand is written as a product of a common part $f(\boldsymbol{\theta})$ and a function $w(\boldsymbol{\theta})$. The integral is then of the form

$$\mathcal{I}(w) = \int_{\mathbb{R}^d} w(\boldsymbol{\theta}) f(\boldsymbol{\theta})\, d\boldsymbol{\theta} \tag{2.1}$$

where $w : \mathbb{R}^d \to \mathbb{R}$, $f : \mathbb{R}^d \to \mathbb{R}^+$ represents the product of the likelihood and the prior distribution, and w is a function (such as $w(\boldsymbol{\theta}) = \theta_1$ or $w(\boldsymbol{\theta}) = \theta_1^2$) of which the posterior expectation $R(w) = \frac{\mathcal{I}(w)}{\mathcal{I}(1)}$ is needed. The function fw is assumed integrable over $\mathbb{R}^d$.

We consider two Bayesian applications from, the *Linear Model* and *Contingency Table* problems of *Example 1* and *2*, respectively, for which various methods are compared in [45]: asymptotic techniques, importance sampling and variance reduction, adaptive importance sampling, and multivariate adaptive integration. Adaptive integration was explored by Genz et al. [46]–[49] for problems emerging in Bayesian analysis, and was found successful in combination with transformations that result in smoothing the integrand behavior so that the transformed integrand behavior is similar to that of a low-degree polynomial. Apart from

the dimensionality and the infinite domain, problems arise from the peaked modal behavior of the posterior density function; a nonnormal tail behavior is also treated in [49]. Approximations to the mode are given in the file `postpack.f` of the Fortran BAYESPACK package, which provides the integrands corresponding to 21 example log-posteriors.

it is our goal to show the performance of the PARINT software in a black-box approach, by truncating the infinite domain of (2.1) to a finite region. In this implimentation from [44], we discuss and test the adaptive strategy in ParInt. The performance of the algorithm is verified by detailed test results for Bayesian integrals arising as posterior expectations for cross-classifications in medical data.

2.2. ParInt

An effective tool that uses high performance computing for automatic integration is ParInt. This is a software package developed at Western Michigan University by de Doncker et al. [6] [50] [51] [52]. The system is designed to solve integration problems numerically via high performance computing. ParInt uses various methods including adaptive integration, quasi-Monte Carlo, and Monte Carlo methods for integration over hyper-rectangular and simplex regions. ParInt has had various experimental versions implemented on a variety of platforms. The package is written in C and runs on a UNIX platform using the Message Passing Interface (MPI). Desired integrand functions can be written as C or FORTRAN functions..

The use of parallel processing in ParInt increased the performance of solving high dimensional integrals considerably. This has had a direct effect on many applications. The user defines the integrand function and the domain, and specifies a relative and absolute error tolerance for the computation, and the system will work as a black-box (or automatic) model to integrate the function [53]. The structure of the package is depicted in Fig. 2.1.

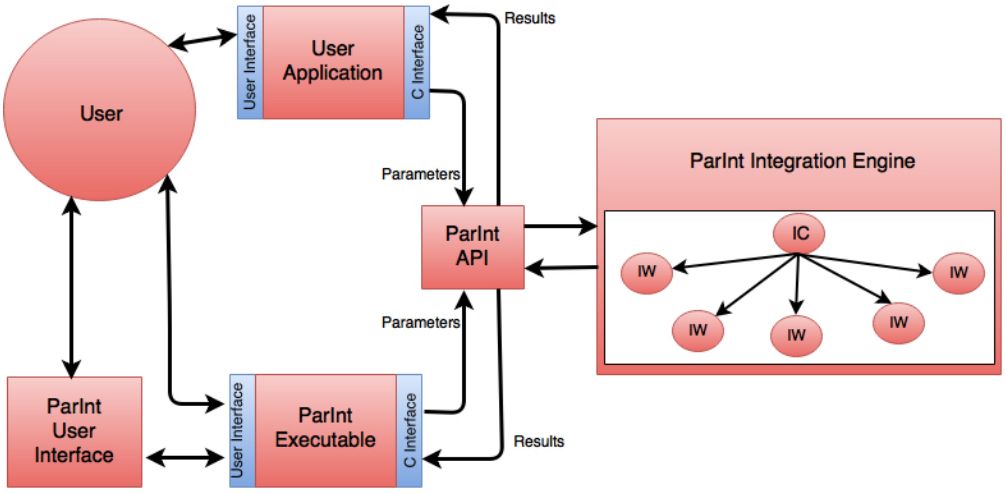


Figure 2.1: ParInt architecture

In [54] the black-box approach is described by a functional that associates the user-defined integrand function $f : \mathbb{R}^d \to \mathbb{R}$, integration domain $\mathcal{D} \subset \mathbb{R}^d$, absolute and relative error tolerances, t_a and t_r, respectively, with an integral approximation Qf and an absolute error estimate or bound $E_a f$ that are intended to satisfy

$$Qf \approx \int_{\mathcal{D}} f(\boldsymbol{x})\, d\boldsymbol{x}, \quad \text{and} \quad |Qf - If| \leq E_a f \leq \tau, \tag{2.2}$$

where the tolerated error $\tau = \max\{t_a, |If|\, t_r\}$ corresponds to the maximum or less strict of the specified absolute and relative tolerances. If an absolute error (only) needs to be achieved, we set $t_r = 0$ (and vice-versa). If $t_a = t_r = 0$, then the program will reach an abnormal termination, typically when the maximum number of function evaluations is reached.

In the black-box approach it is important that no specific problem characteristics are taken into account. Other methods might, for example, absorb certain integrand behavior into a weight function (similar to importance sampling), which should lead to more efficient solutions for the problem class at hand. These techniques would not be viable in applications such as the computation of Feynman loop integrals, where integrand functions may be derived

Algorithm 1: Adaptive partitioning meta-algorithm

```
1 Evaluate initial domain (task) and assign results
2 Initialize priority queue with initial task
3 while task evaluation limit not reached and termination conditions not reached do
4 |   Retrieve task from priority queue
5 |   Split task
6 |   Evaluate new subtasks and update results
7 |   Insert new subtasks into priority queue
8 end
```

Figure 2.2: Adaptive partitioning meta-algorithm

by an automatic generator.

2.3. ParInt Parallel Adaptive Integration

The algorithms and tools in the parallel integration engine include: (i) an adaptive region subdivision algorithm, equipped with a load balancing strategy to handle localized integrand difficulties such as peaks and singularities; (ii) a non-adaptive Quasi-Monte Carlo (QMC) method based on Korobov lattice rules; (iii) Monte Carlo (MC) methods for high dimensions and/or erratic integrands; (iv) a user interface based on the ParInt plugin compiler which pre-processes the user problem specification and integrand function to be linked with the ParInt executable; (v) 1D rules corresponding to those in QuadPack [55], which can be used for repeated integration in successive coordinate directions. The adaptive algorithm applies task partitioning to the division of the domain D into subregions, so that the integration points are concentrated in areas where the integrand behavior is difficult. Each subregion constitutes a task, and the task granularity (amount of work per task) is determined by the number of points used by the integration rules locally over the subregion, and the time required per function evaluation. Fig. 2.2 outlines an adaptive partitioning meta-algorithm.

In the context of numerical integration, the termination conditions can be interpreted in the context of an accuracy requirement of the form (2.2). Initially the integration rules are applied over the entire domain; the results initialize a priority queue keyed by the estimated error of the regions. The available multivariate integration (cubature) rules in ParInt include a set of rules of polynomial degrees 7 and 9 for the d-dimensional cube [25], [56], [57] and of degrees 3, 5, 7 and 9 for the d-dimensional simplex [58]–[60]. The latter are the simplex rules by Grundmann and Möller [60] (which were also found by de Doncker [61] via extrapolation). 1D rules corresponding to those of Dqage and Dqagse in QuadPack [55] include the 7-15, 10-21, 15-31, 20-41, 25-51 and 30-61 point Gauss-Kronrod pairs. Note that an integration rule is said to be of polynomial degree k if it is exact for polynomials up to degree k (and not for all polynomials of degree $k+1$).

The *while* loop iteration in the algorithm consists of: deleting the region with the highest estimated error from the priority queue, bisecting this region, integrating over the two new subregions, updating the overall result and error estimate, and inserting the new subregions into the queue. This process continues until it is estimated that the user's error tolerance (τ in (1)) is achieved, or a user-specified bound on the number of function evaluations is reached. Here τ is estimated by τ_e as $\tau = \max\{t_a, |If|t_r\} \approx \tau_e = \max\{t_a, |Qf|t_r\}$ where Qf is the current global integral estimate.

As depicted in Fig. 2.1 of the distributed, asynchronous adaptive algorithm, all processes act as *integration worker* processes; one process additionally assumes the role of *integration controller*. The initial region is divided among the workers. Each executes the adaptive integration algorithm of Fig. 2.2 on its portion of the domain, while maintaining its local priority queue of regions. The latter is implemented as a max-heap, keyed with the estimated integration errors over the subregions, so that the subregion with the largest estimated error is stored in the root of the heap. If the user specifies a maximum size for the heap structure

on the worker, the task pool is stored as a *heap* or *double-ended heap*, which allows deleting of the maximum as well as the minimum element efficiently (in order to maintain the size of the data structure once it reaches its maximum).

The local region processing is independent of the other workers. All workers periodically send updates of their results to the controller; the worker update contains the differences in the integral and error estimates incurred since the previous update. In turn, the controller provides the workers with updated values of the global estimated tolerated error τ_e. The workers use this value to determine if they have satisfactorily calculated the integral over their portion of the domain. If they have, they become *idle*.

A worker becomes *idle* if the ratio R_E of its total local error to the total tolerated error τ_e drops below its fraction R_V of the total volume (of the original domain D), i.e.,

$$R_E = (\text{total local error})/\tau_e < R_V = (\text{local volume})/\text{volume}(D).$$

The error ratio is defined in [52] as the ratio R_E/R_V. Thus a worker becomes idle when its error ratio falls below 1. In order to keep the load distributed over all the workers and maintain parallel efficiency, work needs to be moved from busy to idle workers.

ParInt employs a *receiver initiated, scheduler based* technique where the controller acts as the scheduler and keeps an IDLE-STATUS list of the workers. The controller is informed of the idle status of the workers via the workers' update messages. When a worker i informs the controller via a regular update message of its non-idle status, the controller selects (in a round-robin fashion) an idle worker j and sends i a message containing the ID of j. Worker i will then send a work message to j containing either new work or an indication that it has no work available. Worker j receives this message and either resumes working or informs the controller that it is still idle.

The load balancing strategy can be tailored to increase the amount of load balancing, e.g., by sending larger amounts of work in individual load balancing steps. The controller

can furthermore select multiple idle workers to be matched up with busy workers within a single load balancing step. The load balancing strategy scales well in many applications, notwithstanding an apparent possible bottleneck at the controller.

2.4. ParInt Applications to Bayesian Statistics

2.4.1. Test 1: Linear model

We consider this application from [45]. The observed response is modeled as

$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \sigma\boldsymbol{z}$, with $\boldsymbol{y} \in \mathbb{R}^{45},\ \boldsymbol{X} \in \mathbb{R}^{45\times 9},\ \boldsymbol{\beta} \in \mathbb{R}^{9},\ \sigma \in (0,\infty),\ \boldsymbol{z} \in \mathbb{R}^{45}$, for simulated data given in [45].

The function $f(\boldsymbol{\theta})$ in the integral (2.1) is specified in $d = 10$ dimensions as a function of the integration variables $\theta_i = \beta_i,\ 1 \le i \le 9,$ and $\theta_{10} = \log\sigma$,

$$f(\boldsymbol{\theta}) = e^{-9\,n\,\theta_{10}} \prod_{i=1}^{9}\prod_{i=1}^{n} g_\lambda(\frac{y_{ij}-\theta_i}{e^{\theta_{10}}}) \tag{2.3}$$

with

$$g_\lambda(v) = \frac{\Gamma(\frac{\lambda+2}{2})}{\Gamma(\frac{1}{2})\Gamma(\frac{\lambda}{2})}\left(1+\frac{v^2}{\lambda-2}\right)^{-\frac{\lambda+1}{2}}\frac{1}{\sqrt{\lambda-2}}$$

for $\lambda = 3,\ n = 5$.

Table 2.1 gives results obtained with the ParInt adaptive algorithm for the integral (2.1) of fw with $f = f(\boldsymbol{\theta})$ in (2.3) and $w = w(\boldsymbol{\theta}) = \theta_1, \theta_2, \theta_4, \theta_{10}, \theta_1^2, \theta_2^2, \theta_4^2, \theta_{10}^2$ (in successive rows of the table).

For the integration with the adaptive ParInt algorithm, the integration domain $\mathbb{R}^{10}$ is truncated to the 10D rectangular region with half-width = 5 and centered at the mode, $\hat{\theta}$. The program is run for a relative error tolerance of 10^{-12}, with the absolute error tolerance set to 0 (in view of the small values of the results, of orders ranging between 10^{-24} and

Table 2.1: ParInt adaptive integration results for *Example 1* over 10D truncated domain, centered at mode with half-width = 5, using 16 (Time only), 64 or 128 MPI processes, compared to results from Evans and Swartz (E&S) [45]. Our results list: integral approximation (Res.), absolute error estimate (E_a) and execution time in seconds for parallel runs, allowing $2B$ using 64 procs., $5B$ and $25B$ evaluations using 128 procs.

	2B, 16 procs.	2B, 64 procs.		5B, 128 procs.			25B, 128 procs.			E&S	E&S
										$\hat{R}$	R
w	Time (s)	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Subr. Ad.	Exact
1	373.0	1.317 $\times 10^{-22}$	0.036 $\times 10^{-22}$	95.7	1.310 $\times 10^{-22}$	0.016 $\times 10^{-22}$	120.4	1.3065 $\times 10^{-22}$	0.0027 $\times 10^{-22}$	592.3		1.31 $\times 10^{-22}$
θ_1	372.4	2.043	0.055	95.7	2.043	0.025	120.3	2.0428	0.0042	591.9	2.040	2.043
θ_2	371.0	0.0957	0.0038	95.2	0.0953	0.0017	120.3	0.09473	0.00034	591.5	0.094	0.095
θ_4	373.5	0.0170	0.0039	95.4	0.0179	0.0020	120.5	0.01801	0.00033	592.0	0.017	0.018
θ_{10}	373.0	-0.0738	0.0084	95.8	-0.0732	0.0043	120.6	-0.07290	0.00065	593.7	-0.102	-0.073
θ_1^2	372.5	4.261	0.112	95.7	4.264	0.052	120.6	4.2636	0.0088	593.4	4.237	4.263
θ_2^2	372.6	0.0788	0.0013	95.8	0.0795	0.0007	120.6	0.08074	0.00016	593.3	0.068	0.081
θ_4^2	372.5	0.0677	0.0010	95.7	0.0683	0.0005	120.2	0.06906	0.00011	592.5	0.055	0.069
θ_{10}^2	374.8	0.0320	0.0016	96.1	0.0325	0.0006	120.5	0.03269	0.00012	593.5	0.032	0.033

10^{-22}). This error tolerance is set so that the program would terminate when the allowed number of function evaluations is reached. For the integration, the rule of polynomial degree 9 is used for the basic rule integration over the subregions.

The integral approximation, absolute error estimate and execution time are given in Table 2.1 for runs with function evaluation bounds of $2B$ (Billion) using 64 MPI processes, and with $5B$ and $25B$ evaluations using 128 processes a cluster. We used 16 processes per 16-core cluster node with Intel Xeon E5-2670, 2.6 GHz dual processors and 128 GB of memory, and the cluster's Infiniband interconnect for message passing via MPI. The integral approximation and absolute error estimate for the integral $\mathcal{I}(w)$ of (2.1) are scaled by the value for $\mathcal{I}(1)$ (in the rows for $w \neq 1$ of Table 2.1).

The integral approximations can be compared to the values ($R(\theta_i)$ and $R(\theta_i^2)$, $i = 1, 2, 4, 10$) from Evans and Swartz [45] (cf., their Tables 2, 3, 4, 5 and 6), listed as *Exact* in the last column of Table 2.1. The preceding column of Table 2.1 gives the estimates $\hat{R}(\theta_i)$

and $\hat{R}(\theta_i^2),\ i = 1, 2, 4, 10$ from Evans and Swartz (their Table 5 p. 265), obtained with a sub-region adaptive procedure. According to [46], [49] they perform a transformation $\boldsymbol{\theta} = \hat{\boldsymbol{\theta}} + C\boldsymbol{u}$, where C is the lower-triangular Cholesky factor of the inverse Hessian matrix H of a function h which satisfies $f(\boldsymbol{\theta}) = e^{-\tau h(\boldsymbol{\theta})}$ in (2.1). This is followed by the transformation $\boldsymbol{v} = \Phi(\boldsymbol{u})$ where Φ denotes the $N(0, 1)$ distribution function applied componentswise to u.

2.4.2. Test 2: Contingency table

This application from [45] analyzes a cross-classification of 132 long-term schizophrenic patients into a table with three rows and three columns, with respect to the frequency of hospital visits and length of stay, respectively. The cell probabilities are of the form

$$p_{ij} = \theta\, \alpha_i(1)\beta_j(1) + (1 - \theta)\, \alpha_i(2)\beta_j(2)$$

and the likelihood function is $\prod_{i=1}^{3} \prod_{j=1}^{3} p_{ij}^{f_{ij}}$, where f_{ij} is the (i, j) cell count in the table. For the prior distribution,
$\theta \sim U(0, \frac{1}{2})$, $(\alpha_1(k), \alpha_2(k), \alpha_3(k)) \sim$ Dirichlet(1,1,1), $(\beta_1(k), \beta_2(k), \beta_3(k)) \sim$ Dirichlet(1,1,1), $k = 1, 2$, assuming independence of the rows and columns.

