12-2019

Scalable Algorithms and Hybrid Parallelization Strategies for Multivariate Integration with ParAdapt and CUDA

Omofolakunmi Elizabeth Olagbemi
Western Michigan University, oeogunlesi@yahoo.com

Scalable Algorithms and Hybrid Parallelization Strategies for Multivariate Integration with ParAdapt and CUDA

by

Omofolakunmi Elizabeth Olagbemi

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
December 2019

Doctoral Committee:

Elise H. de Doncker, Ph.D., Chair
John A. Kapenga, Ph.D.,
Joseph W. McKean, Ph.D.,

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my supervisor, Dr. Elise de Doncker for the invaluable support, guidance and mentoring she provided to me all through the dissertation process. Her insights, encouragement and great patience have been very instrumental in assisting me with achieving this goal of successfully completing my PhD program. It has been an honor and a privilege to have her as my supervisor.

I would also like to express my appreciation to the other members of my dissertation committee - Dr. John Kapenga and Dr. Joseph McKean - for their suggestions, their feedback on my work and their support.

My mother has always been a great support and source of encouragement to me, and this has continued through the process of working on my dissertation. No sacrifice is too great for her to make to provide whatever assistance she anticipates I might need. I am deeply grateful to her.

My husband and two sons have walked this road with me and have provided much-needed support and encouragement along the way. I am very thankful to them and for them.

Last but by no means least, I am eternally grateful to God for seeing me through this process, for bringing it to a successful completion, and for helping me to keep going even in the midst of discouragement when progress in the research was slow and at times seemed to remain at a standstill.

Omofolakunmi Elizabeth Olagbemi

Scalable Algorithms and Hybrid Parallelization Strategies for Multivariate Integration with ParAdapt and CUDA

Omofolakunmi Elizabeth Olagbemi, Ph.D.

Western Michigan University, 2019

The evaluation of numerical integrals finds applications in fields such as High Energy Physics, Bayesian Statistics, Stochastic Geometry, Molecular Modeling and Medical Physics. The erratic behavior of some integrands due to singularities, peaks, or ridges in the integration region suggests the need for reliable algorithms and software that not only provide an estimation of the integral with a level of accuracy acceptable to the user, but also perform this task in a timely manner. We developed ParAdapt, a numerical integration software based on a classic global adaptive strategy, which employs Graphical Processing Units (GPUs) in providing integral evaluations. Specifically, ParAdapt applies adaptive region partitioning strategies developed for efficient integration and mapping to GPUs. The resulting methods render the framework of the classic global adaptive scheme suitable for general functions in moderate dimensions, say 10 to 25. The algorithms presented have been determined to be scalable as evidenced by speedup values in the double and triple digits up to very large numbers of subdivisions. An analysis of the various partitioning and parallelization strategies is given.

TABLE OF CONTENTS

Table of Contents—Continued

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CS	**C**omputer **S**cience
ParInt	**Par**allel **Int**egration
HPC	**H**igh **P**erformance **C**omputing
GPU	**G**raphical **P**rocessing **U**nit
MPI	**M**essage **P**assing **I**nterface

CHAPTER 1

INTRODUCTION

The evaluation of numerical integrals finds applications in many fields including high energy physics [1]–[4] (for modeling of particle interactions), Bayesian statistics [5], computational chemistry, medicine [6], [7] (for example, in the computation of radiation doses for cancer treatment), stochastic geometry, computer graphics/visualization [8]–[11] and behavioral psychology [12]. The importance of the accuracy of results obtained and the timeliness of computing the results of integrals is evident and therefore is of paramount importance in these fields. This therefore provides an area of research that is heavily patronized by researchers, and several numerical integration software applications are available for public use.

The nature of the integration process is such that the number of evaluation (sample) points increases exponentially with the number of dimensions. In this dissertation, the computing power of GPUs (Graphical Processing Units) is harnessed to speed up the evaluations while still achieving accuracy levels that compare favorably with those found in the literature. Results for sample integrals are presented in Chapter 4. These are derived from "real world" applications, or implemented in a manner that simulates real world problems. The resulting integral estimates and speedup values achieved are given.

Our numerical integration software solution, ParAdapt, from which the results in this dissertation were obtained, is derived from ADAPT by Alan Genz [13]. Chapter 2 includes an outline of ADAPT and its subroutines. Comparisons were made with another multivariate integration software ParInt [14].

1.1. Automatic Integration

ADAPT as well as its predecessor HALF by P. van Dooren and L. de Ridder [15], its successors including DCUHRE [16], [17], and the parallel multivariate integration package ParInt [14] adhere to the paradigm of automatic integration.

As illustrated in Fig. 1.1, automatic multivariate integration can be represented as a black box to which the user supplies an integrand function f, the requested accuracy or tolerated error and a region $\mathcal{D}$ over which the integral is to be evaluated. An integral approximation

$$Q \approx I = \int_{\mathcal{D}} f(\vec{x}) d\vec{x} \tag{1.1}$$

and an error estimate E are determined. The computation aims to satisfy an error criterion such as

$$|Q - I| \leq E \leq max\epsilon_a, \epsilon_r |I| \tag{1.2}$$

where the parameters ϵ_a and ϵ_r are user-specified absolute and relative error tolerances.

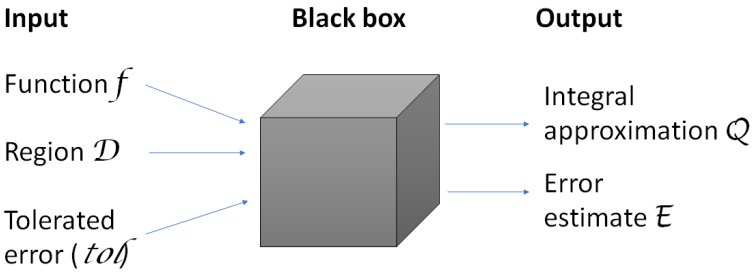

Figure 1.1: Automatic multivariate integration black box

In order to ensure that the function evaluations do not continue indefinitely, the user can specify a limit on the number of function evaluations. It is possible that for a given integral, the limit is reached and yet a result that falls within the given tolerated error may

not have been obtained. This could be due to the nature of the integral itself (perhaps due to integrand singularities, peaks, ridges or discontinuities in the integration region), the effect of round-off errors, the limits on machine precision, or a combination of these [14].

1.2. ParAdapt

With integrals requiring hundreds of millions or more function evaluations (which could run into hours or weeks of program execution time) for results of acceptable accuracy to be obtained, a parallel integration program becomes a necessity to significantly reduce the execution time. While other parallel implementations of numerical integration algorithms exist, ParAdapt, which like ADAPT applies the adaptive strategy, is a parallel multivariate numerical integration solution that harnesses the superior power of GPUs for the parallel execution. ParAdapt utilizes a set of subroutines from the ADAPT code (see Chapter 2) to generate the points and weights for the integration rule, which are then utilized in a CUDA [18] kernel to integrate over the subregions as they are generated. A number of subdivision strategies were tested in ParAdapt and compared for efficiency and accuracy of the results. The GPU kernel is able to evaluate varying numbers of subregions simultaneously: 1, 2, 4, 8, 16 subregions (where currently the number of subregions passed to the kernel must be a power of 2). The parallelization strategy is designed so that, even though many threads work together in evaluating multiple subregions simultaneously, the contributions for each individual subregion are eventually combined, separately from those of other subregions that were evaluated on the GPU at the same time. In the host code, the subregions must be stored on the priority queue with their pertinent information concerning location, results and error estimates in order to allow for subsequent retrieval. The strategies employed in ParAdapt are described in greater detail in Chapter 3 of this dissertation.

1.3. Statement of the Problem

Applications in various fields require the computation of integrals in increasing dimensions, to address the need for the current levels of complexity in sophisticated models. Adaptive integration methods have traditionally allowed considerable flexibility with regard to the integrand behavior. However, in practice, this applicability is restricted to dimensions through, say 12. This is due to the large number of regions needed to partition a multivariate domain effectively, and to the exponential increase of the number of points in the integration rule used locally over each subregion.

In this research, we aim to devise an approach to increase the upper threshold on the dimension for adaptive algorithms, by harnessing the power of GPUs for the computational-intensive local integrations. The method should scale (with sustained speedup) over very large numbers of subdivisions, required for real-world applications.

1.4. Structure of the Dissertation

The content of the next four chapters of this dissertation is laid out as follows. Chapter 2 briefly reviews alternative approaches to numerical integration, provides a more in-depth description of the adaptive approach, and compares the local adaptive strategy to the global approach (which is used in ParAdapt). It also provides details concerning the integration rule in ParAdapt. The subdivision and parallelization strategies with GPU mappings and reductions are detailed in chapter 3. Chapter 4 presents results obtained for sample problems with speedup values achieved in ParAdapt using the strategies developed, and compares the results for a Bayesian integral application and a Feynman loop integral with other published results. The material in Chapters 3 and 4 have been submitted for publication. Chapter 5 gives concluding remarks and discusses future work that flows from this dissertation.