We further use the transformation specified in the BayesPack [48] implementation of the integrand function,

$$\theta = e^{x_1}/(2 + 2e^{x_1})$$

$$\alpha_{11} = e^{x_2}/(1 + e^{x_2} + e^{x_3}),\ \alpha_{21} = e^{x_3}/(1 + e^{x_2} + e^{x_3})$$

$$\alpha_{12} = e^{x_4}/(1 + e^{x_4} + e^{x_5}),\ \alpha_{22} = e^{x_5}/(1 + e^{x_4} + e^{x_5})$$

$$\beta_{11} = e^{x_6}/(1 + e^{x_6} + e^{x_8}),\ \beta_{21} = e^{x_7}/(1 + e^{x_6} + e^{x_8})$$

$$\beta_{12} = e^{x_8}/(1 + e^{x_8} + e^{x_9}),\ \beta_{22} = e^{x_9}/(1 + e^{x_8} + e^{x_9})$$

Table 2.2: PARINT adaptive integration results for *Example 2* over 9D truncated domain, centered at mode with half-width = 8; results include: integral approximation (RES.), absolute error estimate (E_a) and execution time in seconds for parallel runs, allowing $10B$ using 16 procs. (Time only), $10B$ using 32 procs., and $25B$ evaluations using 64 procs.

	10B, 16 PROCS.	10B, 32 PROCS.			25B, 64 PROCS.		
w	TIME (S)	RES.	E_a	TIME (S)	RES.	E_a	TIME (S)
1	1138.5	4.862 $\times 10^{-121}$	0.018 $\times 10^{-121}$	545.3	4.864 $\times 10^{-121}$	0.012 $\times 10^{-121}$	680.6
θ	1138.2	0.4218	0.0016	544.7	0.4217	0.0009	680.5
θ^2	1138.2	0.1807	0.0007	544.5	0.1808	0.0004	679.8

Table 2.2 gives PARINT results for these problems (integral approximation and absolute error estimate scaled with the value for $\mathcal{I}(1)$, and time in seconds).

In this example, the posterior expectations $R(\theta) = 0.422$ and $R(\theta^2) = 0.181$ were obtained as *Exact* values in [45] using importance sampling and variance reduction techniques with a run of 41 hours CPU time (in 1995). They use this as a basis for comparison of their results obtained with various methods. Their subregion adaptive estimates requiring the inverse Hessian matrix and normal probability transforms yield $\hat{R}(\theta) \approx 0.419$ and $\hat{R}(\theta^2) \approx 0.178$ with relative error estimates 0.007 and 0.017, respectively, in 3.6 hours of CPU time.

In Table 2.2 the Exact value is reached (within the given error estimate) using 10B evaluations and 32 MPI processes in less than 10 minutes, and with 25B evaluations and 64 processes in around 11 minutes. By comparing the times for 10B evaluations with 16 and 32 processes, the speedup S_p is slightly larger than 2 (superlinear). Superlinear speedups can sometimes be obtained if the base time is somewhat large e.g., due to system loads, or as a result of memory or caching effects in the executions. In conclusion, the parallel PARINT computations allow for accurate and efficient solutions for this proplem, while offering the

convenience of a black-box numerical computation for the multivariate integration. Results of this and another Baysian application are reported in [44].

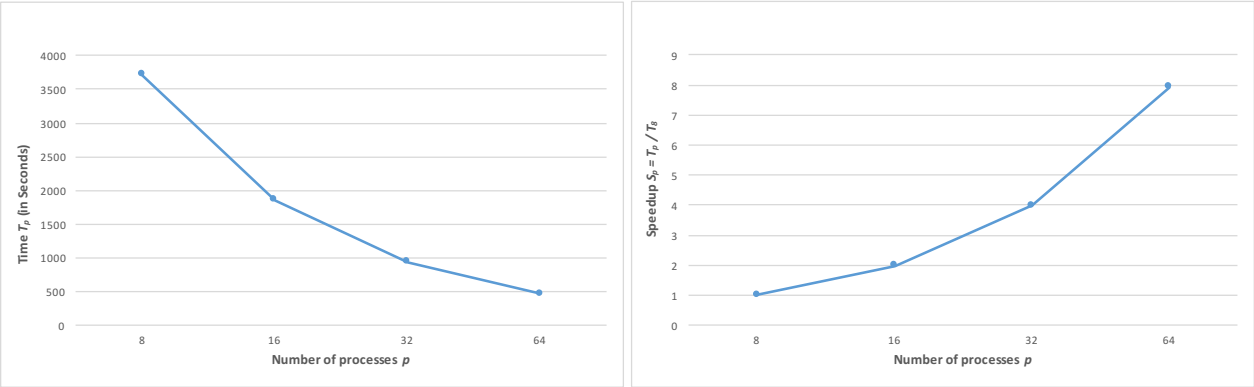
2.4.3. Parallel Performance

For *Examples 1*, Evans and Swartz stated in [45] that their computation of $\hat{R}$ allowing 2×10^7 evaluations did not achieve the relative error requirement of 0.001 and took 2.75 hours (sequentially). Repeating with a 10^8 evaluation limit improved their results slightly but ran for 24 hours.

In comparison, Table 2.1 shows that our results are good with respect to accuracy and time. With 2B evaluations the errors are over-estimated, but this improves considerably for larger numbers of evaluations. Using 25B function evaluation on 128 processes, we have achieved the *Exact* value for every w (with rounding up) in less than 10 minutes for each calculation.

Fig. 2.3 gives timing results for calculating θ_1 from *Examples 1* using 10B evaluations. Fig. 2.3 (a) plots the parallel time T_p for the evaluations in seconds, and Fig. 2.3 (b) gives the speedup as a function of the number of MPI processes p, for $8 \leq p \leq 64$.

In *Examples 2*, the posterior expectations $R(\theta) = 0.422$ and $R(\theta^2) = 0.181$ were obtained as *Exact* values in [45] using importance sampling and variance reduction techniques with a run of 41 hours CPU time. Their subregion adaptive estimates requiring the inverse Hessian matrix and normal probability transforms as for *Example 1* yield $\hat{R}(\theta) \approx 0.419$ and $\hat{R}(\theta^2) \approx 0.178$ with relative error estimates 0.007 and 0.017, respectively, in 3.6 hours of CPU time.



(a) (b)

Figure 2.3: Parallel performance of integral computations for θ_1 from *Examples 1* using 10B evaluations: (a) Computation times (in seconds), (b) Speedup

Our results in Table 2.2 was good as well, and we reached the *Exact* value (with rounding up) using 25B evaluations on 64 MPI processes in around 11 minutes. The speedup S_p can be noticed in the 10B function evaluations, where doubling the number of processes yield $S_p \approx 2$

Thus the parallel ParInt computations allow for accurate and efficient solutions for *Examples 1* and *2*, while offering the convenience of a black-box numerical computation for multivariate integration.

CHAPTER 3

LATTICE RULE CONSTRUCTION

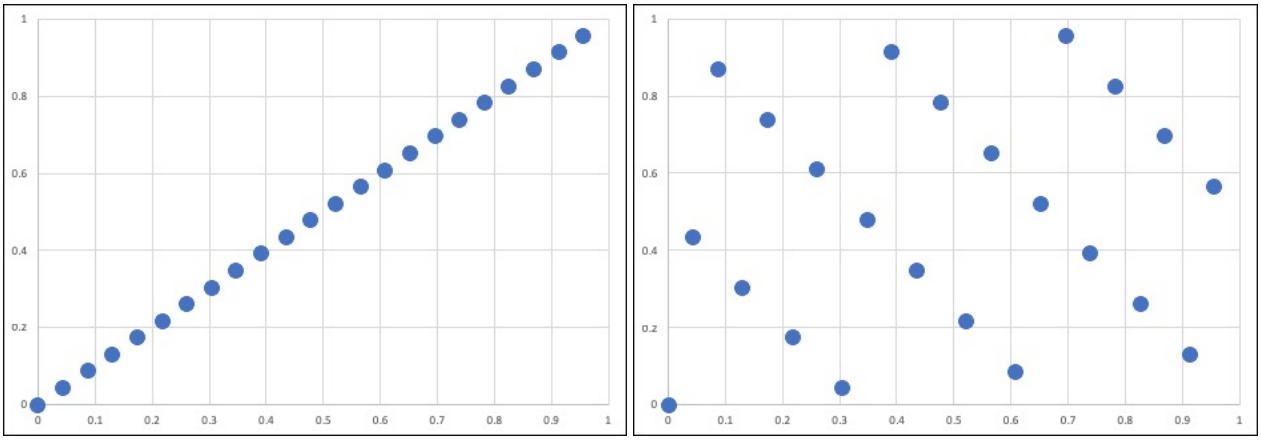
3.1. Overview

Lattice rules are quasi-Monte Carlo rules that can be used for the integration of periodic functions over the d-dimensional unit cube [62]. A "good" lattice rule generator vector is used with the goal of reducing the integration error. This concept has been around since the 1960s (e.g. [63]). However, finding a good lattice rule was typically done with a very extensive search, where the search time grows exponentially with increasing dimension [62]. Sloan and Reztsov changed that by introducing the component-by-component construction algorithm in 2002. The goal of the component-by-component construction algorithm is to search for good lattice rules that minimize the worst-case error in a reproducing kernel Hilbert space [36].

A reproducing kernel Hilbert space is a Korobov space where the evaluation of a function may be written as an inner product with the reproducing kernel K. The utilization of reproducing kernel Hilbert spaces has provided a remarkably rich principle, where all sorts of discrepancies can be recognized in terms of the worst-case error in a particular space. Furthermore, it allows the analysis of the error in nonperiodic spaces [22].

3.2. Good Lattice Rules

A good lattice rule construction requires a good generating vector. Fig. 3.1 shows a lattice rules constructed using a bad generating vector and a good lattice rule using a generating vector that produces a more evenly spread point set.



(a) (b)

Figure 3.1: Two $2d$ lattice points with $n = 23$. (a) A bad lattice rule constructed from generating vector $\mathbf{z} = (1, 1)^t$. (b) A good lattice rule constructed from generating vector $\mathbf{z} = (1, 10)^t$.

The role of any lattice rule construction algorithm is to find a good generator $\mathbf{z} \in \mathbb{Z}^d$ that can be used to construct a good integration lattice. According to Korobov [64], for N prime, the vector $\mathbf{z}$ has optimal coefficients if

$$|Q(\mathbf{z}, N)f - If| \leqslant c\, d(s, \alpha)\, \frac{(\log N)^{\beta(s,\alpha)}}{N^{\alpha}} \tag{3.1}$$

where $f \in E_\alpha(c)$, and $d(s, \alpha)$ and $\beta(s, \alpha)$ are independent of N. Korobov [64] also proved that good lattice points are exist if N is a prime.

The integration error is an important factor in determining if one lattice rule is better than another. The worst-case error for a cubature rule Q to approximate integral I of functions in the unit ball of a Banach space $\mathcal{F}$ is defined as [65]

$$e(Q, \mathcal{F}) = \sup |I(f) - Q(f)| \tag{3.2}$$

where the supremum is taken over the unit ball, $f \in \mathcal{F}$ and $||f||_{\mathcal{F}} \leq 1$. To apply this definition to the construction of lattice rules, it is assumed that $\mathcal{F}$ is a reproducing kernel Hilbert space ($\mathcal{H}$) with kernel $K(\dot{;}\dot{)}$, so that $f(x) = \langle f, K(\dot{;}x)\rangle_{\mathcal{H}(K)}$. This reproducing kernel is usually a function of two variables, but when the function space is periodic, the kernel can actually be written as a function of one variable. Such a kernel is called *shift-invariant* [22]. In this case, for all $\mathbf{x}, \mathbf{y} \in [0,1)^d$,

$$K(\mathbf{x}, \mathbf{y}) = K(\{\mathbf{x} - \mathbf{y}\}, 0) =: K(\{\mathbf{x} - \mathbf{y}\})$$

Finally, we take Q to be a lattice rule such that $\forall \mathbf{x}, \mathbf{y} \in P_n : \{\mathbf{x} - \mathbf{y}\} \in P_n$ where P_n is a point set of size n over $[0,1]^d$.

The squared worst-case error for a shift-invariant kernel K and a rank-1 lattice is given by [66]

$$e^2(Q, K) = -\int_{[0,1)^d} K(\boldsymbol{x}, 0)\, d\boldsymbol{x} + \frac{1}{n}\sum_{j=1}^{n} K\left(\left\{\frac{j\mathbf{z}}{n}\right\}, 0\right) \tag{3.3}$$

Thus, an exact formula for the worst-case error in the desired space is given via the reproducing kernel.

Sloan and Woźniakowski introduced the *weighted* classes functions [67], a significant addition to the theory of reproducing kernel Hilbert spaces. The weights $\gamma_k \geq 0$ are used to indicate the significance of particular sets of variables. The kernel for a shift-invariant and tensor-product weighted reproducing kernel Hilbert space can be written as [22]

$$K(\mathbf{x}, \mathbf{y}) = \prod_{k=1}^{d} (1 + \gamma_k\, \omega(x_k - y_k)) \tag{3.4}$$

In this case, the weights γ_k are used to express the importance of each dimension. For a

rank-1 lattice rule using the weighted space, the squared worst-case error can be expressed as [22]

$$e_d^2(\mathbf{z}) = -1 + \frac{1}{n}\sum_{j=1}^{n}\prod_{k=1}^{d}\left(1 + \gamma_k\ \omega\left(\left\{\frac{jz_k}{n}\right\}\right)\right) \tag{3.5}$$

assuming that

$$\int_0^1 \omega(x)\ dx = 0$$

3.3. Component-by-Component Construction

The CBC algorithm searches and then fixes each component of the generating vector $\boldsymbol{z}$, one component at a time. Given that the first component is 1, the second component can be calculated by: z_1 fixed, choose $z_2 \in \{1 \leq z \leq n-1 \mid \gcd(z,\ n) = 1\}$ to minimize the worst-case error in 2 dimensions. This step is iterated through determining z_d.

Algorithm 2 outlines the CBC algorithm. The γ_k and β_k are components of parameter vectors for weighting per dimension, and U_n is the multiplicative group modulo n defined as: $U_n = \{r \in Z_n : gcd(r, n) = 1\}$. Since it is assumed that n is prime, $U_n = \{1, 2, \ldots, n-1\}$.

Algorithm 2: Component-by-component construction of good lattice rules algorithm

```
for d ← 1 to d_max do
    for all z_d ∈ U_n do
        e_d^2(z_d) = −∏_{k=1}^{d} β_k + (1/n) Σ_{j=1}^{n} ∏_{k=1}^{d} ( β_k + γ_k ω( {j z_k / n} ) )
    end
    z_d =_{z ∈ U_n}  e_d^2(z)
end
```

Figure 3.2: CBC algorithm

The cost of constructing a lattice rule using the CBC algorithm is $O(dn^2)$ where d is the dimension and n is the number of points. The CBC method allows for larger values of d and

n than the older search methods. However, given that it has a quadratic time complexity in n, using very large values is still not feasible [22].

3.4. Fast Component-by-Component Construction

The main step in the CBC algorithm involves calculating the squared worst-case error, which is done in [65] based on

$$e_d^2(\mathbf{z}) = -\prod_{k=1}^{d} \beta_k + \frac{1}{n} \sum_{j=0}^{n-1} \prod_{k=1}^{d} \left(\beta_k + \gamma_k \, \omega \left(\left\{ \frac{j z_k}{n} \right\} \right) \right) \tag{3.6}$$

This step is performed in an iterative process over the dimension.In iteration d, the product defined in the first $d-1$ iterations is fixed. Therefore, Eq. (3.6) can be rewritten as

$$e_d^2(\mathbf{z}) = -\prod_{k=1}^{d} \beta_k + \frac{1}{n} \sum_{j=0}^{n-1} p_{d-1}(j) \left(\beta_d + \gamma_d \, \omega \left(\left\{ \frac{j z_d}{n} \right\} \right) \right) \tag{3.7}$$

where $p_{d-1}(j)$ is a product of $d-1$ factors, and function w is evaluated in n points. The $O(n^2)$ part of the CBC complexity is the result of a matrix-vector multiplication, as can be seen in [65] (Eq. 3.6-7) using the matrix

$$\mathbf{\Omega}_n := \left[\omega \left(\frac{j.z \ mod \ n}{n} \right) \right]_{z=1,\ldots,n-1.\ j=0,\ldots,n-1} \tag{3.8}$$

and the product vector $p_{d-1}(j)$. The worst-case vector can be written in the form:

$$\mathbf{e}_d^2 = -\prod_{k=1}^{d} \beta_k + \frac{1}{n} \left(\beta_d \mathbf{1}_{n-1 \times n} + \gamma_d \, \mathbf{\Omega}_n \right) \mathbf{p}_{d-1} \tag{3.9}$$

Nuyens and Cools [23] proved that the CBC algorithm using the matrix-vector multiplication with the structure in Eq. (3.9) can be performed in $O(dn \log(n))$ time using the

Fast Fourier Transform (FFT). In order to apply FFT, the matrix $\mathbf{\Omega}_n$ must be in a circulant form.

A circulant matrix $\mathbf{C}$ of order $n \times n$ is a Toeplitz matrix of the form

$$\mathbf{C} = \begin{pmatrix} c_0 & c_{n-1} & c_{n-2} & \dots & c_1 \\ c_1 & c_0 & c_{n-1} & \dots & c_2 \\ c_2 & c_1 & c_0 & \dots & c_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n-1} & c_{n-2} & c_{n-3} & \dots & c_0 \end{pmatrix}$$

where each row is derived from the previous row by a circular shift. Note that, by applying FFT, a circulant matrix system can be solved in $O(n \log(n))$ operations [68].

3.4.1. Fast Fourier Transform

Given a vector $\mathbf{x}$ and a circulant matrix $\mathbf{C}_m$, a matrix vector multiplication can be done using FFT by

$$\mathbf{C}_m \mathbf{x} = \mathbf{F}_m^{-1} \operatorname{diag}(\mathbf{F}_m \, \mathbf{c}) \, \mathbf{F}_m \, \mathbf{x} \tag{3.10}$$

where $\mathbf{F}_m$ is the Fourier matrix and $\mathbf{c}$ is the first column of $\mathbf{C}_m$. The multiplication is performed by calculating the inverse FFT ($\mathbf{F}_m^{-1}$) of the discrete Fourier transform of $\mathbf{c}$. Therefore, eq. 3.10 can be rewritten as [69]

$$\mathbf{C}_m \, \mathbf{x} = IFFT(\operatorname{diag}(FFT(\mathbf{c})) \, FFT(\mathbf{x})) \tag{3.11}$$

For the fast CBC algorithm, the number of points has to be prime. Therefore, the rows of $\mathbf{\Omega}_n$ can be permuted by the positive powers of some generator g (where g is the primitive root of n) and the columns by the negative powers of g [69]. That will make $\mathbf{\Omega}_n$ a circulant

matrix with diagonals of constant powers of g. The structure of $\mathbf{\Omega}_n$ will be of the form

$$\mathbf{\Omega}_n = \begin{pmatrix} 1 & g^{-1} & g^{-2} & \dots & g \\ g & 1 & g^{-1} & \dots & g^2 \\ g^2 & g & 1 & \dots & g^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g^{-1} & g^{-2} & g^{-3} & \dots & 1 \end{pmatrix}$$

Applying the transforms in Eq. (3.10) will result in the fast CBC algorithm. Each FFT and IFFT is performed in time of order $O(n \log(n))$.

CHAPTER 4

MONTE CARLO AND LATTICE RULES INTEGRATION ON GPUs

4.1. Overview

Our main interest in this chapter is the accuracy and efficiency of multivariate integration using lattice rules. For comparison purposes, we also show results obtained with Monte Carlo integration, implemented using two different random number generators, NVIDIA cuRAND and Random123. The accuracy is assessed for different categories of integrand functions such as: continuous with discontinuous partial derivatives in the interior or at the boundaries of the integration domain, or with a boundary singularity.

In the following sections we describe our implementation of the lattice rule integration on Graphics Processing Units (GPUs) using CUDA C. We further compare its performance to results we obtained using Monte Carlo on GPU. Section 4.2 below describes our computation environment and implementation of the lattice rule integration in CUDA C; Section 4.3 presents numerical and performance results [70]; and Section 4.4 presents our high-speed evaluation of Feynman loop integrals using lattice rules on GPUs with transformations to alleviate the effects of integrand boundary singularities [71] [72] [73].

4.2. Computing Environment

For our experiments we used the *thor* high performance parallel compute cluster in the Center for High Performance Computational Science and Big Data in the College of Engineering and Applied Sciences at Western Michigan University. The *thor* cluster has 22 compute nodes, 11 of which each have one Nvidia Kepler series K20m GPU accelerator card,

and one node (n7) has four K20m GPUs. Nodes 2-21 have dual 8-core Sandy-Bridge Xeon processors for a total of 16 cores per node, 128 GB of RAM, and a 1 Terabyte local hard drive. Nodes 0 and 1 have four 8-core Sandy-Bridge Xeon processors, for a total of 32 cores per node. Node 1 has 256 GB of RAM, and node 0 has 512 GB of RAM.

Binding the entire cluster together is an ethernet management network and a low-latency Infiniband communication network. All nodes have access to network mounted directories on a high performance, high redundancy RAID file server. For user tasks, the *thor* head node is used to manage the entire cluster and share programs and data.

For our CUDA programs, we used an NVIDIA Tesla K20m [74] GPU with 2496 CUDA cores, 706 MHz GPU clock cycle and 4800 MB of global memory. The device is part of a cluster node with dual Intel Xeon E5-2670, 2.6 GHz CPUs and 128 GB of memory. For the results described in Section 4.3, the number of parallel blocks, and threads per block launched in the GPU runs were set to 64 and 1024, respectively,

In the present work we use the fast CBC construction Matlab code [37] by Nuyens to obtain the generator vectors $\mathbf{z}$ for the lattice rules to be used in our experiments. Table 4.1 shows generator vectors $\mathbf{z}$ for 10 dimensions and various numbers of points n. Approximate values of n are listed; the actual values used are (primes): 1009, 100003, 1000003, 100000007, 350000041. More obtained rules can be found in Appendix A.

4.3. Monte Carlo and Lattice Rule Integration on GPUs

To compare accuracy and performance on the GPU, we evaluated a set of integrals using various methods in [70]. We used four test integrals of which the results can be found in the literature or calculated analytically. Apart from the lattice rules on GPU, we

Table 4.1: Generator vectors $\mathbf{z}$ for 10 dimensions and various numbers of points n.

Number of Points (n)				
10^3	10^5	10^6	10^8	35×10^8
1	1	1	1	1
282	42240	292962	41883906	146940205
381	34871	229698	22682973	127904721
428	12695	326198	44229424	108579162
79	19528	246988	29466837	30890237
320	13709	447010	8176047	166719091
171	48806	170157	49462874	28591254
356	1721	104406	1162485	114681466
130	38753	145823	46871525	54322685
266	46060	425870	36107330	136274288

also implemented the lattice rule integration sequentially to measure the speedup. We pre-generated the lattice rules for the parallel and sequential integration. In order to compare the accuracy, we computed the same integrals using lattice rules and Monte Carlo methods. Since the pseudo-random number generator is a major part of the Monte Carlo integration, we considered both cuRAND (in single and double precision) and Random123 (using single precision random number generation and double precision accumulation). To enhance the performance and accuracy of Monte Carlo with cuRAND in single precision we applied Kahan Summation [75].

We use CUDA C v.6.5 for our test programs, whereas the sequential lattice rule program is written in C and compiled with gcc. The integrand function is implemented as a device function that is invoked by the CUDA kernel, which calculates the lattice points. Fig. 4.1 shows a section of the CUDA kernel for computing the lattice points and calling the integrand function. This code implements the formula of Eq. (1.5). Here `tid` is the ID of the thread executing the code on the GPU. The `temp` variable in this code section accumulates the thread's function values. The `temp` values will be accumulated over the threads further