CHAPTER 2

LITERATURE REVIEW

2.1. Numerical Integration Methods

Several approaches exist for the numerical computation of integrals, with, as might be expected, multi-dimensional integrals requiring more complex and sophisticated algorithms due to the increased number of dimensions. Some of these approaches are: Monte Carlo, quasi-Monte Carlo, non-adaptive, and adaptive cubature methods. Unless otherwise stated, the integration will be over the d-dimensional unit cube $\mathcal{C}_d$,:

$$\mathcal{C}_d = \{\vec{x} = (x_1, x_2, \ldots, x_d) \mid 0 \leq x_i \leq 1 \text{ for } 1 \leq i \leq d\} \tag{2.1}$$

2.1.1. Monte Carlo (MC)

In this section we consider the one-dimensional integral:

$$I = \int_0^1 f(x)dx \tag{2.2}$$

although similar properties hold for multiple dimensions. The Monte Carlo integration approach approximates the integral I by

$$Q_n f = \frac{1}{n} \sum_{i=1}^{n} f(x_i) \tag{2.3}$$

where each x_i represents a random value (drawn from a uniform distribution) in the interval [0,1], and n is the number of sample points. In order to simulate the random selection of points required in the Monte Carlo method, a pseudo-random number generator is used. Increasing the number of points n leads to convergence of the Monte Carlo integral approximation under general conditions. The *Central Limit Theorem* shows that the rate of convergence obtained with Monte Carlo is $O(\sigma(f)/\sqrt{n})$ as $n \to \infty$, where $\sigma(f)$ represents the standard deviation of f. This indicates that the rate of convergence of the Monte Carlo method is not dependent on the dimension of the integral; therefore, a similar level of accuracy can be expected for integrals with similar standard deviation, irrespective of the number of dimensions. Note that the definition of the variance imposes that f is square integrable; however, this is not required for convergence (in probability) of MC [19]. MC is well-suited to high dimensions but has a slow rate of convergence [20].

2.1.2. Quasi-Monte Carlo (QMC)

The Quasi-Monte Carlo approach is similar to Monte Carlo except that in this case, instead of using a set of points selected randomly in the estimation of the integral, a low-discrepancy sequence or equidistributed sequence of points is used. From empirical results, the quasi-Monte Carlo approach achieves a convergence rate of $O(1/n)$ for many (fairly smooth) integrand types and is therefore well-suited to high dimensions for these classes of integrands [20].

2.1.3. Methods based on Interpolatory Cubature Rules

Interpolatory cubature rules [21] can be represented by the formula of approximate integration

$$Q_m f = \int_a^b k(x)f(x)dx \approx \sum_{i=1}^{m} w_i f(x_i) \tag{2.4}$$

such that an exact result can be obtained for all multivariate polynomials up to degree p (i.e., $Ip = Q_m p$ for all polynomials of degree p). The number m of points in the rule is dependent on the formula used and generally increases rapidly as the number of dimensions and desired degree p increases. The points at which the integrand is evaluated (i.e., $x_1, x_2, ..., x_m$)) are fixed and distinct, and lie in the interval $[a, b]$, where a and b are predetermined. For our purposes in this study, as all integration is over the $d-$dimensional unit cube, all points lie in the interval [0,1].

The weights (w_is) in the formula can be determined using either of two approaches, both of which produce the same set of weights. The first approach is to interpolate to the function $f(x)$ at the given m points by a polynomial of degree $p-1$ (which is solved exactly by a rule of degree p). Then the polynomial is integrated and expressed in the form of (2.4). Alternatively, the constants $w_1, w_2, ..., w_m$ can be selected in such a way that the error

$$E(f) = \int_a^b k(x)f(x)dx - \sum_{i=1}^{m} w_i f(x_i) \tag{2.5}$$

is zero for $f(x) = 1, x, x^2, ..., x^{m-1}$. The values of the w_is are determined by solving a system of equations in $f(x)$ [19]. A proof showing that these two approaches yield the same set of weights, along with further details on the determination of the weights, is provided in [19].

Following from the definition in (2.4), the expectation is that integration methods based

on interpolatory cubature rules will be most efficient when applied to functions that can be approximated by low degree polynomials. The definition also suggests that these methods are better suited to integrals in low to moderate dimensions. Schürer, in [20], gives a list of cubature rules in Table 1 of the article. One of the rules included is the degree 7 rule by Genz and Malik [22], which is used in ADAPT and, by extension, in ParAdapt. Also listed in the table are the orders of magnitude of the number of points in each rule, with Genz and Malik's degree 7 rule of order $O(2^d)$. Another category of data listed in the table is the sum of the absolute values of the weights w_i used in computing the formula shown in (2.4). This sum equates to 1 where the weights in the corresponding rule are all positive, and increases in value for rules with negative weights. The value of this sum can be used to rate the quality of a specific rule.

The number of points for any given rule is fixed for a dimension d. Interpolatory cubature rule-based methods therefore split the integration region into k subregions, each of which is evaluated at m points giving rise to a total of km points. Two integration methods apply this region partitioning approach, albeit in different ways: non-adaptive and adaptive [20]. Empirical tests performed using Genz's package for testing multidimensional integration routines [23] indicate that cubature rule-based algorithms (like the adaptive and non-adaptive methods) can produce results with higher accuracy than the quasi-Monte Carlo approach for up to 100 dimensions [20].

2.1.3.1. **Non-adaptive cubature methods.** In the non-adaptive approach, the integral is evaluated using a fixed set of points, which is independent of the nature of the integrand. Specifically, the integration region is split into k subregions of equal width and volume, in a predetermined manner. For example, a region can be split $\lceil \sqrt[d]{k} \rceil$ or $\lfloor \sqrt[d]{k} \rfloor$ times along each

coordinate axis. Applying a scaled version of Q_m to each subregion and accumulating results from all subregions yields an approximation to the integral [20].

2.1.3.2. **Adaptive methods.** Like the non-adaptive approach, the adaptive approach involves approximating a given integral using a fixed set of points, and splitting or partitioning of one or more regions may be required if the accuracy of the approximation is unsatisfactory. The difference between the two approaches arises from how the region subdivisions are performed. In the adaptive approach, the subdivision pattern (or order of performing the subdivisions) varies from one integral to the other, as the choice as to which subregion and how to subdivide is determined by the areas in the domain where the integral is most "difficult" to integrate.

Adaptive methods utilize a combination of integration rules in providing an approximation to an integral as well as an estimate of the absolute error in the result. To obtain an estimate of the error, the common approach is to use two (or more) interpolatory rules (where a rule can be seen as a predetermined set of points within the integration region, where the integrand will be evaluated). When a subdivision is performed, the predetermined set of points is scaled to the subregion. Assuming two rules are used in the error estimation, these can be expressed as:

$$\begin{aligned} Q^{(1)}f &= \sum_{i=1}^{m^{(1)}} w_j^{(1)} f(x_i^{(1)}) \text{ and} \\ Q^{(2)}f &= \sum_{i=1}^{m^{(2)}} w_j^{(2)} f(x_i^{(2)}) \end{aligned} \tag{2.6}$$

where $degree\ Q^{(1)} > degree\ Q^{(2)}$. In this scenario, $Q^{(1)}$, being of a higher degree, will be used

to approximate the integral while the error estimate Ef can be computed as $|Q^{(1)}f - Q^{(2)}f|$. With the rule $Q^{(2)}$ being of a lower degree (and consequently $m^{(2)}$ being smaller than $m^{(1)}$), the additional computations required to estimate the error may come at a small additional cost. In some cases, the points used in the error estimation process are a subset of those used in approximating the integral: these are referred to as *embedded rules*. Where this is the case, no additional integrand evaluations are required to estimate the integral since a subset of evaluations for the integral approximation is used in the error estimation [20].

2.2. Adaptive Integration Strategies

A number of adaptive strategies exist [24] but two major strategies are the local and the global adaptive strategies. Other strategies include the adaptive scheme of O'Hara and Smith [25] and the Cranley-Patterson Scheme [26]. Of the two major strategies used (local and global), the local strategy was the one first used for adaptive integration. It is described in detail in numerical analysis texts such as those by Forsythe et al. [27], Shampine and Allen [28], and in Davis and Rabinowitz [19].

2.2.1. Local Adaptive Strategy

In the local strategy, the result $Q(\Delta_0)$ and error estimate $E(\Delta_0)$ are computed over the given integration region Δ_0. If the accuracy at this point is acceptable given the user's required tolerated error, then the program terminates. Otherwise, the initial region is subdivided into p subregions, Δ_{11}, ..., Δ_{1p}, which are then stored on a list L in some predetermined order. This is now stage 1 of the strategy. At stage n, some subregions on the list L are yet to be evaluated. At this stage, the value of the integral and its error estimate have been computed over those subregions that have satisfied the error criterion, which may be of the

form

$$E(\Delta) \leq \epsilon_a \frac{vol(\Delta)}{vol(\Delta_0)}, \tag{2.7}$$

where ϵ_a is the user-specified absolute accuracy, and the required accuracy over Δ is ϵ_a multiplied with the fraction of the region volume to the volume of the given region Δ_0. Each subregion that satisfies the error criterion is discarded and is no longer available. While subregions exist on list L and neither the maximum number of stages nor the maximum number of function evaluations have been reached, a subregion Δ is selected from L according to some criterion and deleted from L, and $Q(\Delta)$ and $E(\Delta)$ are evaluated. If $E(\Delta)$ is satisfactory, $Q(\Delta)$ and $E(\Delta)$ are added to the running integral and error estimate, respectively. Otherwise, Δ is subdivided into p subregions, which are then added to the list L in some order. This process is repeated until one of the following conditions is met: the list L is empty (for a successful computation), the maximum number of stages has been attained, or the maximum number of function evaluations has been reached [19].