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
double temp = 0;
double temp1, temp2, intpart;
double x[dim];
int i;
while (tid < n) {
   for(i = 0; i < dim; i++) {
   temp1 = (double) tid * z[i];
   temp2 = temp1 / (double) n;
   x[i] = modf(temp2, &intpart);
}
temp += fcn(x);
tid += blockDim.x * gridDim.x;
}
```

Figure 4.1: Section of CUDA kernel code for calculating the sample points and invoking the integrand function `fcn(x)`.

in the kernel (not shown in Fig 4.1) using a synchronization of the threads similar to the reduction for a dot product [76].

For our comparisons with Monte Carlo we used two different pseudo-random number generators, *cuRAND* (of the CUDA toolkit) and *Random123* [77].

The cuRAND library for generating pseudo-random numbers belongs to the CUDA Toolkit [78]. cuRand is able to generate streams of pseudo-random numbers from the host on the device. Yhis eliminates the overhead of moving the sequence from the host memory to the GPU memory. On the other hand, the limited memory space on the GPU may prevent generating a large enough sequence of pseudo-random numbers that may be needed to achieve good accuracy with Monte Carlo. In our tests with cuRAND in double precision we made consecutive kernel calls while adding sequence points in the case of $n = 10^8$.

Random123 is a library of counter-based random number generators (CBRNGs), which can be invoked on CPU and GPU. Random123 consumes much less memory than cuRAND and generates the numbers as needed on the GPU without requiring any processing on the CPU [77]. The use of Random123 instead of cuRAND will make the application faster due

to the dynamic nature of Random123 [14].

The next subsections will highlight the results we obtained for a set of multivariate integrals by applying the aforementioned methods. The problems involve various types of non-smooth integrand functions, which makes them challenging for lattice methods.

4.3.1. Test 1: Continuous integrand with discontinuous partial derivatives

The first test integral (from [19]) is

$$\frac{1}{d}\int_0^1 \dots \int_0^1 \sum_{i=1}^{d} |4x_i - 2| \; dx_1 \dots dx_d = 1 \tag{4.1}$$

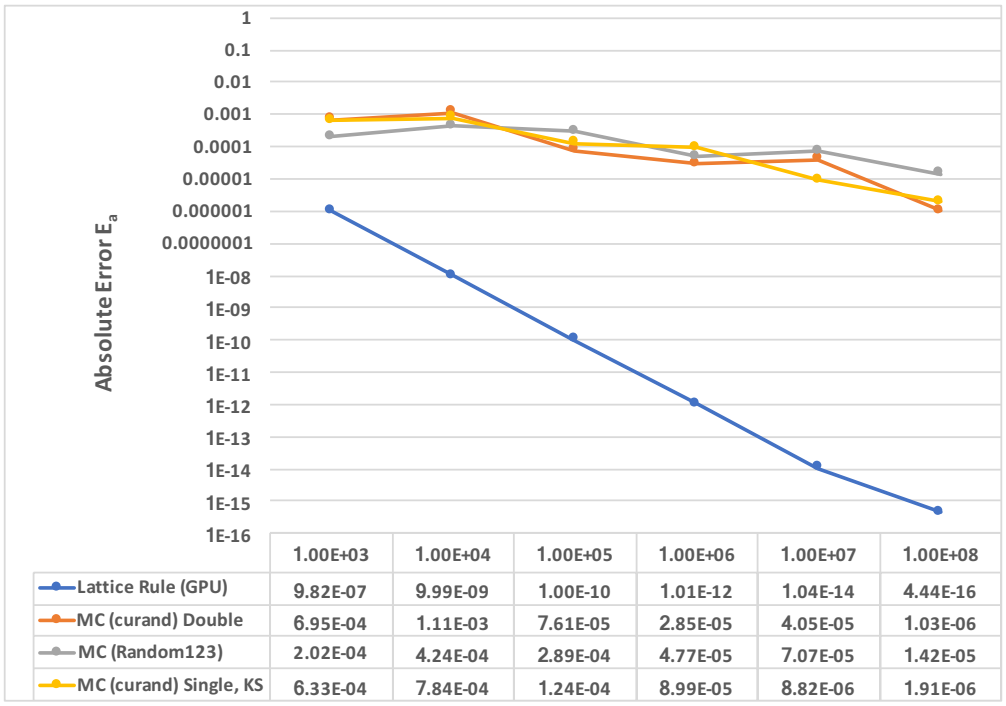
where the integrand function has partial derivative discontinuities, and we choose the number of dimensions $d = 20$. We do not apply a periodization as $|4x_i - 2| = 2$ for $x_i = 0$ and for $x_i = 1,\ 1 \leq i \leq d$. Numerical and performance results using lattice rules and Monte Carlo methods are given in Fig. 4.2, Table 4.2, Fig. 4.3 and Fig. 4.4.

For each n, Fig. 4.2 compares the accuracy of the four (GPU) methods based on the absolute error

$$E_a = |I(f) - Q_n(f)| \tag{4.2}$$

(which in this case is also the relative error as $I(f) = 1$). The results show that the lattice rules yield better accuracy by far compared to the other methods for this integration. The lattice rules have the best accuracy at $n = 10^8$ (with $E_a \approx$ 4.4e-16). Monte Carlo using Random123 is least accurate with error around 1.4e-05 at $n = 10^8$.

Table 4.2 shows the absolute error and time in milliseconds of the lattice rule integration in parallel (LR GPU) and sequentially (LR Seq) for each value of n. The speedup $S = T_{seq}/T_{par}$ is the ratio of sequential to parallel time, and the highest speedup ($S = 239$) is obtained at $n = 10^7$.



	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08
Lattice Rule (GPU)	9.82E-07	9.99E-09	1.00E-10	1.01E-12	1.04E-14	4.44E-16
MC (curand) Double	6.95E-04	1.11E-03	7.61E-05	2.85E-05	4.05E-05	1.03E-06
MC (Random123)	2.02E-04	4.24E-04	2.89E-04	4.77E-05	7.07E-05	1.42E-05
MC (curand) Single, KS	6.33E-04	7.84E-04	1.24E-04	8.99E-05	8.82E-06	1.91E-06

Figure 4.2: Absolute error E_a at each value of n for *Test 1*.

Further regarding the parallel performance, as shown in Fig. 4.3, Monte Carlo with Random123 generates its 10^8 single precision points and processes them in around 92.5 ms, which is the best time among the methods tested. For the same value of n, lattice rules achieve the approximation in 259 ms, while Monte Carlo with (cuRAND single) and (cuRAND double) take around 610 ms and 899 ms, respectively. Fig. 4.4 depicts the obtained speedup S vs. the number of points n.

4.3.2. Test 2: Increasing dimensions

We use the same integral as in Eq. (4.1) but set $d = 25$. The absolute errors of these results do not change much. Fig. 4.5 shows the absolute error for the GPU methods at each value of n. The figure illustrates the difference in accuracy between lattice rules and Mont Carlo for this type of function. We note that the lattice rules achieve the best accuracy at

Table 4.2: Numerical results for *Test 1*, Integral (4.1) with $d = 20$ using lattice rules in parallel (LR(GPU)), and sequentially LR(SEQ)). The results list: the number of points (n), the absolute error (E_a), the time in milliseconds for both methods, and the speedup.

	LR (GPU)		LR (SEQ.)		
n	E_a	TIME (MS)	E_a	TIME (MS)	SPEEDUP
10^3	9.82e-07	2.9e-01	9.82e-07	5.9e-01	2.0e+00
10^4	9.98e-09	3.0e-01	9.98e-09	5.8e+00	1.9e+01
10^5	9.99e-11	6.0e-01	1.00e-10	5.8e+01	9.6e+01
10^6	1.01e-12	2.9e+00	1.03e-12	6.1e+02	2.07e+02
10^7	1.04e-14	2.5e+01	2.50e-14	6.1e+03	2.39e+02
10^8	4.44e-16	2.5e+02	5.01e-13	5.8e+04	2.26e+02

$n = 10^7$; the error value increases at $n = 10^8$ (which did not happen for *Test 1* with $d = 20$). This may be due to roundoff error. On the other hand, the figure clearly shows the slow convergence of the Monte Carlo methods.

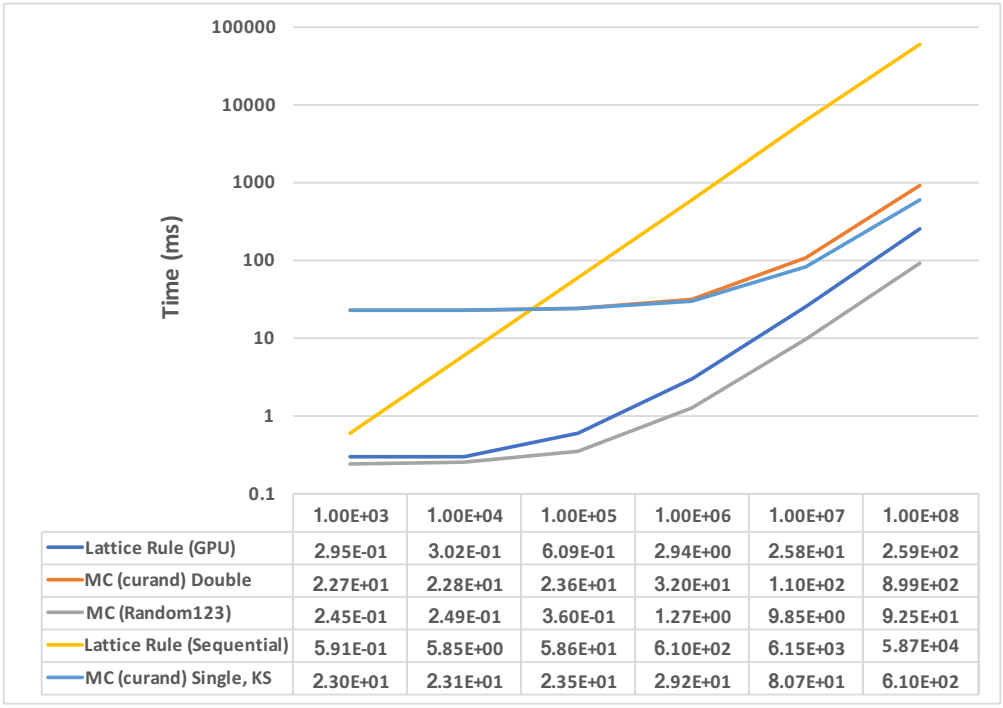
The increase in dimension also changes the time of the integration methods. Fig. 4.6 plots the computation time vs. n. It is expected for the execution time to increase with increasing dimension. Specifically, apart from the increased function evaluation time, the length of the random sequence generated for Monte Carlo is dn. The lattice rule method on GPU achieves computation time (345 ms) similar to Monte Carlo with Random123 (311 ms).

4.3.3. Test 3: Singularity of second order partial derivatives

We use a 10-dimensional integral from [19] of the form

$$\int_0^1 \dots \int_0^1 \left(\sum_{i=1}^{10} x_i \right)^{1.5} dx_1 \dots dx_{10} = 11.32097423155 \tag{4.3}$$

The theory of good lattice points relies on properties of periodic functions and supplies error bounds for functions that are periodic (with period 1) in all variables; but can be extended for



	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08
Lattice Rule (GPU)	2.95E-01	3.02E-01	6.09E-01	2.94E+00	2.58E+01	2.59E+02
MC (curand) Double	2.27E+01	2.28E+01	2.36E+01	3.20E+01	1.10E+02	8.99E+02
MC (Random123)	2.45E-01	2.49E-01	3.60E-01	1.27E+00	9.85E+00	9.25E+01
Lattice Rule (Sequential)	5.91E-01	5.85E+00	5.86E+01	6.10E+02	6.15E+03	5.87E+04
MC (curand) Single, KS	2.30E+01	2.31E+01	2.35E+01	2.92E+01	8.07E+01	6.10E+02

Figure 4.3: Computation times for *Test 1*.

sufficiently smooth non-periodic functions by periodizing transformations. In this example, we give test results using a simple periodization [19] where the integrand function of Eq. (1.1) is replaced by

$$h(x_1, \ldots, x_d) = f(1 - 2|x_1 - \frac{1}{2}|, \ldots, 1 - 2|x_d - \frac{1}{2}|) \tag{4.4}$$

Also note that the transformations by Sidi [21][79][80] could be applied, such as the sin^m-transformation with $m = 2$ given by

$$x_i = t_i - \frac{1}{2\pi}\sin(2\pi t_i), \quad i = 1, \ldots, d \tag{4.5}$$

The integrand (4.3) has second order partial derivatives at the origin and is harder to integrate compared to the previous function. For this integral we show results with the periodization of Eq (4.4) for the lattice rules. We further used the same methods as for the

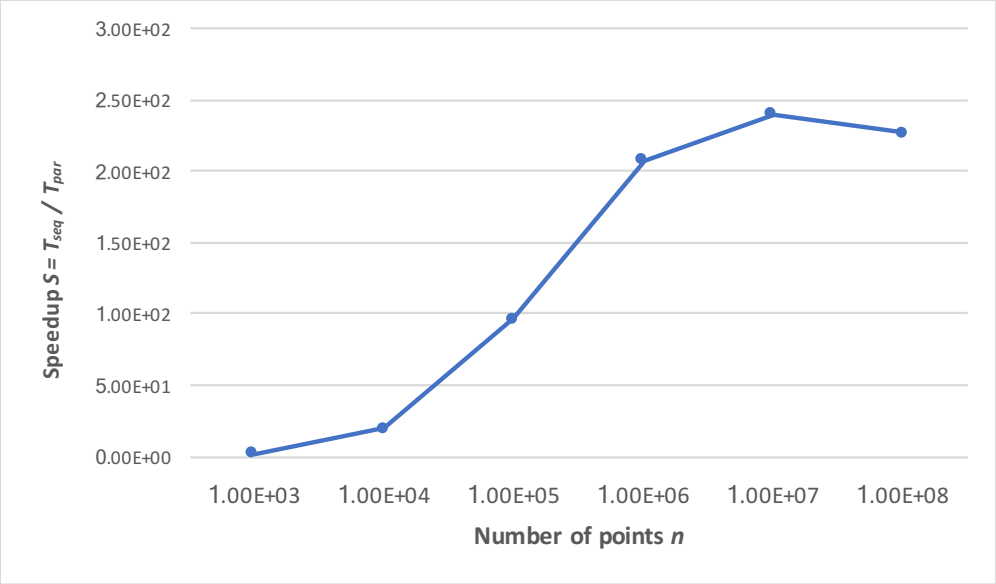
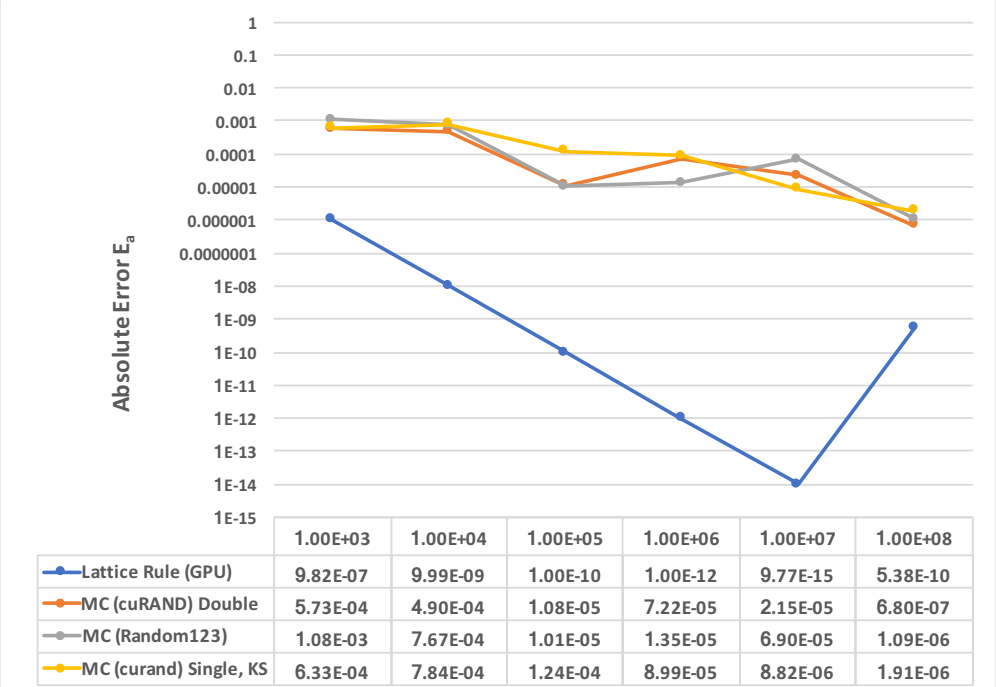
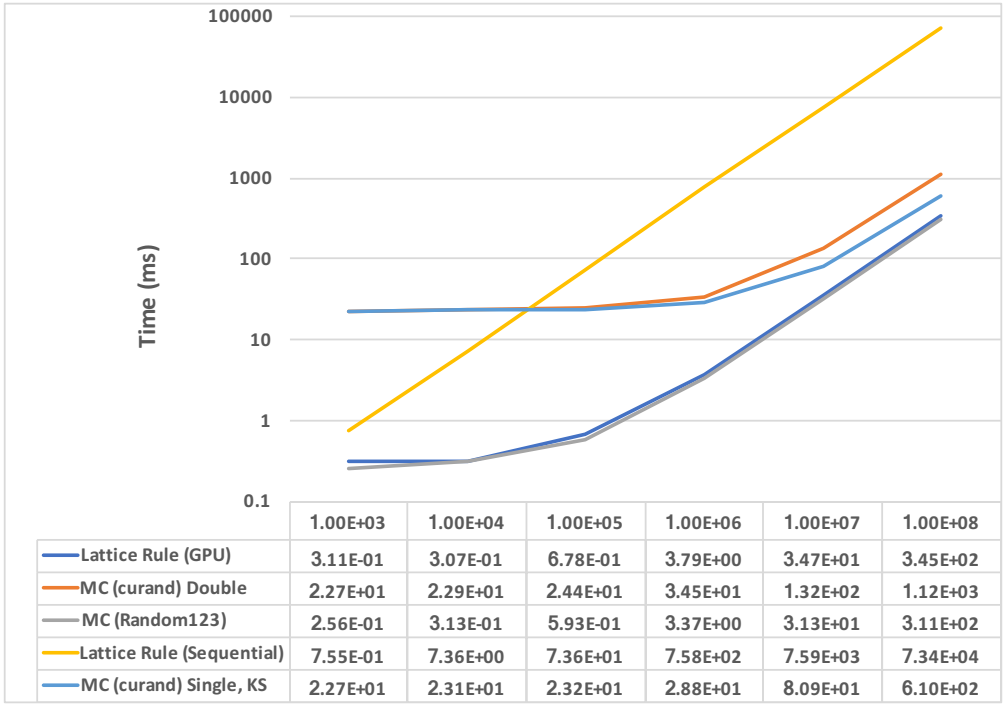


Figure 4.4: Parallel performance (Speedup vs. n) for *Test 1* lattice rule integration.



	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08
Lattice Rule (GPU)	9.82E-07	9.99E-09	1.00E-10	1.00E-12	9.77E-15	5.38E-10
MC (cuRAND) Double	5.73E-04	4.90E-04	1.08E-05	7.22E-05	2.15E-05	6.80E-07
MC (Random123)	1.08E-03	7.67E-04	1.01E-05	1.35E-05	6.90E-05	1.09E-06
MC (curand) Single, KS	6.33E-04	7.84E-04	1.24E-04	8.99E-05	8.82E-06	1.91E-06

Figure 4.5: Absolute error E_a at each value of n for *Test 2*.



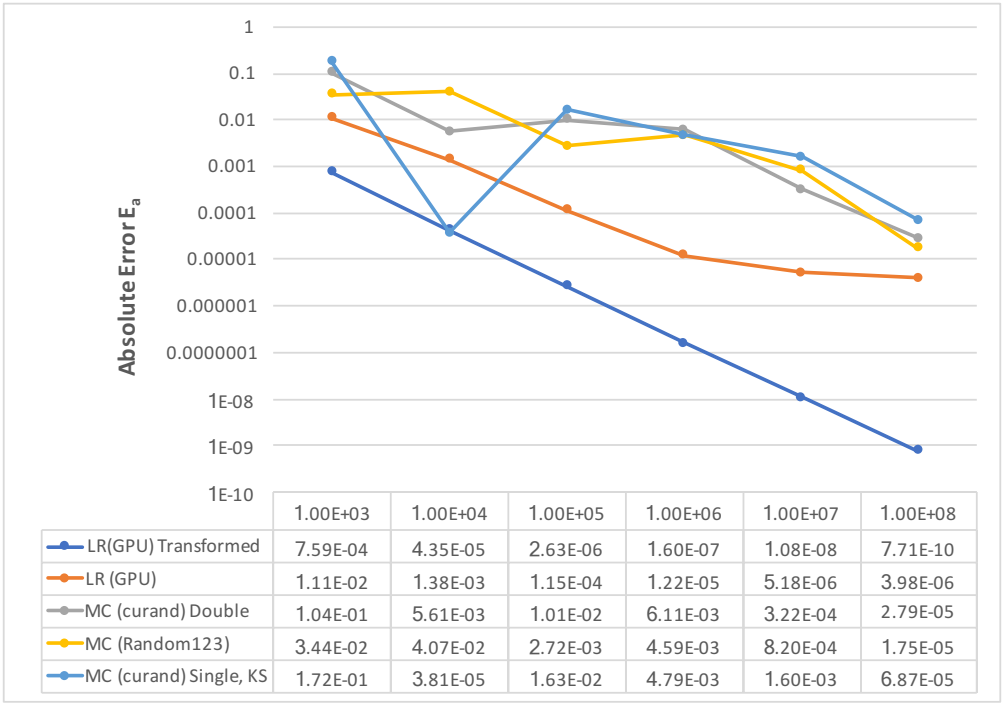
	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08
Lattice Rule (GPU)	3.11E-01	3.07E-01	6.78E-01	3.79E+00	3.47E+01	3.45E+02
MC (curand) Double	2.27E+01	2.29E+01	2.44E+01	3.45E+01	1.32E+02	1.12E+03
MC (Random123)	2.56E-01	3.13E-01	5.93E-01	3.37E+00	3.13E+01	3.11E+02
Lattice Rule (Sequential)	7.55E-01	7.36E+00	7.36E+01	7.58E+02	7.59E+03	7.34E+04
MC (curand) Single, KS	2.27E+01	2.31E+01	2.32E+01	2.88E+01	8.09E+01	6.10E+02

Figure 4.6: Computation times for *Test 2.*

previous integrals. The absolute error results are shown in Fig 4.7.

After the transformation, lattice rules (LR (GPU) Transformed) achieve the best accuracy at $n = 10^8$ with E_a = 7.71e-10. We note that the transformation by Sidi in Eq. (5.8) gives similar accuracy (8.63e-10 at $n = 10^8$) but takes more time (301 ms). Even without transformation, lattice rules yield better accuracy than to the Monte Carlo methods, with an absolute error of 3.98e-06 at $n = 10^8$.

Fig. 4.8 shows the computation times for each method. The times for lattice rules with and without the transformation are very similar and hardly distinguishable on the chart; thus the periodization is effective and does not affect the parallel performance. Lattice rules with the transformation perform the evaluation with 10^8 points in 116 ms, vs. 118 ms without the transformation. Monte Carlo sampling using Random123 achieves the best time (around 67 ms) at $n = 10^8$.



	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08
LR(GPU) Transformed	7.59E-04	4.35E-05	2.63E-06	1.60E-07	1.08E-08	7.71E-10
LR (GPU)	1.11E-02	1.38E-03	1.15E-04	1.22E-05	5.18E-06	3.98E-06
MC (curand) Double	1.04E-01	5.61E-03	1.01E-02	6.11E-03	3.22E-04	2.79E-05
MC (Random123)	3.44E-02	4.07E-02	2.72E-03	4.59E-03	8.20E-04	1.75E-05
MC (curand) Single, KS	1.72E-01	3.81E-05	1.63E-02	4.79E-03	1.60E-03	6.87E-05

Figure 4.7: Absolute error E_a vs. n for *Test 3*, comparing Monte Carlo methods, lattice rules on GPU with periodization (LR (GPU) Transformed), and lattice rules without transformation (LR (GPU)).

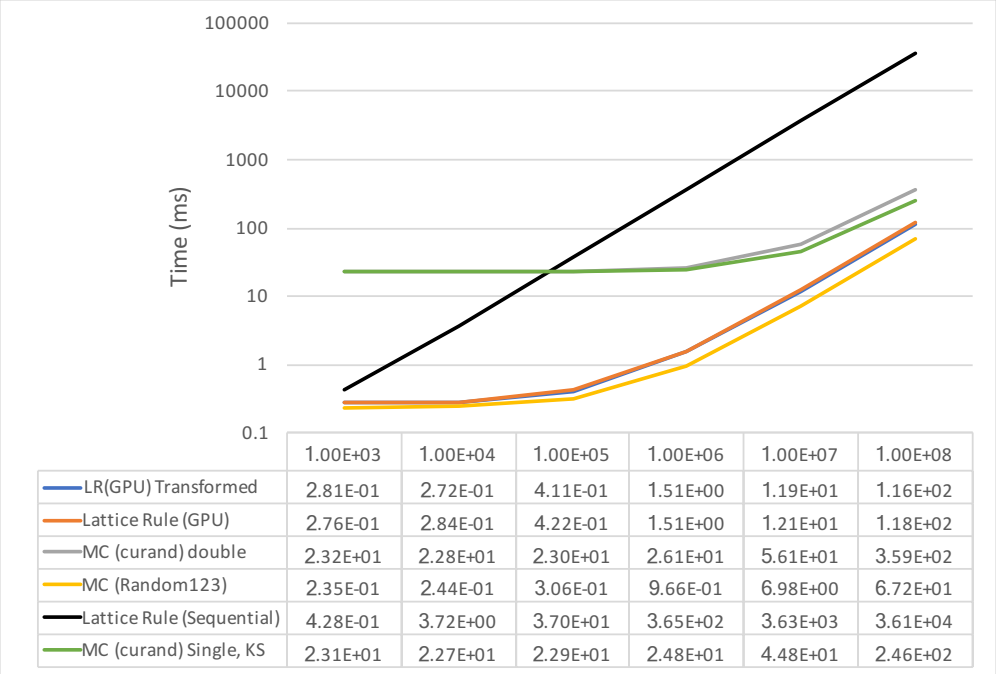
4.3.4. Test 4: Integrand singularity

This integral from [19] is 10-dimensional and of the form

$$\int_0^1 \cdots \int_0^1 \frac{1}{\left(\sum_{i=1}^{10} x_i\right)^2} dx_1 \ldots dx_{10} = 0.04483234483 \tag{4.6}$$

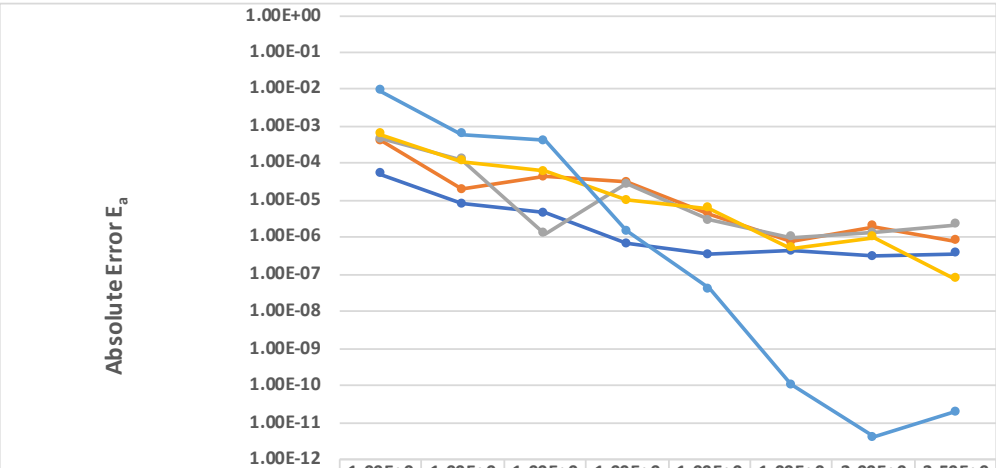
The integrand has a singularity at the origin. Fig. 4.9 plots the absolute errors resulting from the integration by each approach. We also show results of applying the Sidi transformation of Eq. (5.8).

The transformed lattice rules show significant improvements for $n = 10^7$ and higher. Overall, lattice rules are more accurate than Monte Carlo for this problem although, at $n = 3.5 \times 10^8$, Monte Carlo using cuRAND in single precision achieves the best accuracy



	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08
LR(GPU) Transformed	2.81E-01	2.72E-01	4.11E-01	1.51E+00	1.19E+01	1.16E+02
Lattice Rule (GPU)	2.76E-01	2.84E-01	4.22E-01	1.51E+00	1.21E+01	1.18E+02
MC (curand) double	2.32E+01	2.28E+01	2.30E+01	2.61E+01	5.61E+01	3.59E+02
MC (Random123)	2.35E-01	2.44E-01	3.06E-01	9.66E-01	6.98E+00	6.72E+01
Lattice Rule (Sequential)	4.28E-01	3.72E+00	3.70E+01	3.65E+02	3.63E+03	3.61E+04
MC (curand) Single, KS	2.31E+01	2.27E+01	2.29E+01	2.48E+01	4.48E+01	2.46E+02

Figure 4.8: Computation times for *Test 3*.

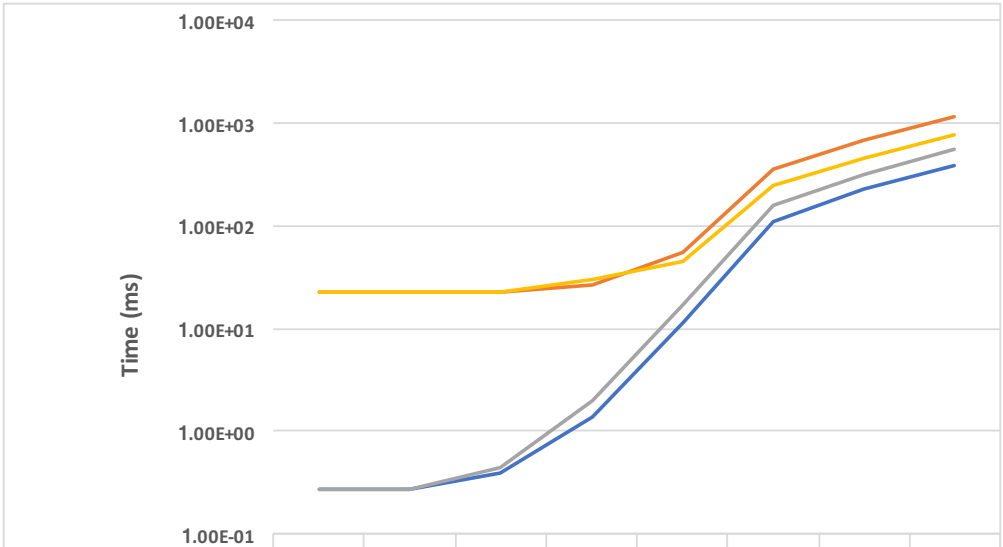


	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08	2.00E+08	3.50E+08
Lattice Rule (GPU)	5.09E-05	8.03E-06	4.71E-06	6.73E-07	3.40E-07	4.28E-07	3.17E-07	3.59E-07
MC (curand) Double	4.16E-04	1.99E-05	4.42E-05	3.18E-05	4.36E-06	7.57E-07	2.02E-06	8.22E-07
MC (Random123)	4.61E-04	1.30E-04	1.30E-06	2.74E-05	2.91E-06	9.88E-07	1.31E-06	2.27E-06
MC (curand) Single, KS	6.24E-04	1.12E-04	6.18E-05	1.01E-05	6.01E-06	5.22E-07	1.03E-06	7.82E-08
Lattice Rule (GPU) Transformed	9.43E-03	6.14E-04	4.06E-04	1.51E-06	4.06E-08	1.08E-10	3.99E-12	2.00E-11

Figure 4.9: Absolute error E_a for *Test 4*.

with $E_a = 7.8\text{e-}08$.

The computation time of the methods on GPU are shown in Fig. 4.10. As for the



	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08	2.00E+08	3.50E+08
Lattice Rule (GPU)	2.69E-01	2.69E-01	3.89E-01	1.40E+00	1.15E+01	1.12E+02	2.24E+02	3.91E+02
MC (curand) Double	2.26E+01	2.27E+01	2.30E+01	2.62E+01	5.58E+01	3.59E+02	6.87E+02	1.18E+03
MC (Random123)	2.74E-01	2.68E-01	4.44E-01	1.98E+00	1.67E+01	1.61E+02	3.22E+02	5.63E+02
MC (curand) Single, KS	2.30E+01	2.28E+01	2.29E+01	2.98E+01	4.47E+01	2.46E+02	4.62E+02	7.85E+02

Figure 4.10: Computation times for *Test 4*.

previous integrals, lattice rules and Monte Carlo using Random123 are much faster than cuRAND when n is relatively small. By increasing the number of points, the computing time of these methods increases significantly, especially between $n = 10^6$ and $n = 10^8$. For $n = 2 \times 10^8$, $n = 3.5 \times 10^8$, the time increases at a steady rate similar to the other methods.

4.4. Feynman Loop Integrals

Feynman diagrams are used extensively in high energy physics to describe particle interactions. In perturbation theory, these Feynman diagrams are translated into formulas that may contain integrals called "loop integrals" [81]. For a particle interaction, higher-order

contributions to the perturbation series of the amplitude in quantum field theory are expressed in terms of Feynman loop integrals. These correspond to Feynman diagrams, each of which represents a possible configuration. The square sum of the amplitudes gives the probability or cross section of the process. A Feynman loop integral with L loops and N propagators can be represented in Feynman parameter space by

$$I_N = \frac{\Gamma(N - \frac{\nu L}{2})}{(4\pi)^{\nu L/2}}(-1)^N \int_0^1 \prod_{r=1}^{N} dx_r \delta(1 - \sum x_r) \frac{C^{N-\nu(L+1)/2}}{(D - i\varrho C)^{N-\nu L/2}} \tag{4.7}$$

where C and D are polynomials determined by the topology of the corresponding diagram and physical parameters, and ν is the space-time dimension. For 1-loop diagrams, well-known analytic solutions are available. However, analytic integration is generally impossible for multi-loop diagrams with multiple external lines.

We applied lattice rule integration to classes of 2-loop box, as well as 3- and 4-loop self-energy Feynman diagrams, after a sin^m-transformation (Sidi [79] [80]), which was performed not only as a periodizing transformation but also to deal with integrand boundary singularities. We examined the effects of higher order transformations. The implementation in CUDA C was executed utilizing one Kepler 20m GPU for the integrand evaluations and summations over the blocks. The results, which are reported in [71], were found far more accurate and efficient than parallel adaptive integration layered over MPI and using 64 CPU cores in four cluster nodes.

In [72] we applied rank-1 lattice rules with $tanh$ and sin^m type transformations for the integration of functions which may have singularities on the boundaries of the integration domain. Sidi's sin^m-transformation (with $m = 6$) overall attained better accuracy than the tanh-transformation, and the combined lattice rule and transformation techniques achieved excellent accuracy and parallel efficiency.

Furthermore, in [73] we computed Feynman loop integrals for diagrams with massless internal lines, using lattice rule approximations accelerated on a GPU. Results are given for finite and UV-divergent diagrams having between 4 and 11 propagators, and showing promise for higher-dimensional applications. The method is fully numerical and thus does not require symbolic manipulations tailored to specific problems. The GPU results are accurate and efficient in execution time compared to other numerical methods and architectures.

CHAPTER 5

EMBEDDED LATTICE RULES

5.1. Overview

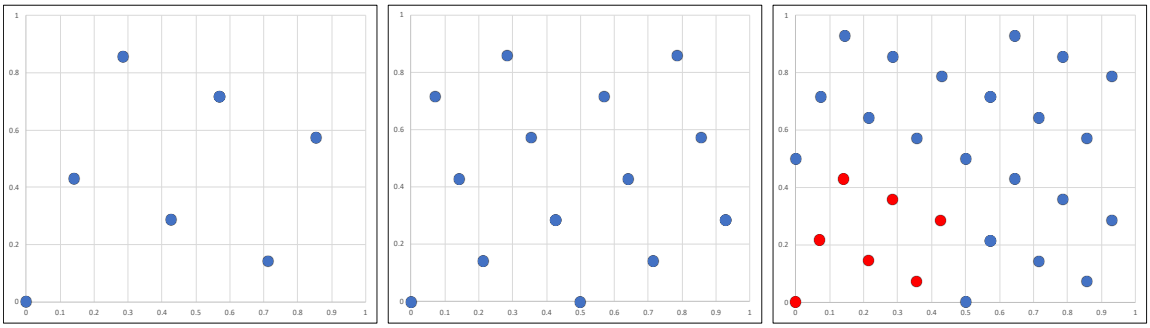
This chapter discusses the benefits of using a sequence of embedded lattice rules to improve the integration accuracy. This is a high-cost practical method that also provides an error estimation technique for the integration. Starting with a rank-1 rule Q_0, the embedded sequence grows to include every following rank up to d, which is the maximal rank. We will go over the algorithm by Sloan and Joe, who gave a Fortran code to implement the embedded sequence [21]. Then we will show some results from our code ported to CUDA C for accelerating the computations.

5.2. Embedded Lattice Sequence

As established in the previous chapter, good lattice rules with n points can be constructed using the CBC algorithm. An embedded sequence of lattice rules is defined over the unit cube as an m^d copy of a lattice rule that is scaled to size $1/m$ in each coordinate direction and $m \geq 1$ is the maximum level. The embedded sequence is given in [21] as

$$Q_r f = \frac{1}{m^r n} \sum_{k_r=0}^{m-1} \cdots \sum_{k_1=0}^{m-1} \sum_{j=0}^{n-1} f\left(\left\{\frac{j}{n} z + \frac{k_1, ..., k_r, 0, ..., 0}{m}\right\}\right) \tag{5.1}$$

for $1 \leq r \leq d$, and where m and n are relatively prime. In the embedded sequence, the points of Q_r are included in the points of Q_{r+1}. For $r = d$, the Q_d lattice rule can be written



(a) Q_0 (b) Q_1 (c) Q_2

Figure 5.1: a 2-dimensional embedded sequence with 7 points

as

$$Q_d f = \frac{1}{m^d n} \sum_{k_d=0}^{m-1} \ldots \sum_{k_1=0}^{m-1} \sum_{j=0}^{n-1} f\left(\left\{\frac{j}{mn} z + \frac{k_1, \ldots, k_d}{m}\right\}\right) \tag{5.2}$$

and corresponds to the m^d copy rule.

As an illustration, we use the following example with 7 points (similar to the 5-point example of [21]). Given

$$d = 2, n = 7, m = 2, z = (1, 3)$$

the quadrature points of the embedded sequence are shown in Fig 5.1. The figure shows the rank-1 basic lattice Q_0 with 7 points. Q_1 has all the points of Q_0 and incorporates two scaled copies of Q_0 by halving the unit square across the vertical coordinate direction and and applying the scaled copy to both halves. Then, Q_2 include all the points of Q_1 and further halves the square across the horizontal coordinate direction, to obtain four scaled copies, so that Q_2 is the 2^2 copy of Q_0.

Joe and Sloan [82] proposed an error estimation method based on the embedded sequence as:

$$E_t = c \left(\sum_{i=1}^{d} (Q_d f - Q^{(i)} f)^2 / d \right)^{\frac{1}{2}} \tag{5.3}$$

where c is a heuristic constant factor, $Q^{(i)}$ is a lattice rule of order $m^{d-1}n$ and $Q^{(d)} = Q_{d-1}$. Furthermore, $Q^{(i)}$ is embedded in Q_d for $1 \leq i \leq d$. The embedded sequence algorithm with its error estimation is implemented in the C code of Appendix B.

5.3. Numerical Results

To show the accuracy and performance of the embedded rules on the GPU, we revisit *Test 3* and *Test 4* from *Chapter 4* [70]. We focus on the accuracy and running time achieved for the 2^d copy rule. For the results given in this section, our CUDA programs are executed using an NVIDIA Tesla K20m [74] GPU with 2496 CUDA cores, 706 MHz GPU clock cycle and 4800 MB of global memory. The device is part of a cluster node with dual Intel Xeon E5-2670, 2.6 GHz CPUs and 128 GB of memory. The number of parallel blocks and threads per block launched in the GPU runs were set to 64 and 1024, respectively.

5.3.1. Test 1: Singularity of second order partial derivatives

This is a 10-dimensional integral from [19] of the form

$$\int_0^1 \dots \int_0^1 \left(\sum_{i=1}^{10} x_i \right)^{1.5} dx_1 \dots dx_{10} = 11.32097423155 \tag{5.4}$$

The integrand has second order partial derivative singularities at the origin, which makes it relatively hard to integrate. In this example, we give test results using a simple periodization [19] where the integrand function of Eq. (1.1) is replaced by

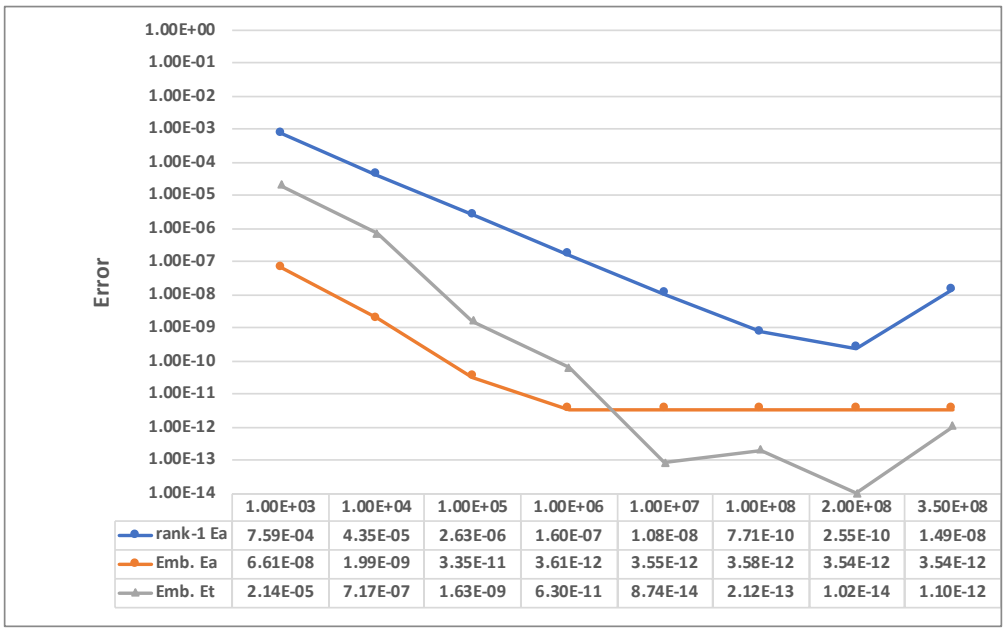
$$h(x_1, \dots, x_d) = f(1 - 2|x_1 - \frac{1}{2}|, \dots, 1 - 2|x_d - \frac{1}{2}|) \tag{5.5}$$

Numerical and performance results with the embedded rules are given in Fig. 5.2.

For each n, Fig. 5.2 compares the accuracy of the 2^d copy rule against the rank-1 lattice rule, based on the absolute error

$$E_a = |I(f) - Q_n(f)| \tag{5.6}$$

The figure also shows the estimated error (E_t) of the embedded sequence, which is calculated using Eq (5.3). In this case, the embedded sequence shows more accuracy than the rank-1



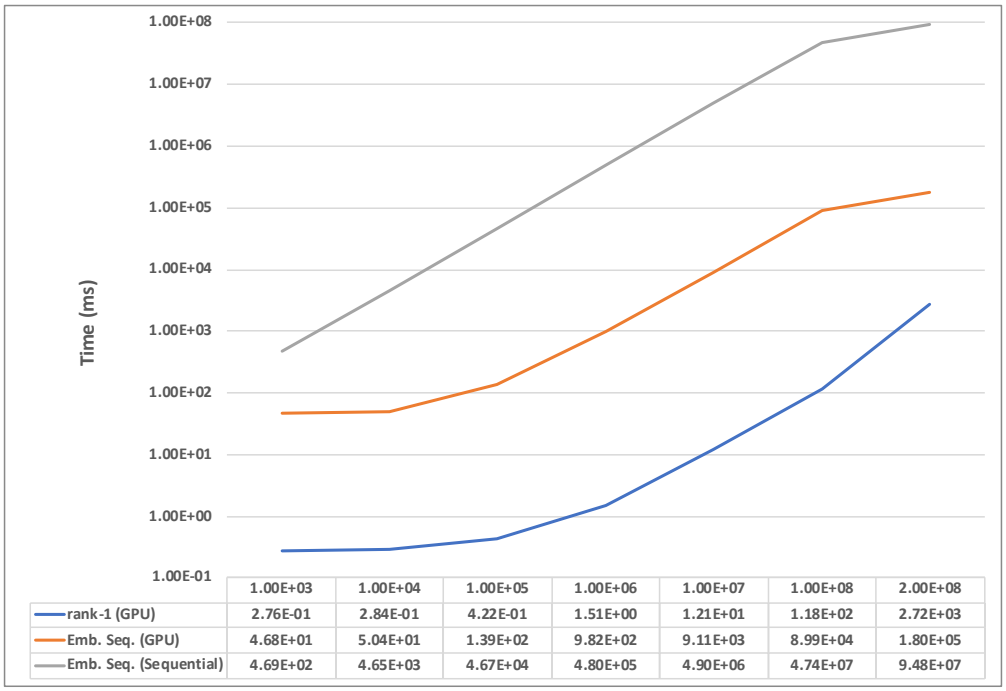
	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08	2.00E+08	3.50E+08
rank-1 Ea	7.59E-04	4.35E-05	2.63E-06	1.60E-07	1.08E-08	7.71E-10	2.55E-10	1.49E-08
Emb. Ea	6.61E-08	1.99E-09	3.35E-11	3.61E-12	3.55E-12	3.58E-12	3.54E-12	3.54E-12
Emb. Et	2.14E-05	7.17E-07	1.63E-09	6.30E-11	8.74E-14	2.12E-13	1.02E-14	1.10E-12

Figure 5.2: Absolute error E_a for rank-1 lattice rule and 2^d copy rule, and estimated error E_t for 2^d copy rule based on embedded sequence, on GPU at each value of m for *Test 1*

lattice rule. However, the error rate does not decrease beyond one Million points. The error estimate E_t is under-estimated for larger numbers of points.

Fig. 5.3 shows the computation times for each method in addition to the embedded sequence implemented sequentially. The execution time for the rank-1 lattice rules on the GPU is clearly much smaller than that of the embedded sequence. At 200 Million points, the rank-1 lattice rule took less than 3 seconds to return the results, while the embedded

sequence time on the GPU is around 3 minutes. However, it remains much faster than the sequential embedded sequence, which took around 26 hours to perform the integration.



	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08	2.00E+08
rank-1 (GPU)	2.76E-01	2.84E-01	4.22E-01	1.51E+00	1.21E+01	1.18E+02	2.72E+03
Emb. Seq. (GPU)	4.68E+01	5.04E+01	1.39E+02	9.82E+02	9.11E+03	8.99E+04	1.80E+05
Emb. Seq. (Sequential)	4.69E+02	4.65E+03	4.67E+04	4.80E+05	4.90E+06	4.74E+07	9.48E+07

Figure 5.3: Computation times for *Test 1.*

Fig. 5.4 depicts the obtained speedup S vs. the number of points m for *Test 1.* The speedup $S = T_{seq}/T_{par}$ is the ratio of sequential to parallel time, and the highest speedup ($S = 538$) is obtained at $n = 10^7$.

5.3.2. Test 2: Integrand singularity

This integral from [19] is 10-dimensional and of the form

$$\int_0^1 \dots \int_0^1 \frac{1}{\left(\sum_{i=1}^{10} x_i\right)^2} dx_1 \dots dx_{10} = 0.04483234483 \tag{5.7}$$

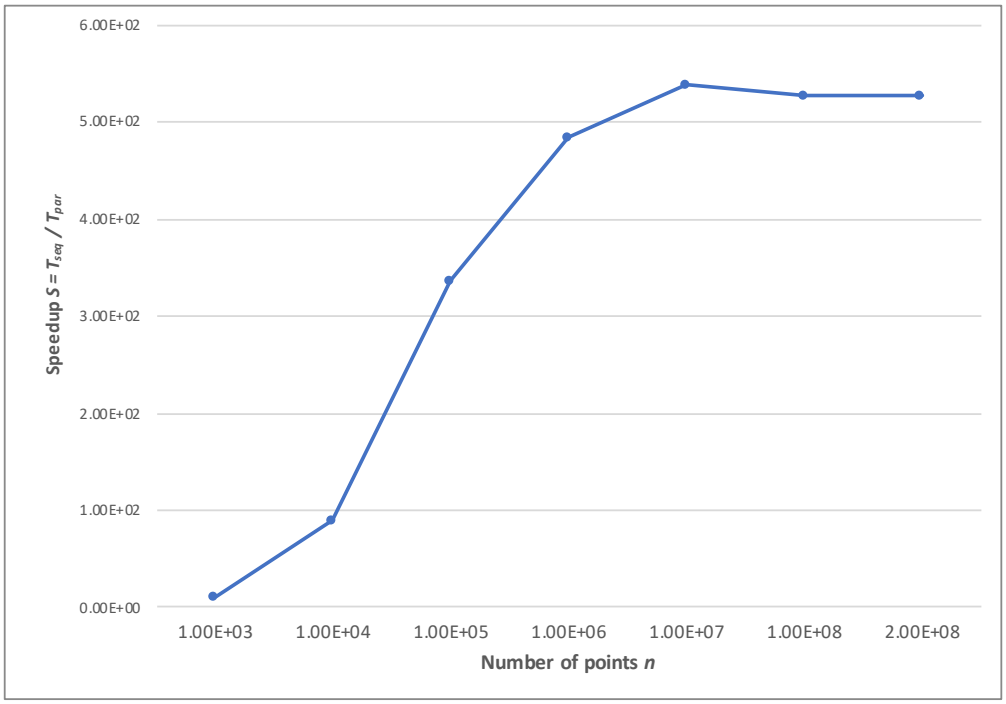


Figure 5.4: Parallel performance (Speedup vs. n) for *Test 1* lattice rule integration.

The integrand has a singularity at the origin. We show results after applying a transformation by Sidi [21][79][80], the sin^m-transformatiwith $m = 2$ given by

$$x_i = t_i - \frac{1}{2\pi}\sin(2\pi t_i), \quad i = 1, \ldots, d \tag{5.8}$$

Fig. 5.5 plots the absolute error of the rank-1 lattice rule and the 2^d copy rule, and the estimated error resulting from the embedded sequence. The figure clearly shows that the embedded sequence (2^d copy rule) gives more accurate results especially for a lower number of points. At $n = 10^6$ the embedded sequence achieved $E_a = 4.56\,\text{e-}12$ compared to the rank-1 lattice rule $E_a = 1.51\,\text{e-}6$.

Regarding performance, Fig. 5.6 plots the the parallel and sequential computing times of the rank-1 lattice rules and the embedded sequence. Similar to $Test1$, the rank-1 lattice rule

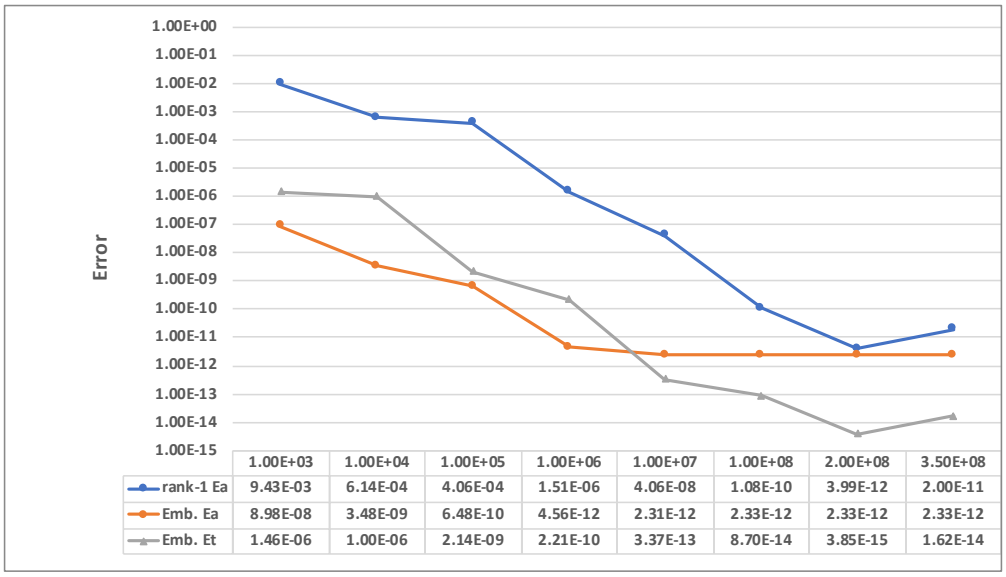


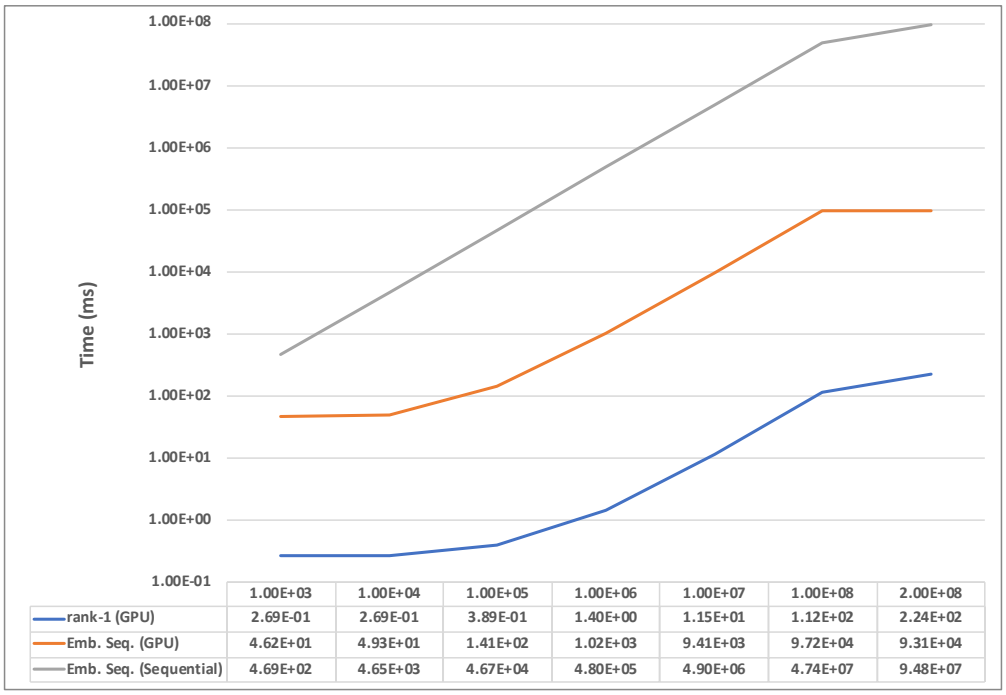
Figure 5.5: Absolute error E_a for rank-1 lattice rule and 2^d copy rule, and estimated error E_t for 2^d copy rule based on embedded sequence, on GPU at each value of n for *Test 2*.

performs far better with 2.7 seconds at $n = 200$ Million points compared to the embedded sequence that took around 3 minutes for the same number of points. On the other hand, the sequential embedded sequence was far slower, consuming around 26 hours to return the results.

With respect to speedup, Table 5.1 shows the ratio of the sequential to parallel time, which scores the best at $n = 10^7$ points with a speedup of 423.77.

Table 5.1: Parallel performance (Speedup vs. n) for *Test 1*.

n	SPEEDUP
10^3	8.21
10^4	74.33
10^5	268.41
10^6	377.24
10^7	423.77
10^8	309.96
2×10^8	413.21



	1.00E+03	1.00E+04	1.00E+05	1.00E+06	1.00E+07	1.00E+08	2.00E+08
rank-1 (GPU)	2.69E-01	2.69E-01	3.89E-01	1.40E+00	1.15E+01	1.12E+02	2.24E+02
Emb. Seq. (GPU)	4.62E+01	4.93E+01	1.41E+02	1.02E+03	9.41E+03	9.72E+04	9.31E+04
Emb. Seq. (Sequential)	4.69E+02	4.65E+03	4.67E+04	4.80E+05	4.90E+06	4.74E+07	9.48E+07

Figure 5.6: Computation times for *Test 2*.

CHAPTER 6

ACCELERATING LATTICE RULES CONSTRUCTION

6.1. Overview

The fast CBC construction of rank-1 lattice rules indeed improves the CBC algorithm performance considerably. However, as we have shown in Chapter 4, increasing the number of lattice points may increase the integration accuracy. In order to generate a lattice rule for a large number of points, even the fast CBC construction will take time to calculate the rule. This chapter will address the fast CBC implementation using Matlab and C, and a parallelization for GPU accelerators using CUDA.

6.2. Fast CBC Implementation

There are a limited number of tools that implement the fast CBC algorithm. This section we will give a brief overview of some of these implementations.

6.2.1. Fast Rank-1

Nuyens has provided a Matlab implementation of the fast CBC construction of rank-1 lattice rules in [65]. The script uses the Matlab built-in FFT library that can be easily invoked to perform the matrix vector multiplication. A pseudo-code adapted from the Matlab script for the fast rank-1 construction is given in Fig 6.1. The inputs for this code are [37]:

- n: A prime number that represents the number of points in the lattice.
- dim: The dimension of the generator vector.

- *omega*: A symmetric function handle for the shift-invariant kernel function. In our experiments we used $2\pi^2(x^2 - x + \frac{1}{6})$.

- *gamma*, *beta*: Parameters for weighting per dimension.

In this implementation, w is the variable kernel function and it is symmetric around 1/2. Therefore the FFT transforms need only half the size n. Matlab only has complex to complex FFT transforms. Since w is real-valued, this can be solved by using the FFTW real to complex transform in C (and complex to real inverse transform) [65], thereby decreasing the memory consumption.

6.2.2. Lattice Builder

The *Lattice Builder* library is written in C++ and it can be used as a standalone application or invoked by other programs. The package supports the following methods [41]:

1. *Exhaustive search*

   This method considers all possible generating vectors $\mathbf{z} \in U_n^d$, where $U_n = \{z \in \mathbb{Z}_n \mid \gcd(z, n) = 1\}$ for $n > 1$ (set of units modulo n, identified with the multiplicative group of integers modulo n). The resulting vector will have the smallest error margin.

2. *Random search*

   Instead of searching all possible $\mathbf{z}$, this method selects x random values of $\mathbf{z}$ uniformly distributed in U_n^d and returns the best generating vector. The goal is to reduce the search time.

3. *Korobov method*

   This classic method for constructing lattice rules is similar to the exhaustive search

Algorithm 3: Fast rank-1 lattice rules construction