2.2.2. Global Adaptive Strategy

The global adaptive strategy has some similarities with the local approach. The initial region is the domain Δ_0. The tolerated error ϵ can be absolute or relative, or a combination of both. At the beginning, $Q(\Delta_0)$ and $E(\Delta_0)$ are both evaluated. If $E(\Delta_0) \leq \epsilon_a$ or $E(\Delta_0) \leq \epsilon_r |Q(\Delta_0)|$, then the desired accuracy has been obtained and the integration process terminates. It also terminates if the maximum number of integrand evaluations has been reached at this point. If neither of those conditions has been met, the initial region Δ_0 is subdivided into p subregions, Δ_{11}, ..., Δ_{1p}, and the global integral $Q = \Sigma_{i=1}^{p} Q(\Delta_i)$ and error estimate $E = \Sigma_{i=1}^{p} E(\Delta_i)$ are computed. A test is performed to determine if one of the termination

criteria has been met, for the accuracy, $\Sigma E(\Delta_{1i}) \leq \epsilon_a$ or $\Sigma E(\Delta_{1i}) \leq \epsilon_r |\Sigma Q(\Delta_{1i})|$, or with respect to the maximum evaluations. If either condition is satisfied, the integration terminates. Otherwise, the subregion with the highest error estimate is subdivided, and $Q(\Delta)$ and $E(\Delta)$ are computed for each of the new subregions.

The sums Q and E are updated, and the termination criteria are tested. The process repeats until a termination criterion is satisfied, i.e., the requested accuracy has been achieved (success) or the limit on the allowed number of function evaluations (or allowed number of subregions) has been reached. The result and error estimates are returned in addition to an error flag indicating that either the program terminated successfully or that it terminated abnormally and failed to reach the required error tolerance [19].

A meta-algorithm for global adaptive integration is depicted in Fig. 2.1 (see, e.g., [20], [29], [30]).

Evaluate initial region and update results
Initialize priority queue with initial region
while (evaluation limit not reached and estimated error too large)
    Retrieve region from priority queue
    Split region
    Evaluate new subregions and update results
    Insert new subregions into priority queue

Figure 2.1: Global adaptive integration meta-algorithm

Global adaptive algorithms are used, e.g., in Quadpack [31] (for 1D integration) and in the parallel multivariate integration package ParInt [14]. Both also include non-adaptive (QMC based) algorithms. Other integration packages include Cubpack [32] and Cuba [33].

As an example, consider the function

$$f(x, y, z) = x^{-0.2} y^{-0.2} z^{-0.2} (x + y + z)^{-0.2} \tag{2.8}$$

which is singular at $x = y = z = 0$. On integrating this function, over the 3-dimensional unit cube, Fig. 2.2 shows how the global adaptive algorithm, through repeated subdivisions, concentrates the integration efforts on subregions in the integration domain that present the most difficulty. The colors in Fig. 2.2 correspond to the estimated error over the subregions, green (low) to red (high), showing the difficult regions at the faces of the cube, and intensifying near the axes and the origin.

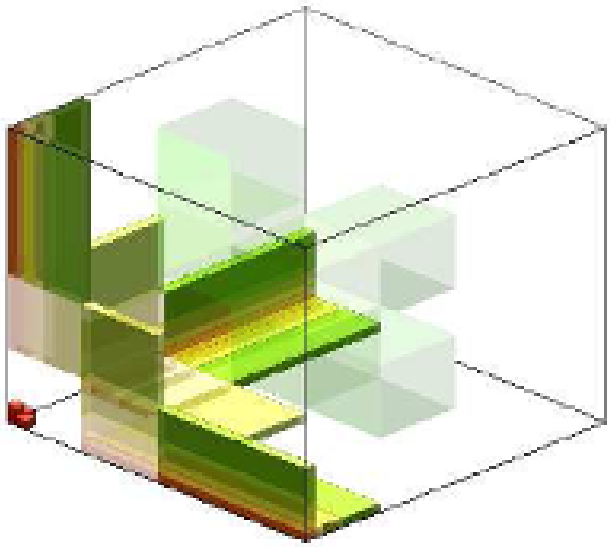

Figure 2.2: Snapshot of computations (subregion evaluations in the $3D$ unit cube) for the integral of (2.8) [10]

2.2.3. Global Strategy versus Local Strategy

One of the advantages of the global strategy over the local is that the former always returns an approximation to the integral, even if the required error tolerance specified by the user has not been met. The local strategy on the other hand may not return an approximation to the integral over the entire region if one of the terminating criteria has been met before all sub-intervals on the list L have been evaluated. However, in the global strategy, since all subregions are typically retained (and not discarded after evaluation as is done in the local

strategy), much computer storage is required to store all the subregions and their respective results and error estimates. This may no longer be as much of a problem since computers now have significantly more storage capacities than was previously the case.

The global strategy requires a priority queue data structure to store the subregions. In one of the earlier globally adaptive algorithms, by van Dooren and de Ridder [15], a list was used to store all the subregions in decreasing order of their respective absolute error estimates. This requires a re-ordering of the subregions anytime new subregions are added to the list, which, as the number of subregions increases, significantly increases the execution time. Following recommendations by Malcolm and Simpson [34], Genz and Malik improved upon this by using a heap instead of a list for storage of the subregions [22]. The worst case complexity for sorting a list of length n is of the order $O(n^2)$, compared to order $O(n \log n)$ for a heap. Even though the time for reordering a heap (of order $O(\log n)$) both when subregions are added and when a subregion is removed from the heap could be significant as the number of subregions increases, the choice of the heap is a significant improvement over the list option.

2.2.4. Adaptive Approach in this Study

The majority of the results presented in this thesis were obtained using ParAdapt. The global adaptive approach is used in ADAPT and retained in ParAdapt. While sequential adaptive strategies are generally considered effective for integral dimensions through about 10 or 12, it is our goal to move this threshold up considerably.

Interpolatory cubature rules were discussed in an earlier section. The next section provides more in-depth details of the integration rules used in ParAdapt.

2.3. Integration Rules in ParAdapt

An integration rule is of polynomial degree p if it provides an exact result for all monomials $x_1^{k_1} x_2^{k_2} ... x_n^{k_n}$ where $\Sigma k_i \leq p$ and fails to provide an exact result for at least one monomial of degree $p+1$. See the surveys by Cools [35], [36]. A large body of work has been devoted to the construction of optimal cubature rules for various multi-dimensional regions, with the goal to use a minimum number of evaluations points for a given polynomial degree.

ParAdapt, derived from ADAPT [13], utilizes a degree 7 rule for approximating the integral, and two degree 5 and one degree 3 comparison rules for error estimation and for computing the fourth order divided differences (which are used to determine the next dimension(s) to be subdivided). The basic rule for ADAPT (as described in [22]) is of the form:

$$\begin{aligned} BF = w_1 F(0,0,...,0) + w_2 \sum F(\lambda_2,0,0,...,0) + w_3 \sum F(\lambda_3,0,0,...,0) \\ + w_4 \sum F(\lambda_4,\lambda_4,0,0,...,0) + w_5 \sum F(\lambda_5,\lambda_5,...,\lambda_5) \end{aligned} \tag{2.9}$$

All sums in the basic rule form are fully symmetric over all permutations of coordinates, including sign changes. Standard techniques described by Stroud [21] were used to find the parameters w_i and λ_i (for values of i ranging from 1 to 5) that make BF a degree seven rule. Genz and Malik in [22] provide a detailed overview of how the parameters are determined.

The fully symmetric basic rule BF was first introduced by Genz and Malik in [22], and is based on the rule used for the adaptive algorithm HALF by van Dooren and de Ridder [15]. A definition of a "fully symmetric rule" was provided by Bernsten et al. in [16] as follows: "A rule for the cube $[-1,1]^n$ is fully symmetric if, whenever the rule contains a point $x = (x_1, x_2, ..., x_n)$ with associated weight w, it contains all points that can be generated

from x by permutations and/or sign changes of the coordinates with the same associated weight." An in-depth review of fully symmetric integration rules was given by Genz and Malik in [37]. Implemented in the integration program DCUHRE, Berntsen et al. [16] explain how the integration rules are used in the integral approximation, with null rules used for the error estimation. They give a table that shows rule options in DCUHRE where each value of the user-specified parameter "key" indicates a different set of fully symmetric integration rules to be applied. The available keys in DCUHRE [16] are options from 0 to 4 (with 0 being the default key value). The sets of rules for keys 3 (of degree 9) and 4 (of degree 7) are based on the work of Genz and Malik in [37]. Berntsen et al. [16] include a table that shows the resulting number of evaluation points for dimensions from 2 to 10 for the rules, based on that work. The rules from [16] are also used for the adaptive integration in ParInt (see the user manual via the "documentation" link in [14]).

ADAPT provides three possible sets of rules from which one option is selected based on the number of dimensions of the integral. The first set of rules is for a one-dimensional integral. The second (of degree 7) is for adaptive integration in dimensions from 2 to 11; in our work, this is extended to 25 dimensions. The third set of rules in ADAPT is for higher dimensions (with significantly fewer points). In this study, ParAdapt uses the second (degree 7) rule, which evaluates the integrand function at

$$N = 2^d + (2 * d * (d + 2)) + 1 \tag{2.10}$$

points where "d" is the dimension. Table 2.1 shows the number of points N used in selected dimensions from 10 to 25.