```
Is n a prime? Continue
m = (n − 1)/2
cumbeta = cumulative_products_of (beta)
g = all_primitive_roots_of (n)
for i=1 to m do
    perm[i] = mod(perm[i] ∗ g, n)
end
for i=1 to m do
    perm[i] = minimum (n – perm[i], perm[i])
    psi[i] = omega(perm/n)
end
fft_psi = fft(psi)
q[m] = {1,1,...,1}
for i=1 to dim do
    E2 = ifft(fft_psi * fft(q))
    min_E2 = minimum of (E2)
    w = index_of_minimum_of (E2)
    if i=1 and w=1 then
        noise = abs(E2[1] - min_E2)
    end
    z[i] = perm[w]
    e2[i] = -cumbeta[i] + (beta[i] * (q[1] + 2 * sum_q) + gamma[i] * (psi[1] * q[1] + 2
     * min)) /n
    wcount = w
    mcount = m-1
    for k=1 to w do
        psiord[k] = psi[wcount]
        wcount = wcount - 1
    end
    for k=w+1 to m do
        psiord[k] = psi[mcount]
        mcount = mcount - 1
    end
    for k=1 to m do
        q[k] = q[k] * (beta[i] + gamma[i] * psiord[k])
    end
    print z[i], e2[i]
end
```

Figure 6.1: Pseudo-code adapted from Nuyens Matlab code [65].

method. However, it only considers generating vectors of the form
$\mathbf{z} = (1,\ a,\ a^2 \mod n, ...,\ a^d \mod n)^t$ with $a \in U_n$.

4. *Random Korobov method*

   As in the random search, this method selects x random values of a uniformly distributed in U_n and giving rise to the Korobov vector $\mathbf{z}$.

5. *CBC method*

   This is the component by component construction of rank-1 lattice rules algorithm [36] [83]. In this method, the generating vector $\mathbf{z}$ is obtained one component at a time, starting with the first component $z_1 = 1$. With $z_1, \ldots, z_{j-1}$ fixed at step j, the algorithm chooses $z_j \in U_n$ to minimize the worst-case error in a shift-invariant reproducing kernel Hilbert space.

6. *Random CBC*

   As a randomized CBC this algorithm chooses x random values z uniformly distributed in U_n to be considered as z_j for $j = 2, ..., d$.

7. *Fast CBC*

   Fast CBC is a faster version of the CBC algorithm where FFT operations are used to reduce the algorithm complexity from $O(dn^2)$ to $O(dn\ \log(n))$. This implementation requires the number of points n to be a power of a prime number. *Lattice Builder* uses the FFTW library [84] for the implementation.

8. *Range of points*

   Lattice rules built with the CBC method are extensible in dimension d but not extensible in the number of points n. Consequently, when n needs to be modified, the generating vector has to be constructed again [85]. Cools et al. proposed a method

for constructing lattice rules for a range of n based on the fast CBC algorithm, with complexity of $O(dn(log(n))^2)$ in [85]. *Lattice Builder* applies this method to augment the maximum number of points $n_m = b^m$, for embedded lattice rules in base b and maximum level m, to $n_{m+1} = b^{m+1}$ [41].

6.2.3. Accuracy Issues

Through our experiments, we noticed that running the code on different machines may produce different rules. Table 6.1 shows two rules that are constructed for 1009 points and 10 dimension. Rule 1 was constructed on a computer that has a 3.06GHz Core 2 Duo processor and 8 GB 1067MHz DDR3 RAM. Rule 2 was obtained using a machine with a 2.9 GHz Intel Core i5 processor and 8 GB 2133 MHz LPDDR3 RAM.

Table 6.1: Two Rules generated with the same number of points ($n = 1009$) and the same dimension ($d = 10$) using two different computers

Rule 1	1, 282, 381, 428, 79, 320, 171, 356, 130, 266
Rule 2	1, 390, 267, 435, 469, 316, 96, 402, 250, 187

Applying both rules to an integration gives similar results. For example, for the integral

$$\int_0^1 \dots \int_0^1 \left(\sum_{i=1}^{10} x_i\right)^{1.5} dx_1 \dots dx_{10} = 11.32097423155 \tag{6.1}$$

Rule 1 yields the integration result 11.309868 while Rule 2 gives the result 11.306319. Thus the results are quite similar for both rules.

Further examination of the code reveals that the differences in the rules are directly associated with the FFT operations, which vary on different machines. According to [86] there are four possible causes of error in FFT computations:

1. Unstable factorization in computing FFT.

2. Unstable DFT blocks.

3. Inaccuracy of computing trigonometric constants.

4. Computation roundoff errors.

The last two error sources are far more common when using the FFTW library [86]. FFTW implements the Cooley-Tukey algorithm, which has a floating-point error of $O(\log N)$ for a size N transform [87]. The error bound can be increased by an inaccurate calculation of the trigonometric (twiddle) factors. A poor calculation of the twiddle factors can increase the FFT floating-point error up to $O(N^2)$ [88].

6.3. Numerical Results

In order to test and enhance the performance of the fast CBC, we developed a C code for the algorithm. The cornerstone of implementing the fast CBC is a good FFT library. We have chosen the FFTW library to perform the FFT and IFFT operations. We also use the cuFFT library by NVIDIA to accelerate the construction of lattice rules.

6.3.1. FFTW

The Fastest Fourier Transform in the West (FFTW) is a well optimized open source software library written in C. It was developed at the Massachusetts Institute of Technology (MIT) by Matteo Frigo and Steven Johnson [84] [89]. FFTW is an implementation of the Cooley-Tukey FFT algorithm [90] that uses a technique that adapts to the computer architecture, and specialized compiler optimization for better performance [84]. The FFTW performance is usually better than other publicly accessible FFT programs. That was shown by a number of benchmarks done on a variety of platforms [84].

6.3.2. CuFFT

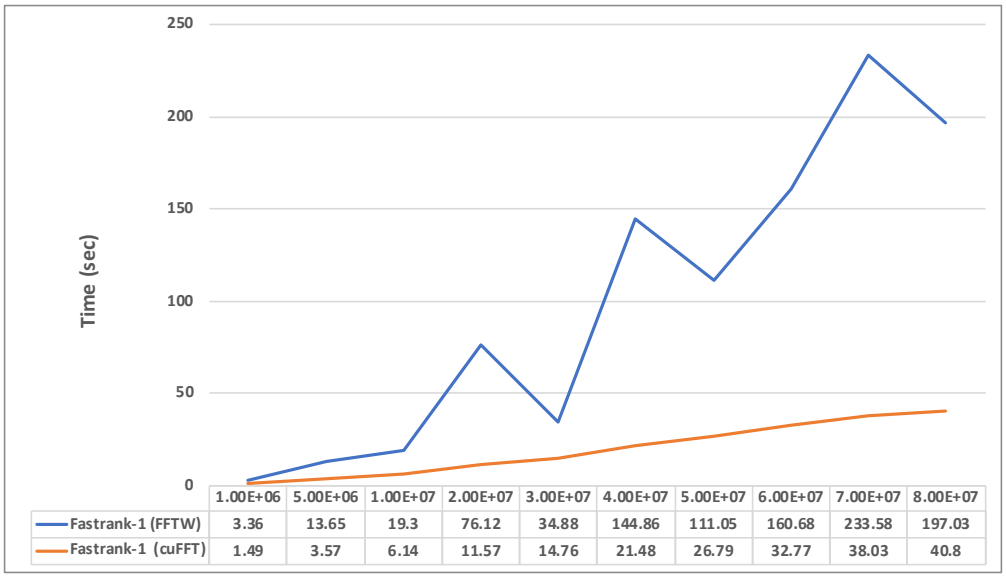
The NVIDIA CUDA Fast Fourier Transform library (cuFFT) provides GPU accelerated FFT implementations that, in theory, perform much faster than on CPU only. The library uses simple APIs, which enable the user to accelerate CPU based FFT implementations in some applications with little code changes. cuFFT is a foundational library based on the popular Bluestein [91] and Cooley-Tukey [90] algorithms. It supports academic and commercial use across disciplines like computational physics, quantum chemistry, molecular dynamics, medical and seismic imaging [92].

6.3.3. Results

To test the C code, using both the FFTW and cuFFT libraries, we generated a number of lattice rules for various numbers of points and dimensions. The main focus in these experiments was on the large number of points in order to test the performance sequentially and in parallel. Therefore, we chose the range of points to be between one million and 80 million, which is the largest number of points our GPU memory can contain.

For the tests described in this section, we used an NVIDIA Tesla M2090 Fermi GPU with 650 MHz GPU clock cycle and 6 GB of global memory. The device is part of a cluster node with dual Intel Xeon E5-2670, 2.6 GHz CPUs and 128 GB of memory. The number of parallel blocks and threads per block launched in the GPU runs were set automatically by the cuFFT calls.

For our first test, we generated 10-dimensional lattice rules for the target range of points. Fig. 6.2 shows the computation times for constructing these lattice rules sequentially and in parallel. The graph shows that at one million points, FFTW and the cuFFT are performing similarly. As the number of points increases, their performance starts to differ. At 80 million



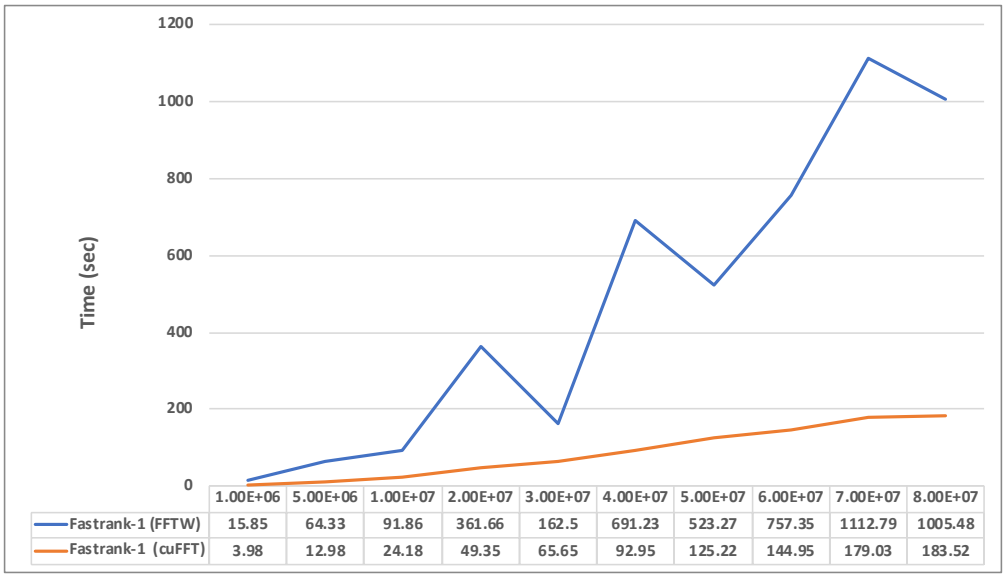
	1.00E+06	5.00E+06	1.00E+07	2.00E+07	3.00E+07	4.00E+07	5.00E+07	6.00E+07	7.00E+07	8.00E+07
Fastrank-1 (FFTW)	3.36	13.65	19.3	76.12	34.88	144.86	111.05	160.68	233.58	197.03
Fastrank-1 (cuFFT)	1.49	3.57	6.14	11.57	14.76	21.48	26.79	32.77	38.03	40.8

Figure 6.2: Computation times for constructing a 10-dimensional rule for various number of points n.

points, the code using FFTW generated the rule in around 3.3 minutes. The code using cuFFT constructed the rules in 40.8 seconds. The speedup for this test ranged between 2.2 at one million points and 6.74 at 40 million points.

For our second test, we increased the number of dimensions in the lattice rules to 50. Fig. 6.3 shows the computation times for constructing these rules using FFTW and cuFFT. The charted times look similar to the first test graph except for the increase in time for each n for FFTW and cuFFT. The speedup remains similar to the previous test, ranging between 3.9 for one million and 7.4 for 40 million. Constructing the rule for 80 million points using FFTW took around 16.75 minutes as opposed to cuFFT, which took around 3 minutes.

In the third test, we constructed lattice rules of dimension 100 for the aforementioned rang of n. Fig. 6.4 shows the computation times for constructing these rules using FFTW and cuFFT. The graph shows that the performance difference is similar to the previous tests. Constructing the 100-dimensional rules with 80 million points took around 32 minutes while cuFFT took around 7 minutes. The speedup for this test ranges between 5.13 for 5 million



	1.00E+06	5.00E+06	1.00E+07	2.00E+07	3.00E+07	4.00E+07	5.00E+07	6.00E+07	7.00E+07	8.00E+07
Fastrank-1 (FFTW)	15.85	64.33	91.86	361.66	162.5	691.23	523.27	757.35	1112.79	1005.48
Fastrank-1 (cuFFT)	3.98	12.98	24.18	49.35	65.65	92.95	125.22	144.95	179.03	183.52

Figure 6.3: Computation times for constructing a 50-dimensional rule for various numbers of points n.

points and 7.77 for 20 million. Table 6.2 shows the times and the speedup for the given range of n.

The results we obtained with cuFFT to accelerate the construction of rank-1 lattice rules for large numbers of points, appear to be promising. As mentioned before, the limitation of the Tesla M2090 memory size prevented us from constructing lattice rules larger than 80 million points. It would be interesting to construct the lattice rules using a state of the art GPU with larger memory or even multiple GPUs to increase the efficiency.

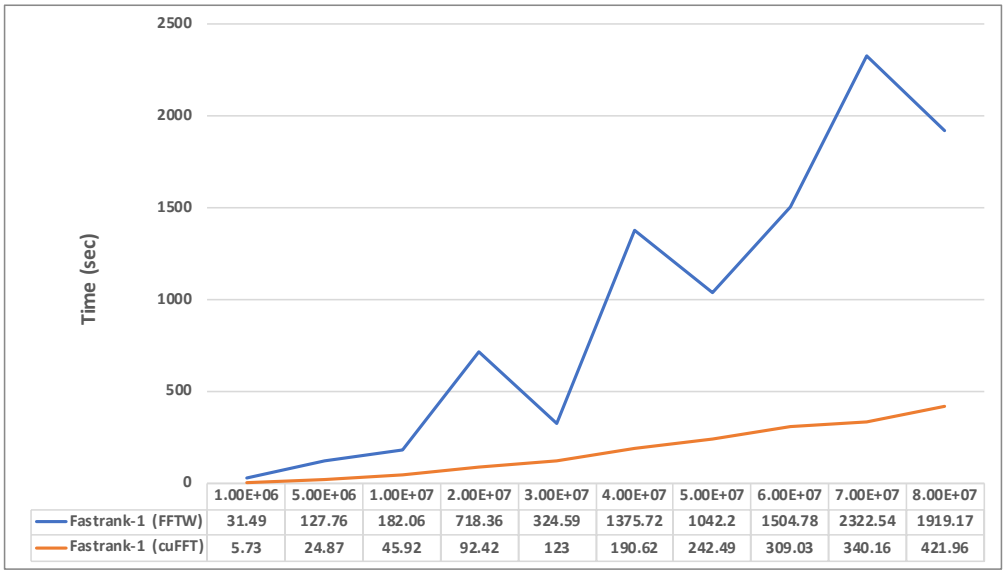


Figure 6.4: Computation times for constructing a 100-dimensional rule for various numbers of points n.

Table 6.2: Computation times and speedup for *Test 3*.

n	FFTW Time (s)	cuFFT Time (s)	Speedup
$1M$	31.49	5.73	5.49
$5M$	127.76	24.87	5.13
$10M$	182.06	45.92	3.96
$20M$	718.36	92.42	7.77
$30M$	324.59	123.00	2.63
$40M$	1375.72	190.62	7.21
$50M$	1042.20	242.49	4.29
$60M$	1504.78	309.03	4.8
$70M$	2322.54	340.16	6.8
$80M$	1919.17	421.96	4.5

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

In this dissertation we explored the potentials of high performance computing for multivariate integration. Our main focus was the use of a large number of function evaluations in order to increase the integration accuracy while achieving good parallel efficiency.

In a study of adaptive integration and task partitioning we presented applications from Bayesian statistics, including an application to health care data that led to the computation of difficult multivariate integrals. The ParInt package supplies tools for automatic multivariate integration in a black-box approach, to alleviate the need for a detailed specialized analysis of the integral by the user. We demonstrated the versatility of the package for use in different settings and found that it can be applied to these problems easily and effectively. We note that similar adaptive partitioning and load balancing techniques can support various other divide-and-conquer methods beyond numerical integration.

With regard to the adaptive techniques in ParInt, future work includes the development and testing of new subdivision and load balancing strategies, as well as a careful analysis of the need to deal with roundoff as the number of subdivisions increases significantly, especially near singularities.

Our work on GPU computations of lattice rules was motivated in part as an extension of the number-theoretic (quasi-Monte Carlo and Monte-Carlo) methods in the ParInt integration package. We derived an efficient GPU implementation in CUDA C for multidimensional integration using lattice rules. Compared to the Monte Carlo method, the lattice rules yield better accuracy for the various functions considered. We observed good convergence even for non-smooth integrand behavior, especially singularities on the boundaries of the domain,

which were treated with general transformations to smoothen the singular behavior. Unlike Monte Carlo, our lattice rule integration scheme does not rely on a pseudo-random number generator; the latter affects the computation time. Thus our test results for the lattice rule integration show improvements over Monte Carlo with respect to efficiency as well as accuracy. The lattice rule procedure further achieves excellent speedups on GPU compared to the sequential implementation. This benefits the approximation of integrals in higher dimensions using a large number of points.

Regarding embedded lattice sequences, we achieved significant improvements in efficiency and accuracy by utilizing the computational power of GPUs. This approach further provides an efficient method for estimating the integration error. The parallel tests we carried out show remarkable speedups against the sequential method.

Constructing good lattice rules for a large number of points can be a computationally intensive process. For constructing good lattice rules in a timely manner, the fast CBC algorithm relies on performing multiple FFT and IFFT operations as a major part of the algorithm. We used the NVIDIA cuFFT library to accelerate the FFT operations, with consistent and promising results.

The parallel algorithms coded in CUDA C have allowed significant contributions to the calculation of Feynman loop integrals where the integrand has boundary singularities handled with transformations. The integration algorithm has delivered a powerful black-box procedure to tackle these difficult problems.

BIBLIOGRAPHY

[1] R. E. Bellman, *Adaptive control processes: a guided tour*. Princeton university press, 1961.

[2] M. Simonovits, "How to compute the volume in high dimension?", *Mathematical programming*, vol. 97, no. 1-2, pp. 337–374, 2003.

[3] E. de Doncker, A. Genz, and M. Ciobanu, "Parallel computation of multivariate normal probabilities", *Computing Science and Statistics*, vol. 30, 1999.

[4] E. de Doncker and R. Assaf, "GPU integral computations in stochastic geometry", *Lecture Notes in Computer Science, VII Workshop Comp. Geometry and Applics. (CGA)*, vol. 7972, pp. 129–139, 2013, doi:10.1007/978-3-642-39643-4_10.

[5] D. J. Wilkinson, "Parallel Bayesian computation", in *Handbook of Parallel Computing and Statistics*, E. J. Kontoghiorghes, Ed., Marcel Dekker/CRC Press, 2005, ch. 16, pp. 481–512.

[6] E. de Doncker, K. Kaugars, L. Cucos, and R. Zanny, "Current status of the ParInt package for parallel multivariate integration", in *Proc. of Computational Particle Physics Symposium (CPP 2001)*, 2001, pp. 110–119.

[7] S. H. Paskov and J. F. Traub, "Faster valuation of financial derivatives", *J. Portfolio Management*, vol. 22, no. 1, pp. 113–120, 1995.

[8] S. Ninomya and S Tezuka, "Toward real-time pricing of complex financial derivatives", *Applied Mathematical Finance*, vol. 3, pp. 1–20, 1996.

[9] M. Evans and T. Swartz, *Approximating integrals via Monte Carlo and deterministic methods*. OUP Oxford, 2000, vol. 20.

[10] R. Eckhardt, "Stan Ulam, John von Neumann, and the Monte Carlo Method", *Los Alamos Science*, vol. 15, 1987. [Online]. Available: `http://la-science.lanl.gov/lascience15.shtml`.

[11] W Feller, "The strong law of large numbers", *An introduction to probability theory and its applications*, vol. 1, pp. 243–245, 1968.

[12] R. E. Caflisch, "Monte Carlo and Quasi-Monte Carlo methods", *Acta numerica*, vol. 7, pp. 1–49, 1998.

[13] R. Wikramaratna, "Pseudo-random number generation for parallel Monte Carlo-a splitting approach", *SIAM News*, vol. 33, no. 9, pp. 1–5, 2000.

[14] E. De Doncker, J. Kapenga, and R. Assaf, "Monte Carlo automatic integration with dynamic parallelism in CUDA", in *Numerical Computations with GPUs*, Springer, 2014, pp. 273–298.

[15] R. Schürer, "A comparison between (quasi-) Monte Carlo and cubature rule based methods for solving high-dimensional integration problems", *Mathematics and computers in simulation*, vol. 62, no. 3, pp. 509–517, 2003.

[16] J. Monahan and A. Genz, "Spherical-radial integration rules for Bayesian computation", *Journal of the American Statistical Association*, vol. 92, pp. 664–674, 1997.

[17] A. B. Owen, *Monte Carlo theory, methods and examples.* 2013.

[18] G. Leobacher and F. Pillichshammer, *Introduction to quasi-Monte Carlo integration and applications*. Springer, 2014.

[19] P. J. Davis and P. Rabinowitz, *Methods of numerical integration*. Courier Corporation, 2007.

[20] J. Dick, F. Y. Kuo, and I. H. Sloan, "High-dimensional integration: the Quasi-Monte Carlo way", *Acta Numerica*, vol. 22, pp. 133–288, 2013.

[21] I. H. Sloan and S. Joe, *Lattice methods for multiple integration*. Oxford University Press, 1994.

[22] R. Cools and D. Nuyens, "A Belgian view on lattice rules", in *Monte Carlo and Quasi-Monte Carlo Methods 2006*, Springer, 2008, pp. 3–21.

[23] D. Nuyens and R. Cools, "Fast Algorithms for Component-by-Component Construction of Rank-1 Lattice Rules in Shift-Invariant Reproducing Kernel Hilbert Spaces", *Mathematics of Computation*, vol. 75, pp. 903–920, 2006.

[24] R. Cranley and T. N. L. Patterson, "Randomization of number theoretic methods for multiple integration", *SIAM J. Numer. Anal.*, vol. 13, pp. 904–914, 1976.

[25] J. Berntsen, T. O. Espelid, and A. Genz, "An adaptive algorithm for the approximate calculation of multiple integrals", *ACM Trans. Math. Softw.*, vol. 17, pp. 437–451, 1991.

[26] J Kanzaki, "Monte Carlo integration on GPU", *The European Physical Journal C-Particles and Fields*, vol. 71, no. 2, pp. 1–7, 2011.

[27] E. de Doncker, J. Kapenga, and R. Assaf, "Monte Carlo automatic integration with dynamic parallelism in cuda", in *Numerical Computations with GPUs*, V. Kindratenko, Ed., ISBN 978-3-319-06548-9, Springer, 2014.

[28] D. Szałkowski and P. Stpiczyński, "Using distributed memory parallel computers and gpu clusters for multidimensional Monte Carlo integration", *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 923–936, 2015.

[29] B. Shareef, E. De Doncker, and J. Kapenga, "Monte Carlo simulations on Intel Xeon Phi: Offload and native mode", in *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, IEEE, 2015, pp. 1–6.

[30] R. Cools and A. Haegemans, "Cubpack: Progress report", in *Numerical Integration*, Springer, 1992, pp. 305–315.

[31] ——, "Algorithm 824: Cubpack: A package for automatic cubature; framework description", *ACM Transactions on Mathematical Software (TOMS)*, vol. 29, no. 3, pp. 287–296, 2003.

[32] T. Hahn, "Cuba?a library for multidimensional numerical integration", *Computer Physics Communications*, vol. 168, no. 2, pp. 78–95, 2005.

[33] ——, "Concurrent cuba", *Computer Physics Communications*, vol. 207, pp. 341–349, 2016.

[34] E. de Doncker, J. Fujimoto, N. Hamaguchi, T. Ishikawa, Y. Kurihara, Y. Shimizu, and F. Yuasa, "Quadpack computation of Feynman loop integrals", *Journal of Computational Science (JoCS)*, vol. 3, no. 3, pp. 102–112, 2011, doi:10.1016/j.jocs.2011.06.003.

[35] S. H. Paskov, "Computing high dimensional integrals with applications to finance", Columbia University, Department of Computer Science, Tech. Rep., 1994, CUCS-023-94.

[36] I Sloan and A Reztsov, "Component-by-component construction of good lattice rules", *Mathematics of Computation*, vol. 71, no. 237, pp. 263–273, 2002.

[37] D. Nuyens. (2006). Fast component-by-component constructions codes, [Online]. Available: `https://people.cs.kuleuven.be/~dirk.nuyens/fast-cbc/` (visited on 05/19/2017).

[38] P. L'Ecuyer, *SSJ: Stochastic simulation in Java, software library*, `http://simul.iro.umontreal.ca/ssj/`, 2016.

[39] P. L'Ecuyer, L. Meliani, and J. Vaucher, "SSJ: A framework for stochastic simulation in Java", in *Proceedings of the 2002 Winter Simulation Conference*, E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, Eds., Available at `http://simul.iro.umontreal.ca/ssj/indexe.html`, IEEE Press, 2002, pp. 234–242.

[40] C. Lemieux, M. Cieslak, and K. Luttmer, "RandQMC user's guide", 2002.

[41] P. L'Ecuyer and D. Munger, "Latticebuilder: A general software tool for constructing rank-1 lattice rules", *ACM Transactions on Mathematical Software*, 2016.

[42] D. P. Kroese, T. Brereton, T. Taimre, and Z. I. Botev, "Why the Monte Carlo method is so important today", *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 6, no. 6, pp. 386–392, 2014.

[43] A. Srinivasan, "Parallel and distributed computing issues in pricing financial derivatives through quasi Monte Carlo", in *International Parallel and Distributed Processing Symposium., Proceedings, IPDPS 2002, Abstracts and CD-ROM*, IEEE, 2001, 6–pp.

[44] E. de Doncker and A. Almulihi, "Adaptive Task Partitioning for Bayesian Applications", in *Computational Science and Computational Intelligence (CSCI), 2016 International Conference on*, IEEE, 2016, pp. 572–577.

[45] M. Evans and T. Swartz, "Methods for approximating integrals in statistics with special emphasis on Bayesian integration problems", *Statistical Science*, vol. 10, pp. 254–272, 1995.

[46] A. Genz and R. E. Kass, "An application of subregion adaptive numerical integration to a Bayesian inference problem", *Computing Science and Statistics*, vol. 23, pp. 441–444, 1991.

[47] A. Genz, R. E. Kass, and A. Erkanli, "Adapting subregion adaptive integration software to problems in Bayesian inference", *Computing Science and Statistics*, vol. 24, pp. 179–181, 1992.

[48] A. Genz and R. Kass. (1997). BAYESPACK: A collection of numerical integration software for Bayesian analysis, [Online]. Available: `http://www.sci.wsu.edu/math/faculty/genz/homepage`.

[49] ——, "Subregion adaptive integration of functions having a dominant peak", *J. Comp. Graph. Stat.*, vol. 6, pp. 92–111, 1997.

[50] ParInt, http://www.cs.wmich.edu/parint.

[51] E. deDoncker, A. Gupta, J. Ball, P. Ealy, and A. Genz, "Parint: A software package for parallel integration", in *Proceedings of the 10th international conference on Supercomputing*, ACM, 1996, pp. 149–156.

[52] E. de Doncker, R. Zanny, K. Kaugars, and L. Cucos, "Performance and irregular behavior of adaptive task partitioning", in *Lecture Notes in Computer Science*, vol. 2074, Springer-Verlag, 2001, pp. 118–127.

[53] E. De Doncker, A. Gupta, P Ealy, J Liu, A Sureka, and A. Genz, "Use of parint for parallel computation of statistics integrals", in *Computing Science and Statistics*, Citeseer, 1996.

[54] P. J. Davis and P. Rabinowitz, *Methods of Numerical Integration*. Academic Press, New York, 1984.

[55] R. Piessens, E. de Doncker, C. W. Überhuber, and D. K. Kahaner, *QUADPACK, A Subroutine Package for Automatic Integration*, ser. Springer Series in Computational Mathematics. Springer-Verlag, 1983, vol. 1.

[56] A. Genz and A. Malik, "An adaptive algorithm for numerical integration over an n-dimensional rectangular region", *Journal of Computational and Applied Mathematics*, vol. 6, pp. 295–302, 1980.

[57] ——, "An embedded family of multidimensional integration rules", *SIAM J. Numer. Anal.*, vol. 20, pp. 580–588, 1983.

[58] A. Genz and R. Cools, "An adaptive numerical cubature algorithm for simplices", *ACM Trans. Math. Soft.*, vol. 29, pp. 297–308, 2003.

[59] A. Genz, "An adaptive numerical integration algorithm for simplices", in *Lecture Notes in Computer Science*, N. A. Sherwani, E. de Doncker, and J. A. Kapenga, Eds., vol. 507, 1990, pp. 279–285.

[60] A. Grundmann and H. Möller, "Invariant integration formulas for the n-simplex by combinatorial methods", *SIAM J. Numer. Anal.*, vol. 15, pp. 282–290, 1978.

[61] E. de Doncker, "New Euler-Maclaurin expansions and their applications to quadrature over the s-dimensional simplex", *Math. Comp.*, vol. 33, pp. 1003–1018, 1979.

[62] I. Sloan and S. Joe, *Lattice Methods for Multiple Integration*. Oxford University Press, Oxford, 1994.

[63] N. M. Korobov, *Number theoretic methods in approximate analysis*, 1963.

[64] N. M. Korobov, "The approximate computation of multiple integrals", *Doklady Akademii Nauk SSSR*, vol. 124, pp. 1207–1210, 1959, (Russian).

[65] D. Nuyens, "Fast construction of good lattice rules", PhD thesis, Katholieke Universiteit Leuven, 2007.

[66] F. J. Hickernell, "Lattice rules: How well do they measure up?", in *Random and quasi-random point sets*, Springer, 1998, pp. 109–166.

[67] I. H. Sloan and H. Woźniakowski, "When are quasi-Monte Carlo algorithms efficient for high dimensional integrals?", *Journal of Complexity*, vol. 14, no. 1, pp. 1–33, 1998.

[68] M. Chen, "On the solution of circulant linear systems", *SIAM Journal on Numerical Analysis*, vol. 24, no. 3, pp. 668–683, 1987.

[69] D. Nuyens and R. Cools, "Fast component-by-component construction, a reprise for different ernels", in *Monte Carlo and Quasi-Monte Carlo Methods 2004*, Springer, 2006, pp. 373–387.

[70] A. Almulihi and E. de Doncker, "Accelerating High-Dimensional Integration using Lattice Rules on GPUs", in *Computational Science and Computational Intelligence (CSCI), 2017 International Conference on*, IEEE Computer Society, 2017, pp. 1588–1593.

[71] E de Doncker, A Almulihi, and F Yuasa, "High-speed evaluation of loop integrals using lattice rules", in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1085, 2018, p. 052 005.

[72] E De Doncker, A Almulihi, and F Yuasa, "Transformed lattice rules for feynman loop integrals on gpus", in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1136, 2018, p. 012 002.

[73] E de Doncker, F Yuasa, and A Almulihi, "Efficient GPU Integration for Multi-loop Feynman Diagrams with Massless Internal Lines", in *Springer Lecture Notes in Computer Science (To Appear)*, 2019.

[74] NVIDIA. (2012). Tesla K20 graphics processing unit (GPU) accelerator, [Online]. Available: `https://www.nvidia.com/content/PDF/kepler/Tesla-K20-Passive-BD-06455-001-v05.pdf` (visited on 05/20/2017).

[75] W. Kahan, "Further remarks on reducing truncation errors", *Comm. ACM*, vol. 8, p. 40, 1965.

[76] J. Sanders and E. Kandrot, *CUDA by Example - An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011.

[77] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: As easy as 1, 2, 3", in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE, 2011, pp. 1–12.

[78] NVIDIA. (2017). Cuda Toolkit Documentation - cuRAND, [Online]. Available: `https://developer.nvidia.com/cuda-downloads` (visited on 05/20/2017).

[79] A. Sidi, "A new variable transformation for numerical integration", in *Numerical Integration IV, International Series of Numerical Mathematics*, 1993, pp. 359–373.

[80] ——, "Extension of a class of periodizing transformations for numerical integration", *Math. Comp.*, vol. 75, pp. 327–343, 2006.

[81] S. Weinzierl, "Introduction to feynman integrals", *Geometric and topological methods for quantum field theory*, pp. 144–187, 2010.

[82] S. Joe and I. H. Sloan, "Implementation of a lattice method for numerical multiple integration", *ACM Transactions on Mathematical Software (TOMS)*, vol. 19, no. 4, pp. 523–545, 1993.

[83] F. Y. Kuo and S. Joe, "Component-by-component construction of good lattice rules with a composite number of points", *Journal of Complexity*, vol. 18, no. 4, pp. 943–976, 2002.

[84] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT", in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*, IEEE, vol. 3, 1998, pp. 1381–1384.

[85] R. Cools, F. Y. Kuo, and D. Nuyens, "Constructing embedded lattice rules for multivariate integration", *SIAM Journal on Scientific Computing*, vol. 28, no. 6, pp. 2162–2188, 2006.

[86] J. C. Schatzman, "Accuracy of the discrete fourier transform and the fast fourier transform", *SIAM Journal on Scientific Computing*, vol. 17, no. 5, pp. 1150–1166, 1996.

[87] W. M. Gentleman and G. Sande, "Fast fourier transforms: For fun and profit", in *Proceedings of the November 7-10, 1966, fall joint computer conference*, ACM, 1966, pp. 563–578.

[88] M. Tasche and H. Zeuner, "Improved roundoff error analysis for precomputed twiddle factors", *Journal of Computational Analysis and Applications*, vol. 4, no. 1, pp. 1–18, 2002.

[89] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[90] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series", *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[91] L. Bluestein, "A linear filtering approach to the computation of discrete Fourier transform", *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 4, pp. 451–455, 1970.

[92] NVIDIA. (2018). Cufft, [Online]. Available: http://docs.nvidia.com/cuda/cufft/index.html (visited on 04/09/2018).

APPENDICES

A. Generator vectors

Table A.1: Generator vectors z for various d and n.

d	n	Z
3	10^6	1, 419868, 1094386
3	10^7	1, 3675449, 4456704
3	10^8	1, 38278307, 43388112
3	2×10^8	1, 76384079, 90104983
3	35×10^8	1, 146940205, 127904721
4	10^6	1, 292962, 79173, 403401
4	10^7	1, 2928962, 1859617, 3250721
4	10^8	1, 38278307, 43388112, 5988368
4	2×10^8	1, 76384079, 90104983, 20251157
4	35×10^8	1, 146940205, 127904721, 59228284
5	10^6	1, 419868, 109438, 337057, 135243
5	10^7	1, 3675449, 4456704, 3864450, 4934306
5	10^8	1, 38278307, 43388112, 48206750, 45950883
5	2×10^8	1, 76384079, 93229505, 12830863, 56635299
5	35×10^8	1, 133694234, 154078417, 9896546, 125929561
6	10^6	1, 292962, 79173, 403401, 59097, 255500
6	10^7	1, 2928962, 1859617, 4304839, 2689131, 2592686
6	10^8	1, 38278307, 43388112, 48206750, 45950883, 6360155
6	2×10^8	1, 53716462, 63701826, 51044020, 15378683, 37953591
6	35×10^8	1, 146940205, 127904721, 27682752, 91275310, 62729638
7	10^6	1, 419868, 109438, 337057, 135243, 47828, 145504
7	10^7	1, 2928962, 1859617, 4304839, 4474033, 273363, 1426318
7	10^8	1, 41883906, 22682973, 44229424, 29466837, 15518263, 42112409
7	2×10^8	1, 76384079, 93229505, 12830863, 56635299, 54395013, 90891159
7	35×10^8	1, 146940205, 127904721, 27682752, 91275310, 62729638, 70372598
8	10^6	1, 292962, 79173, 403401, 59097, 255500, 14967, 191964
8	10^7	1, 3675449, 4456704, 3864450, 1127878, 144000, 893683, 1090872
8	10^8	1, 38278307, 43388112, 48206750, 45950883, 6360155, 7372151, 19169705
8	2×10^8	1, 53716462, 63701826, 51044020, 28591915, 54939628, 40142478, 52782023
8	35×10^8	1, 146940205, 127904721, 27682752, 91275310, 162555981, 69228497, 128037316

(Table 1 cont.) Generator vectors z for various d and n.

d	n	Z
9	10^6	1, 292962, 229698, 326198, 246988, 447010, 170157, 104406, 499703
9	10^7	1, 3675449, 2181023, 834814, 4235339, 3173487, 171907, 735463, 2446815
9	10^8	1, 38278307, 43388112, 48206750, 45950883, 6360155, 7372151, 19169705, 1149527
9	2×10^8	1, 76384079, 93229505, 12830863, 66233237, 7041901, 10846592, 88109201, 68680388
9	35×10^8	1, 146940205, 127904721, 27682752, 91275310, 162555981, 69228497, 128037316, 123676276
10	10^6	1, 292962, 229698, 326198, 246988, 447010, 170157, 104406, 145823, 425870
10	10^7	1, 3675449, 2181023, 834814, 4235339, 3173487, 171907, 735463, 2446815, 1818771
10	10^8	1, 41883906, 22682973, 44229424, 29466837, 8176047, 49462874, 1162485, 46871525, 36107330
10	2×10^8	1, 53716462, 63701826, 51044020, 67272988, 11162620, 9925795, 63388095, 54986324, 19264518
10	35×10^8	1, 146940205, 127904721, 108579162, 30890237, 166719091, 28591254, 114681466, 54322685, 136274288

B. Embedded sequence code