Table 2.1: Number of Evaluation Points for Selected Dimension (10 to 25)

Dimensions	Number of points
10	1265
14	16833
15	33279
17	131719
19	525087
20	1049457
21	2098119
22	4195361
24	16778465
25	33555783

2.4. ADAPT

ADAPT, available online at [13], is a sequential numerical integration software by Alan Genz [22]. It was written as a Fortran subroutine for the numerical integration of a vector function over a hyper-rectangular region and is based on the global adaptive integration strategy in the subroutine HALF by P. van Dooren and L. de Ridder [15]. We translated the Fortran version of ADAPT to C using the f2c program [38] and ParAdapt was developed from the C version of ADAPT. We implemented integration over the subregions in one CUDA kernel, and the new subdivision strategies are described in Chapter 3.

ADAPT comprises the following subroutines: *adapt*, *bsinit*, *rulnrm*, *adbase*, *basrul*, *fulsum*, *differ* and *trestr*.

- **adapt**: is called by "main" from which it receives the values of the parameters and the main (work) array to store the subregion data resulting from the adaptive integration.

adapt is a driver for *adbase* which it calls after initializing variables and dividing up the work array for storage of different groups of data.

- **bsinit**: computes or determines the basic integration rule points and weights for a degree 7 rule, two degree 5 comparison rules and one degree 3 comparison rule. It also initializes the symmetric sum parameters.

- **rulnrm**: is called by the *bsinit* subroutine. It computes the orthonormalized null rules.

- **adbase**: is the main adaptive integration subroutine. It calls differ to determine in which coordinate direction to perform the subdivision, determines the resulting subregions, which are then passed to subroutine *basrul* for computation of the integral approximations and the corresponding absolute error estimates over each of the subregions. When *adbase* subsequently passes the subregions to the heap routine *trestr*, the absolute error estimates are used for inserting the subregions into the heap. Following this, *adbase* checks for termination using two criteria. First, if the total error estimate over all the regions falls below the user's tolerated error, the program terminates, returning the total integral and error estimate. However, if the total error estimate is still larger than the tolerated error, *adbase* checks whether the accumulated number of function evaluations at that point has exceeded the maximum specified by the user. If it has, the program returns the current integral and absolute error estimates to the user, with an error code indicating that the maximum evaluations set by the user precludes the program from continuing until the total error estimate falls below the tolerated error, and terminates. If neither of the conditions for termination is met, the subregion with the highest error estimate is removed from the heap, passed to *differ*, and subdivided in the direction perpendicular to the direction indicated by *differ*, and

the process outlined above repeats until one or the other of the termination conditions is met.

- **basrul**: performs the basic rule integration and error estimation over the subregions passed to it (from *adbase*) using values returned from a call to *fulsum*.

- **fulsum**: computes fully symmetric basic rule sums that are used by *basrul*.

- **differ**: computes fourth order divided differences for all coordinate directions on the region passed to it and returns the direction with the highest fourth order divided difference. This provides an indication of the level of difficulty of the integrand function in a given direction, and the behavior to be expected after possible subdivision without actually performing the subdivisions and the integration over the subregions. The subdivision is subsequently performed perpendicularly to the direction with the highest fourth order divided difference.

- **trestr**: maintains a max-heap (priority queue), keyed by the absolute error estimate, in which the subregions generated from the initial region are stored. The subregion with the highest error estimate is at the root of the heap and will always be the one to be removed if it is determined that yet another subdivision is required. After the subregion at the root of the heap is removed, a "heapify" procedure is performed to ensure that the heap property is maintained (i.e., the error estimate at any node equals or exceeds that of its children nodes, so that the subregion with the highest error estimate will be at the root of the heap).

- **main**: allocates the required memory, sets the integration parameters, calls adapt to begin the integration, and prints the result, error estimate and error code, which is

based on how the integration terminated. The user supplies a *main* program to call *adapt*, and the integrand specification in f:

- **f**: defines the integrand function. If any transformations are required to handle integrand difficulties such as singularities, in the integration region, they are part of this function.

2.5. The ParInt Software

Some results in Chapter 4 were compared with results from ParInt (PARallel INTegration), a software package for automatic multivariate integration using a user-specified number of processors. ParInt also served as a yardstick against which to assess the accuracy of our results for some of the problems presented. The software was developed by a group of faculty members and Computer Science students at WMU, led by Dr. Elise de Doncker. The project was funded through National Science Foundation (NSF) grants [1] [2].

ParInt, which is layered over MPI (Message Passing Interface), is used for integration over hyper-rectangular and simplex regions. The integration is done using one of a number of algorithms available in the software package. A global adaptive region-subdivision algorithm (see 2.1) repeatedly subdivides the regions with higher error estimates and evaluates the integrand until either the limit on the number of function evaluations has been reached or the estimated error falls below the user-specified tolerated error. The adaptive algorithm in ParInt, which is the preferred option for integrals in low to moderate dimensions, is implemented with load-balancing to ensure optimal and efficient use of the worker processes. Other (non-adaptive) integration options available in ParInt are quasi-Monte Carlo (QMC)

[1] NSF ACI-0000442
[2] NSF Award 1126438

and Monte Carlo (MC) methods.The QMC method is typically used for fairly smooth functions possibly in higher dimensions, while the MC method is suitable for integrals of an erratic integrand function or a high dimension, or both. Like the adaptive method, both QMC and MC are implemented with load balancing [39].

CHAPTER 3

METHODOLOGY

3.1. Subdivision Strategies used in ParAdapt

The global adaptive strategy of Fig. 2.1 produces (local) integral and error estimates over each subregion and maintains a total integral and error estimate over the entire domain. A priority queue such as a max-heap keyed with the local error estimates allows selecting the region with the largest error estimate to be subdivided next; its subregions are evaluated and inserted into the heap. The adaptive partitioning succeeds in creating hot-spots in the vicinity of singularities and other irregular integrand behavior.

Apart from the region selection, important components of the (hyper-rectangular) partitioning strategy involve: into how many subregions and in which directions to subdivide. While HALF and ADAPT use region bisections, subdivisions into more than two subregions at a time may seem promising to partition regions more effectively in higher dimensions.

In ADAPT, subroutine *differ* computes fourth order divided differences of the integrand function in all coordinate directions over a given subregion. These are used to assign a relative degree of difficulty to the coordinate directions, so that the largest divided difference designates the most difficult direction, and the next subdivision of the region is destined to be perpendicular to this direction.

ADAPT applies its main rule (of polynomial degree of accuracy 7) and three comparison rules (for error estimation) over each subregion. The number of points (N) used for the local integration is exponential with respect to the integral dimension (d): $N = 2^d + 2d(d+2) + 1$. (see (2.10)).

Table 2.1 lists the number of points used for a range of dimensions. It can be seen that, e.g., a 10-dimensional integration uses 1265 points per subregion. Thus utilizing, say, a K20 GPU (with 2496 cores) for a single region evaluation would be an under-utilization of the device. ParAdapt can apply one of several subdivision strategies to generate two ($2R$), four ($4R$), eight ($8R$) or sixteen ($16R$) subregions, which are then passed to the GPU kernel to be evaluated simultaneously. For $4R$, three different subdivision strategies are outlined subsequently, followed by explanations of the subdivision strategies for $8R$ and $16R$.

$4R_0$ (*subdivision into four regions in the same coordinate direction*): In this strategy, the region is subdivided into four regions in the same direction, corresponding to the value of the variable *divaxn*, which is returned from a call to *differ*. The region to be subdivided is passed to *differ* as one of the parameters and *differ* computes fourth order divided differences to determine the direction with the largest fourth divided difference. That direction is stored in the return variable *divaxn*. The program then uses the value of *divaxn* to divide the region (in the *divaxn* direction) into four subregions of equal volume.

$4R_1$ (*three subdivisions based on two calls to* differ): Three subdivisions are performed successively, following two calls to *differ*. The direction in which the first subdivision is done is determined by the *divaxn* value returned from the first call to *differ*; the resulting subregions are both subdivided in the *divaxn* direction returned by the second call to *differ*.

$4R_2$ (*three subdivisions based on one* differ *call and the second largest fourth divided difference*): In this subdivision strategy, the first subdivision is performed based on the *divaxn* value returned from the call to *differ*. In the same call, the program determines the coordinate direction of the second largest fourth difference, and the second subdivision is performed in that direction (in each of the two subregions).

$8R_3$ is an extension of $4R_2$, with eight subregions resulting from one *differ* call and the largest, second and third largest fourth divided differences. $16R_4$ is an extension of $8R_3$

where fifteen subdivisions are performed based on one *differ* call, the largest, second and third largest fourth differences, and a fourth direction chosen in a round robin fashion.

3.2. Parallelization Strategies

3.2.1. GPU Mapping

After subregions have been generated by dividing a region as outlined above, the resulting subregions are passed simultaneously to the GPU kernel to be evaluated. The evaluation process yields the results (including information to estimate the error) over the individual subregions. The error estimates for the subregions are used later to insert the regions in their proper positions in the heap. The GPU kernel must ensure that the results for each individual region are kept separate from those of other subregions, even when multiple subregions are passed to the kernel simultaneously to be processed in parallel by many threads in multiple blocks.

Since multiple subregions are sent to the kernel simultaneously, keeping the results of all these subregions separate while they are being processed by multiple threads across multiple blocks presents a challenge. Let us consider the scenario in which four subregions are passed to the kernel simultaneously for a specific integral. In the GPU kernel, the total number of rule evaluation points will be the product of the number of points in the integration rule with the number of subregions to be processed simultaneously. It should be noted that the same set of integration points is used, scaled to each particular subregion. In the kernel, each thread (with ID *tid*) is assigned a point index computed as

$$pointId = \lfloor tid/numRegions \rfloor \tag{3.1}$$

where *numRegions* is the number of subregions passed simultaneously to the kernel. The index of the region to be processed by the thread is determined as

$$regionIndex = tid - pointId * numRegions \tag{3.2}$$

The resulting assignments of point and region indices to threads are shown in Table 3.1 for 400 evaluation points in four regions. The divisions in Table 3.1 are to be interpreted as integer divisions.