```
int main( void ) {
  double  q1[51], q2[51], *partial_q;
  double  *dev_partial_q;
  double exact;
  int i,j,l,w,k[51];
  double val = 0.0, dd, pp, E = 0.0;
  int *dev_k;
  // Allocate memory on GPU and CPU
  cudaMalloc( (void**)&dev_k, dim * sizeof(int) );
  partial_q = (double*)malloc( blocksPerGrid*sizeof(double));
  cudaMalloc( (void**)&dev_partial_q, blocksPerGrid*sizeof(double));
  q1[1] = 0.0;
  for(i=1; i<=dim; i++){ // Initialize k, q1 and q2 arrays
    k[i-1] = 0;
    q1[i+1] = 0.0;
    q2[i] = 0.0;
  }
  k[0] = -1;
  l = 1;
  w = 0;
  val = 0.0;
  while (l <= dim){
    if(k[l-1] < n-1){
      k[l-1] = k[l-1] + 1;
      l = 1;
```

```
    val = 0.0;
    // Transfer k to GPU
    cudaMemcpy( dev_k, k, dim * sizeof(int),
       cudaMemcpyHostToDevice);
    // Call kernel
    lat1<<<blocksPerGrid,threadsPerBlock>>>(dev_partial_q, dev_k )
       ;
    // Transfer partial_q from GPU
    cudaMemcpy( partial_q, dev_partial_q, blocksPerGrid*sizeof(
double), cudaMemcpyDeviceToHost );
    // Finalize accumulations
    for (j=0; j<blocksPerGrid; j++) {
      val += partial_q[j];
    }
    for (i=w; i<=dim; i++){
      q1[i+1] = q1[i+1] + val;
    }
    for (i=1; i<=dim; i++){
      if (k[i-1]==0)
      q2[i] = q2[i] + val;
    }
    if (w == 0)
      w = 1;
  }
  else{
    k[l-1] = 0;
```