Table 3.1: Indices of Points and Regions Assigned to GPU Threads for Multiple (4) Regions Passed to the GPU Kernel

Tid	Point Index	Region Index
	$\lfloor tid/numRegions \rfloor$	$tid - pointId * numRegions$
0	$0/4 = 0$	$0 - 0 * 4 = 0$
1	$1/4 = 0$	$1 - 0 * 4 = 1$
2	$2/4 = 0$	$2 - 0 * 4 = 2$
3	$3/4 = 0$	$3 - 0 * 4 = 3$
4	$4/4 = 1$	$4 - 1 * 4 = 0$
5	$5/4 = 1$	$5 - 1 * 4 = 1$
6	1	2
7	1	3
8	$8/4 = 2$	$8 - 2 * 4 = 0$
...	...	...
392	$392/4 = 98$	$392 - 98 * 4 = 0$
393	98	1
394	98	2
395	98	3
396	99	0
397	99	1
398	99	2
399	99	3

Each thread accumulates the results and intermediate error estimate values of the specific subregion assigned to it. The kernel uses four arrays shared among the threads in each block. The first array contains contributions to the integration rule sum, which need to be summed to yield the integral approximation. The other three arrays have contributions to values needed for the comparison rules to determine the error estimation over the subregions.

3.2.2. Simultaneous Reductions

After the computations are completed by all threads, a reduction is performed on the GPU to accumulate results shared across a block into *numRegions* (= 4 in this case) threads in the block. The reduction and synchronization of the threads is done in the kernel along the lines of [40], but extended here to achieve *numRegions* reductions simultaneously. An explanation of this procedure follows, where we keep *numRegions* = 4.

A GPU block can be visualized as a single row partitioned into a number *threadsPerBlock* of cells which represent the threads within the block. These cells will have indices from 0 to *threadsPerBlock* −1. We assume that the first subregion with index 0 is processed by threads 0, 4, 8, ..., the second subregion is processed by threads 1, 5, 9, ..., the third subregion by threads 2, 6, 10, ..., and the fourth subregion by threads 3, 7, 11, ... As a result of the reduction process, all the results in threads 0, 4, 8, ... are accumulated into thread 0, those in the threads 1, 5, 9, ... are accumulated into the thread 1, and so on. These values are shared across the block to allow the reduction, which takes $\log_2$*(threadsPerBlock)* stages (where *threadsPerBlock* is a power of 2). There are *blocksPerGrid* blocks, each of which ends up with the partial sum (over its threads) for each of the *numRegion* (= 4 in this case) subregions on the GPUs. The array of *blocksPerGrid* $*$ *numRegions* partial sums is then returned by the kernel to the CPU, which performs the summation over the *blocksPerGrid* rows to determine the approximation for each of the regions (0, 1, ...*numRegions* −1).

The parallel computation thus ends with four sets of results for each of the subregions. One is the integral approximation and the remaining three are for the comparison integrals used on the CPU to complete the computation of the absolute error estimate.

CHAPTER 4

RESULTS AND ANALYSIS

The results presented in this chapter were obtained using ParAdapt, utilizing the subdivision and parallelization strategies described in Chapter 3. Two clusters were used due to limitations related to the integral dimension for some of the problems presented. Details of the clusters and GPUs used are outlined subsequently.

4.1. GPU Specifications

All programs evaluating integrals through dimension 14 were executed on the *thor* cluster at the Center for High Performance Computing and Big Data [41] in the Department of Computer Science at Western Michigan University (WMU). The cluster has twenty-two compute nodes, with 15 NVIDIA Tesla (Kepler) K20 GPU accelerator cards (on a subset of the nodes). The Tesla K20 has a Video Random Access Memory (VRAM) capacity of 4.8GB, which is insufficient for our implementation in dimensions higher than 14, mainly because of the memory required to store the points for the integration rule as well as the different categories of weights for the computation of the results and error estimates over the subregions. For higher dimensions, we used the cluster made available through Penguin Computing®On-Demand™(POD™) HPC-as-a-service [42]. The cluster on POD has Tesla K40 GPU accelerator cards, an enhanced version of the K20 cards. The K40 has a VRAM capacity of 12GB. Programs for evaluating integrals in up to 25 dimensions were executed on the POD cluster. A comparison of some of the features of the K20 and the K40 is shown in Table 4.1.

Table 4.1: Comparison of **NVIDIA's** Tesla K20 and K40 GPU Accelerator Cards [43]

	Tesla K20	Tesla K40
Number of cores	2496	2880
VRAM	4.8GB	12GB
Architecture	Kepler	Kepler
Core Clock	706MHz	745MHz
Memory Clock	5.2GHz GDDRS	6GHz GDDRS
Boost Clock(s)	NA	810MHz, 875MHz
Memory Bus Width	320-bit	384-bit

4.2. GPU Kernel Call Parameters

Two of the special parameters specified in a kernel call are the number of threads in each block used in the computations (*threadsPerBlock*) and the number of blocks per grid (*blocksPerGrid*). At the preliminary stages of obtaining the results presented in this chapter, various combinations of *threadsPerBlock* and *blocksPerGrid* were used, and their results were compared, including:

- 128 threadsPerBlock and 32 blocksPerGrid
- 128 threadsPerBlock and 64 blocksPerGrid
- 256 threadsPerBlock and 32 blocksPerGrid
- 256 threadsPerBlock and 64 blocksPerGrid
- 512 threadsPerBlock and 32 blocksPerGrid

For example, some results comparing these combinations for a 14-dimensional problem are presented in Table 4.2. For this problem, the option with 128 *threadsPerBlock* and

Table 4.2: ParAdapt adaptive integration results in dimension $d = 14$ for the integral of $1/d(\sum_{i=1}^{d} |4x_i - 2|)$ over the unit cube $\mathcal{C}_d, d = 14$ (with exact result equal to 1), showing integration results and corresponding speedups obtained for five combinations of *threadsPerBlock* (tpb) and *blocksPerGrid* (bpg) for maximum evaluations set to $50M$ (50 million).

	1 region at a time				2 regions at a time				4 regions at a time			
	Res.	E_a	Time (s)	Speedup	Res.	E_a	Time (s)	Speedup	Res.	E_a	Time (s)	Speedup
$128tpb, 32bpg$	0.987435	1.112E-02	5.1	56	0.987435	1.112E-02	4.7	62	0.969533	2.696E-02	4.4	66
$128tpb, 64bpg$	0.987435	1.112E-02	3.4	83	0.987435	1.112E-02	3.2	89	0.969533	2.696E-02	2.9	101
$256tpb, 32bpg$	0.987435	1.112E-02	4.2	67	0.987435	1.112E-02	3.6	83	0.969533	2.696E-02	3.0	96
$256tpb, 64bpg$	0.987435	1.112E-02	4.7	60	0.987435	1.112E-02	3.9	74	0.969533	2.696E-02	3.4	85
$512tpb, 32bpg$	0.987435	1.112E-02	4.7	59	0.987435	1.112E-02	4.0	73	0.969533	2.696E-02	3.4	86

64 *blocksPerGrid* gave the highest speedups. Better choices generally depend on the work needed for each integrand evaluation and on the number of points evaluated at once on the GPU. The latter depends on the number of points (N) in the rule (per region) and the number of regions evaluated at the same time (regionsAtOnce). Unless otherwise stated, the results are obtained using:

$$threadsPerBlock = 128$$

$$blocksPerGrid = min(64, (regionsAtOnce * N + threadsPerBlock - 1)/threadsPerBlock) \tag{4.1}$$

This divides the total number of points (regionsAtOnce * N) over *blocksPerGrid* blocks with *threadsPerBlock* threads in each block (see also [44]).

Results presented in Table 4.2 are in 3 categories: those obtained when only 1 subregion is sent to the GPU each time to be evaluated, results when 2 subregions are sent to the GPU to be evaluated simultaneously, and finally results when 4 subregions are sent simultaneously. The integral is $\frac{1}{d} \int_{\mathcal{C}_d} \sum_{i=1}^{d} |4x_i - 2| d\vec{x}$ with $d = 14$. The integrand function for this problem is very fine-grained, i.e., the work involved in each integrand evaluation is very small. Therefore

the implementation of the integrand is made to loop 100 times to simulate a more complex integrand (i.e., a real-world problem). Processing one or two regions at a time mimics the sequential strategy. A simple subdivision strategy was used in obtaining the results in Table 4.2, but results from more suitable strategies (with respect to maintaining the accuracy) will be given later in this chapter.

4.3. Results and Analysis

In this section, results are presented for four sample integrals on GPUs using ParAdapt. The third and fourth sample problems are real-world problems arising in Bayesian statistics and High Energy Physics respectively. The first two are used to illustrate the behavior for types of integrand functions that are known to be difficult for numerical integration in general: discontinuous derivatives and integrand singularities. Since, for problems 1 and 2, the integrands take very little time to evaluate at each point, we implement them with a real-world flavor by putting a loop around their evaluation that iterates a fixed number of times (set to 100 iterations).

Chapter 3 provides a detailed description of the subdivision strategies employed in ParAdapt. Results from sending two subregions at once to the GPU for evaluation ($2R$) are compared with four ($4R_0, 4R_1, 4R_2$), eight ($8R_3$) and sixteen ($16R_4$) regions being sent simultaneously. The $8R$, $16R$ and some of the $4R$ results were obtained using the compilation flag: *-arch=sm_35*. As stated in the CUDA Toolkit Documentation website [45], this command line option "specifies the name of the class of NVIDIA virtual GPU architecture for which the CUDA input files must be compiled". It provides some functional capabilities such as Kepler support and unified memory programming support - inherited from its predecessors (*sm_30* and *sm_32*) - and in addition provides dynamic parallelism support [45] (which we are not using). For our use, the *-arch=sm_35* command-line compilation flag

gives the same results as without the flag, while in some instances it gave a better speedup. Specifically, the $4R$ results for sample problems 1, 2 and 4 were obtained using that compilation flag. It was not used for sample problem 3. The results for sample problems 1 to 3 were also presented in Olagbemi and de Doncker [30].

4.3.1. Sample Problem 1 (*discontinuous partial derivatives*)

The first integral is of the form:

$$I_1 = \int_{\mathcal{C}_d} f_1(\vec{x})\ d\vec{x} = \frac{6}{5d} \int_{\mathcal{C}_d} \sum_{i=1}^{d} |3x_i - 1|\ d\vec{x} \tag{4.2}$$

where d denotes the number of dimensions. The exact result of the integral is 1.

To simulate a more difficult problem, the function in the integrand implementation is made to loop 100 times, with each loop fully computing an estimation of the integrand at the given point.

Table 4.3: ParAdapt adaptive integration results in 10 dimensions for 2R and 4R with the three subdivision strategies $4R_0$, $4R_1$, and $4R_2$. The results, error estimates and times (in seconds) are presented for maximum evaluations set at $100M$ (100 million), $1B$ and $5B$, with corresponding speedups.

	100M			1B			5B			
	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Speedup
$2R$	1.000807	1.192E-03	18.1	1.000205	3.026E-04	183.1	1.000053	7.774E-05	921.7	29
$4R_0$	1.003063	4.521E-03	10.8	1.001368	2.019E-03	107.6	1.000564	8.327E-04	542.8	49
$4R_1$	1.000668	9.859E-04	18.9	1.000164	2.421E-04	189.3	1.000041	6.045E-05	983.1	28
$4R_2$	1.000742	1.095E-03	10.8	1.000190	2.810E-04	130.2	1.000056	8.264E-05	544.9	50

According to Table 4.3, the more accurate results are obtained by $4R_1$. This follows from the fact that there are two calls to *differ* for the three subdivisions performed. The first subdivision is performed in the *divaxn* dimension returned from the first *differ* call, and the

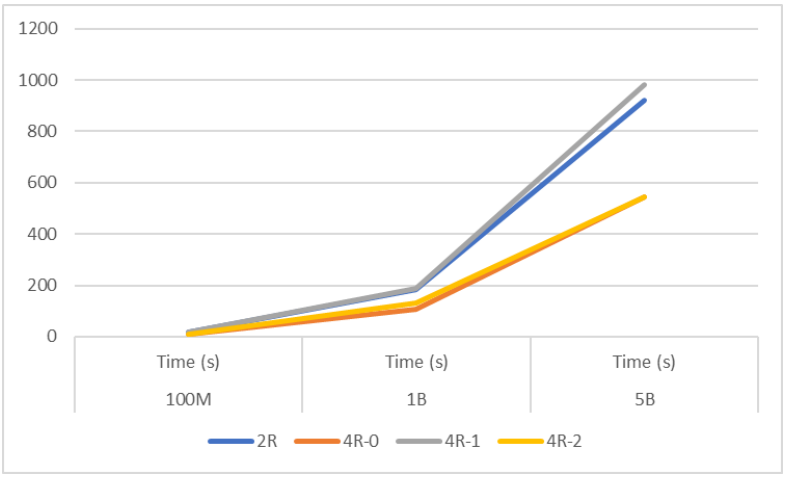

Figure 4.1: Execution times for $2R$, $4R_0$, $4R_1$, and $4R_2$ from Table 4.3 (integral in 10 dimensions)

last two subdivisions in the *divaxn* direction determined in the second *differ* call (and thus the subdivisions are based on the dimensions with the highest fourth differences). However, as can be seen from the speedup values, this accuracy comes at a cost, due mainly to the second *differ* call. $4R_1$ has the lowest speedup of all four sets of results. $4R_0$ employs the simplest approach and as such does the least work of all three $4R$ strategies. Even though this strategy has the second highest speedup, its accuracy is the lowest of all four sets of results. $4R_2$ attempts to strike a compromise between $4R_0$ and $4R_1$ by using *divaxn* for the first subdivision (as is done by the other two $4R$ strategies), after which it uses *secondMax* (the dimension with the second largest fourth difference, computed in the same call to *differ* that determines the value of *divaxn*) for the last two subdivisions. Its result, though not as accurate as $4R_1$, is reasonably close to it, and is also more accurate than $4R_0$. In this case, it has the best speedup of all four sets of results.

It is also worth noting that for the four sets of results, the time in seconds increases approximately with the factor with which the number of evaluations increases; for example, for $2R$, the time for 100M evaluations is 19.2s and it can be seen that when the number

of evaluations increases by a factor of 10 to 1B, the time also increases by approximately that factor ($19.2 * 10 \approx 193$). This indicates that the amount of work done in the GPU computations dominates the program execution. The speedups achieved also indicate good scalability of the parallel implementation of the program (with respect to the total work done). The graph in Figure 4.1 is a pictorial representation of the execution times for the four strategies given in Table 4.3.

The results for the integral of (4.2) in 14 dimensions are presented in Table 4.4 along with the corresponding graph showing execution times for the various strategies.

Table 4.4: ParAdapt adaptive integration results in 14 dimensions for $2R$, $4R_0$, $4R_1$, and $4R_2$ were obtained with 128 *threadsPerBlock* and *blocksPerGrid* according to (4.1). The results, error estimates and times (in seconds) are given for maximum evaluations set at $100M$ (100 million), $1B$ and $5B$ with corresponding speedups.

	100M			1B			5B			
	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Speedup
$2R$	1.004506	7.044E-03	7.3	1.002664	4.164E-03	73.2	1.001956	3.058E-03	365.7	91
$4R_0$	1.006949	1.086E-02	5.7	1.005745	8.979E-03	56.6	1.004788	7.483E-03	283.0	117
$4R_1$	1.004127	6.450E-03	6.8	1.002557	3.997E-03	67.7	1.001792	2.801E-03	347.5	99
$4R_2$	1.004157	6.498E-03	5.5	1.002632	4.114E-03	58.0	1.001851	2.894E-03	290.4	123

The accuracy of the results for all four strategies is less for the 14-dimensional integral than the lower-dimensional one in 10 dimensions. Note that for the same maximum number of function evaluations, the execution times for the 14-dimensional (14D) integral are significantly less than those for the 10D integral. This is due to the fact that the number of evaluations per region is higher as the dimension increases, and thus the total number of regions processed is lower in dimension 14. This also indicates that a larger maximum number of evaluations is justified in higher dimensions. Indeed, the function evaluations are performed in a massively parallel way in our application. The speedup for $4R_2$, the most efficient strategy of the four when considering both the accuracy and execution time, is as

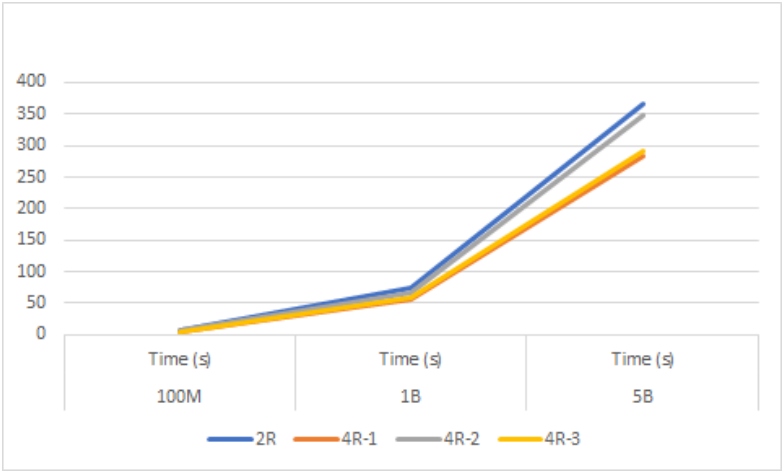

Figure 4.2: Execution times for $2R$, $4R_0$, $4R_1$, and $4R_2$ (integral in 14 dimensions)

high as 123. The lower accuracy and execution times seen (with a fixed maximum number of evaluations) for the $14D$ program when compared with the $10D$ program are due mainly to the fact that the number of evaluation points in the integration rule increases exponentially as the dimension increases (see (2.1). Since one of the terminating criteria is that the program terminates when the maximum number of evaluations has been reached, programs for integrals in higher dimensions reach or exceed this maximum limit with fewer subdivisions than in lower dimensions. It should be pointed out that the actual error generally depends on the dimension (dimensional effect [19]).

Results for the integral of (4.2) in 15 to 25 dimensions are listed in Table 4.5 using the $4R_2$ subdivision strategy, as that strategy gives the best combination of accuracy and efficiency in this case.