```
    l++;
    if (w<l)
    w = l;
  }
}
dd = (double) n;
for (i=0; i<=dim; i++){
  pp = (double) m * pow (dd, (double) i);
  q1[i+1] = q1[i+1]/pp;
}
// Compute error estimate
E = 0.0;
pp = (double) m * pow (dd, (double) dim-1);
for (i=1; i<=dim; i++){
  q2[i] = (double) q2[i] / pp;
  E = E + (q1[dim+1] - q2[i]) * (q1[dim+1] - q2[i]);
}
E = sqrt(E/(double) dim);
exact = 1; //exact value of the function (example)


printf( "threadsPerBlock = %d",threadsPerBlock);
printf( " blocksPerGrid = %d\n",blocksPerGrid);
printf( "For m = %d , n = %d\n", m,n );

for (i=1; i<=dim; i++){
```

```
  printf( " Qcopy%d = %23.16le    Qsup%d = %23.16le\n", i, q1[i+1],i
     , q2[i] );
  }
  printf( " Estimated Error = %23.16le\n ", E );
  printf( "Error = %23.16le Result = %23.16le Exact = %23.16le\n",
     fabs(exact-q1[dim+1]),q1[dim+1],exact );
  printf( "Elapsed time par. :  %le ms\n", 1000*e);

  // free memory on GPU
  HANDLE_ERROR( cudaFree( dev_partial_q ) );

  // free memory on CPU
  HANDLE_ERROR(cudaFree( dev_k) );
  free( partial_q );
}
```