From the results in Table 4.5, for reasons given earlier, the accuracy degrades for the same number of function evaluations as the number of dimensions increases (including the dimensional effect, and fewer subdivisions being possible because of the limit on the allowed number of function evaluations). The time increases because of the exponential increase in

Table 4.5: ParAdapt adaptive integration $4R_2$ results for 15, 17, 19, 20, 21, 22, 24 and 25 dimensions. The results, error estimates and times (in seconds) are shown for maximum evaluations of $100M$ (100 million), $1B$ and $5B$, with corresponding speedups.

	100M			1B			5B			
	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Speedup
$15D$	1.005361	8.373E-03	5.2	1.003500	5.468E-03	52.1	1.002528	3.949E-03	260.4	107
$17D$	1.007125	1.270E-02	5.4	1.005406	9.633E-03	53.3	1.004274	7.617E-03	267.1	117
$19D$	1.008506	1.822E-02	6.4	1.006932	1.485E-02	63.5	1.005910	1.266E-02	317.6	106
$20D$	1.008991	2.134E-02	6.3	1.007632	1.812E-02	62.0	1.006560	1.557E-02	308.9	116
$21D$	1.009625	2.555E-02	7.1	1.008212	2.180E-02	69.4	1.007286	1.934E-02	343.5	104
$22D$	1.010015	2.987E-02	7.9	1.008786	2.620E-02	75.3	1.007815	2.331E-02	372.8	97
$24D$	1.010744	4.037E-02	12.5	1.009742	3.660E-02	82.4	1.008906	3.346E-02	405.2	89
$25D$	1.010955	4.592E-02	15.5	-	-	-	-	-	-	73

the number of points (see (2.1)) used by the rule (see Chapter 2). Values for $1B$ and $5B$ could not be obtained for $25D$ as the program timed out, which is likely due to the high number of evaluation points in the integration rule for $25D$ ($> 33.5M$).

4.3.2. Sample Problem 2 (*corner singularity*)

ParAdapt is applied to approximate the integral in (4.3), which has an integrand singularity at the origin. The integral is of the form:

$$I_2 = \int_{C_d} \frac{1}{(\sum_{i=0}^{d-1} x_i)^2} \tag{4.3}$$

where d is the number of dimensions in the integral.

$2R$ and $4R_2$ results are presented for the integral in 10 and 14 dimensions. Table 4.6 lists results for the 10-dimensional integral. The function in the integrand implementation is made to loop a hundred times to simulate a real world problem. Table 4.7 gives similar results but for the 14-dimensional integral, also with a loop of a hundred iterations in the

function definition. Table 4.8 presents results, also for the 14-dimensional integral, in which the function does not loop. The *-arch* flag was used in the compilation for these integrals. Speedup values for lower numbers of evaluations are included in the tables, but not those for higher evaluations as the sequential program would run for too long. Furthermore, the two speedup results recorded for each category of results give a good indication of the speedup values that can be expected for the higher numbers of evaluations.

Table 4.6: ParAdapt Sample 2 $2R$ and $4R_2$ results are presented in $10D$ for $100M$, $500M$, $5B$, $10B$ and $12B$ evaluations with a loop of hundred iterations in the function. ParAdapt results were obtained with 128 *threadsPerBlock* and *blocksPerGrid* according to (4.1) for all parallel runs.

	$2R$				$4R_2$			
Evals.	Res.	E_a	Time (s)	Speedup	Res.	E_a	Time (s)	Speedup
$100M$	0.044832353651	1.975E-07	13.4	22.0	0.044832351763	1.626E-07	9.3	43.1
$500M$	0.044832347091	6.828E-08	66.3	22.2	0.044832346599	5.499E-08	48.4	41.8
$5B$	0.044832345212	1.691E-08	716.2	-	0.044832345138	1.438E-08	627.5	-
$10B$	0.044832345057	1.188E-08	1,366.7	-	0.044832345013	1.016E-08	1,394.0	-
$12B$	0.044832345031	1.087E-08	1,641.8	-	0.044832344992	9.281E-09	1,616.9	-

From the results for $10D$ in Table 4.6, the $4R_2$ strategy yields the more accurate results based on the lower error estimate values, and further achieves higher speedups, almost double those of $2R$. Both sets of results for this sample problem, as shown in the corresponding tables, are consistent up to the respective error estimates with results obtained by ParInt using large numbers of evaluations.

The $14D$ results in Table 4.7 are consistent with those of Table 4.6 in that the $4R_2$ strategy outperforms $2R$ in terms of both accuracy and execution time. Even though speedup values for $4R_2$ are not up to double those of $2R$ as was the case for the $10D$ integral, they are still much higher.

Table 4.8 gives results for the integration without a loop in the implementation of the function. In this scenario, the $2R$ strategy outperforms the $4R_2$ in terms of execution time

Table 4.7: ParAdapt Sample 2 $2R$ and $4R_2$ results are presented in $14D$ for $100M$, $500M$, $5B$, $10B$, $12B$ and $25B$ evaluations with a loop of hundred iterations in the function. ParAdapt results were obtained with 128 *threadsPerBlock* and *blocksPerGrid* according to (4.1) for all parallel runs.

Evals.	Res.	$2R$ E_a	Time (s)	Speedup	Res.	$4R_2$ E_a	Time (s)	Speedup
$100M$	0.022067923724	4.283E-07	6.0	69.7	0.022067916221	1.718E-07	5.4	91.9
$500M$	0.022067892909	1.859E-07	29.9	69.8	0.022067890589	1.672E-07	27.2	92.2
$5B$	0.022067881617	9.052E-08	299.7	-	0.022067880791	8.202E-08	299.4	-
$10B$	0.022067879822	7.204E-08	597.9	-	0.022067879212	6.531E-08	792.7	-
$12B$	0.022067879416	6.802E-08	962.5	-	0.022067878816	6.111E-08	951.2	-
$25B$	0.022067878074	5.247E-08	2,640.4	-	0.022067877630	4.676E-08	1,440.0	-

Table 4.8: ParAdapt Sample 2 $2R$ and $4R_2$ results (without a loop in the function) are presented in $14D$ for $100M$, $500M$, $5B$, $10B$, $12B$ and $25B$ evaluations. ParAdapt results were obtained using 128 *threadsPerBlock* and *blocksPerGrid* according to (4.1) for all parallel runs.

Evals.	Res.	$2R$ E_a	Time (s)	Speedup	Res.	$4R_2$ E_a	Time (s)	Speedup
$100M$	0.022067923724	4.283E-07	0.6	30.3	0.022067916221	3.718E-07	1.2	15.3
$500M$	0.022067892909	1.859E-07	3.1	30.0	0.022067890589	1.672E-07	6.2	15.3
$5B$	0.022067881617	9.052E-08	31.2	-	0.022067880791	8.202E-08	61.8	-
$10B$	0.022067879822	7.240E-08	62.4	-	0.022067879212	6.531E-08	123.6	-
$12B$	0.022067879416	6.802E-08	75.1	-	0.022067878816	6.111E-08	93.3	-
$25B$	0.022067878074	5.247E-08	156.8	-	0.022067877630	4.676E-08	308.1	-

with speedup values about double those of $4R_2$. However, in terms of accuracy, the error estimates for $4R_2$ are consistently lower than those for $2R$. Without the loop, the work to be done in each integrand evaluation is very fine-grained. This decreases the GPU performance, which in turn affects the performance of both strategies, with the $4R_2$ strategy seeing a larger decline in speedup values. A significant increase in the work, achieved by including a loop, results in the $4R_2$ strategy producing a better speedup than $2R$, as can be seen from the results in Table 4.7. The results for this sample problem, as shown in the corresponding tables, are consistent up to the respective error estimates with results obtained by ParInt

using large numbers of evaluations in long double precision.

It is also noted in Table 4.8 that one of the timing values obtained for $4R_2$ is not consistent with timing results obtained for other numbers of evaluations. Specifically, the time for 12B evaluations is less than the time for 10B evaluations. This may be due to a different course of the adaptive strategy, which may lead to uncovering important regions unexpectedly.

4.3.3. Sample Problem 3 - Linear Model (*peak singularity*)

This integral derives from the field of Bayesian Statistics and was presented with results by Evans and Swartz in [46]. In [5], de Doncker and Almulihi compared their results for this integral using the ParInt software with those from Evans and Swartz. This section compares results obtained by ParAdapt with those of Evans et al. and de Doncker et al.

The integral takes the form:

$$I(w) = \int_{R^k} w(\theta) f(\theta) d\theta, \tag{4.4}$$

where $w : R^k \to R$ and $f : R^k \to R^+$; f represents the *common part*, which in Bayesian statistics refers to the product of the likelihood and the prior. Choosing the approximation technique to be used very much depends on the common part, hence the form of the integral presented.

The simulated data from [46] flows from the observed response value $\mathbf{y}$ specified as follows: $\mathbf{y} = X\beta + \sigma\mathbf{z}$, where $\mathbf{y} \in R^{45}$ and $X \in R^{45\times 9}$. Here $x_{ij} = 1$ for i and j satisfying $5(j-1)+1 \leq i \leq 5j$, and 0 otherwise.

The integration variables for the function $f(\theta)$, which is specified in 10 dimensions, are $\theta_i = \beta$ for values of i satisfying $1 \leq i \leq 9$, and $\theta_{10} = \log \sigma$, and $f(\theta)$ is given by

$$f(\theta) = e^{-9n\theta_{10}} \prod_{i=1}^{9} \prod_{i=1}^{n} g_\lambda(\frac{y_{ij} - \theta_i}{e^{\theta_{10}}}) \tag{4.5}$$

with

$$g_\lambda(z) = \frac{\Gamma(\frac{\lambda+2}{2})}{\Gamma(\frac{1}{2})\Gamma(\frac{\lambda}{2})}(1 + \frac{z^2}{\lambda - 2})^{-\frac{\lambda+1}{2}} \frac{1}{\sqrt{\lambda - 2}} \tag{4.6}$$

where $\lambda = 3$ and $n = 5$.

The integrand function has a peak located at the mode $\hat{\theta}$. The integration region with the mode at the center is truncated using a half-width of 5 (i.e., the integration domain becomes a cube of width 10 in each coordinate direction). From experimental results, the choice of the half-width can significantly affect the results obtained. as shown in Table 4.9. Half-width values 5.0, 5.3, 5.5 and 6.0 were used with the $2R$ and $4R_2$ subdivision strategies. From that comparison, a half-width of 5.0 gives good results (and is also used by [5]). Subsequent results presented here for this Sample Problem are therefore based on a half-width of 5.0.

The absolute error tolerance is set to 0 due to the small values of the results for the integral $I(1)$ (of the order of 10^{-22}), while the relative error tolerance is set to 10^{-12} to allow the program to continue running till the maximum number of evaluations has been exceeded.

The integrand function fw in (4.4) is comprised of f and w where f is specified as $f(\theta)$ in (4.5) and $w = w(\theta) = \theta_1, \theta_2, \theta_4, \theta_{10}, \theta_1^2, \theta_2^2, \theta_4^2, \theta_{10}^2$. Results obtained for the integral of fw using ParAdapt with the $2R$ subdivision strategy are shown in Table 4.10. The results and error estimates in the table for θ_i and θ_i^2 are scaled using the result for $I(1)$ (i.e., with $w = 1$), and compared with results from ParInt in [5] (Table 1 in that article) and results for $R(\theta_i)$ and $R(\theta_i^2)$ from Evans and Swartz in [46] (Tables 2, 3, 4, 5, 6).

Table 4.9: ParAdapt $4R_2$ results for $I(1)$ using half-widths of 5.0, 5.3, 5.5, and 6.0. The results, error estimates and times (in seconds) are listed for maximum evaluations of $500M$ (500 million), $1B$ and $5B$, with results for $8B$ and $10B$ shown for higher half-width values.

	$2R$			$4R_2$		
Evals.	Res.	E_a	Time (s)	Res.	E_a	Time (s)
			$Halfwidth = 5.0$			
$500M$	1.29762E-22	6.25E-25	57.3	1.27667E-22	6.67E-25	30.8
$1B$	1.29791E-22	3.97E-25	114.5	1.28387E-22	4.05E-25	61.6
$5B$	1.29990E-22	1.50E-25	572.3	1.29536E-22	1.48E-25	306.2
			$Halfwidth = 5.3$			
$500M$	1.30312E-22	8.00E-25	57.3	1.20309E-22	9.82E-25	30.8
$1B$	1.30027E-22	4.53E-25	114.6	1.28880E-22	5.78E-25	61.5
$5B$	1.30322E-22	1.81E-25	608.3	1.30554E-22	1.98E-25	307.3
			$Halfwidth = 5.5$			
$500M$	1.30967E-22	1.16E-24	57.3	1.18141E-22	1.07E-24	30.8
$1B$	1.30315E-22	5.78E-25	114.1	1.18915E-22	5.35E-25	61.4
$5B$	1.30257E-22	1.91E-25	759.4	1.30495E-22	1.83E-25	477.0
$8B$	1.30386E-22	1.45E-25	1,215.5	1.30524E-22	1.36E-25	776.9
$10B$	1.30447E-22	1.27E-25	1,465.6	1.30537E-22	1.18E-25	881.0
			$Halfwidth = 6.0$			
$500M$	1.30900E-22	1.17E-24	57.6	1.30528E-22	9.95E-25	31.1
$1B$	1.30633E-22	5.77E-25	114.3	1.30043E-22	5.01E-25	62.0
$5B$	1.30410E-22	1.79E-25	720.2	1.30485E-22	1.65E-25	438.6
58	1.30442E-22	1.32E-25	1,137.7	1.30482E-22	1.21E-25	601.0
$10B$	1.30459E-22	1.14E-25	1,304.8	1.30479E-22	1.04E-25	841.1

The ParAdapt results generally compare favorably with those from both ParInt and Evans et al., with a significantly lower number of function evaluations than ParInt. For these tests ParAdapt achieved speedups of around 27.5 ($2R$) and 51 ($4R_2$). The ParInt results were obtained using 64 or 128 MPI processes [47] on four cluster nodes, allowing $2B$

Table 4.10: ParAdapt $2R$ results are compared with ParInt results and results from Evans and Swartz for *Example 1* over 10D truncated domain, centered at mode with half-width = 5. ParAdapt results are displayed for $500M$ and $1B$ evaluations while ParInt results were obtained using 64 or 128 MPI processes, allowing $2B$ evaluations using 64 procs. and $25B$ evaluations using 128 procs.

	$2R$, 500M evals.			$2R$, 1B evals.			2B, 64 procs.			25B, 128 procs.			E&S $\hat{R}$	E&S R
w	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Subr. Ad.	Exact
1	1.2976 $\times 10^{-22}$	6.26E-25	57.3	1.2979 $\times 10^{-22}$	3.97E-25	114.5	1.317 $\times 10^{-22}$	3.6E-24	95.7	1.3065 $\times 10^{-22}$	2.7E-25	592.3		1.31 $\times 10^{-22}$
θ_1	2.043	9.91E-03	57.6	2.043	6.25E-03	115.1	2.043	5.5E-02	95.7	2.0428	4.2E-03	591.9	2.040	2.043
θ_2	0.0947	1.19E-03	57.6	0.0949	8.05E-04	114.8	0.0957	3.8E-03	95.2	0.09473	3.4E-04	591.5	0.094	0.095
θ_4	0.0182	1.04E-03	57.2	0.01821	6.76E-04	114.7	0.0170	3.9E-03	95.4	0.01801	3.3E-04	592.0	0.017	0.018
θ_{10}	-0.0749	1.26E-03	57.3	-0.07498	7.51E-04	114.7	-0.0738	8.4E-03	95.8	-0.07290	6.5E-04	593.7	-0.102	-0.073
θ_1^2	4.264	2.07E-02	57.2	4.265	1.30E-02	114.9	4.261	1.12E-01	95.7	4.2636	8.8E-03	593.4	4.237	4.263
θ_2^2	0.0806	4.91E-04	56.9	0.0807	3.14E-04	115.1	0.0788	1.3E-03	95.8	0.08074	1.6E-04	593.3	0.068	0.081
θ_4^2	0.06838	4.32E-04	57.3	0.06867	2.88E-04	114.3	0.0677	1.0E-03	95.7	0.06906	1.1E-04	592.5	0.055	0.069
θ_{10}^2	0.03201	4.29E-04	57.2	0.0321	2.57E-04	114.1	0.0320	1.6E-03	96.1	0.03269	1.2E-04	593.5	0.032	0.033

evaluations using 64 procs. and $25B$ evaluations using 128 procs.

4.3.4. Sample Problem 4 - Feynman Loop Integral (*boundary singularity*)

This application of multivariate integration using ParAdapt is from the field of Computational Physics, specifically Feynman loop integrals, which provide higher-order corrections that enable accurate predictions of the cross-section for particles interactions. We will compute a 10-dimensional Feynman loop integral from [48], written in the form:

$$I_{L,N} = (-1)^N \Gamma(N - \frac{vL}{2}) \int_{C_N} \prod_{j=1}^{N} dx_j \delta(1 - \sum x_j) \frac{C^{N-v(L+1)/2}}{(D - i\varrho C)^{N-vL/2}} \tag{4.7}$$

where L is the number of loops in the Feynman diagram, N is the number of internal lines, and C and D are homogeneous polynomials in $x_1, ..., x_N$. The factor ϱ ensures that the denominator does not evaluate to zero, while the space-time dimension $v = 4$. Eliminating the δ function results in the integral of (4.8), which is expressed over the d-dimensional unit

simplex ($d = N - 1$), $S_d = \{x \in C_d \mid 0 \leq \sum_{j=1}^{d} x_j \leq 1\}$,

$$I_{L,N} = (-1)^N \Gamma(N - \frac{vL}{2}) \int_{S_{N-1}} \frac{C^{N-v(L+1)/2}}{(D - i\varrho C)^{N-vL/2}} dx \tag{4.8}$$

In the ParAdapt code, specifically in the function with the integral definition, two transformations are applied to the integral: the first transformation converts the integration domain from the unit simplex to the unit cube (since ParAdapt is not designed to compute integrals over a simplex), while the second transformation, Sidi's sin^m transformation [49], [50], is used to handle singularities at the boundaries of the integration domain to make the numerical integration easier. Preliminary results obtained with and without Sidi's transformation showed that the integration without the transformation, even at significantly higher values of function evaluations, resulted in significantly lower accuracy.

A result for the integral is given as 35.1 in [51]. Results obtained from ParInt for the same integral are given in [48] (in Table 3 of that paper). ParAdapt integration results are presented in Table 4.11.

Table 4.11: ParAdapt adaptive integration $2R$, $8R_3$ and $16R_4$ results are shown for $200M$, $500M$, $1B$, $5B$, $10B$ and $12B$ evaluations.

	$2R$			$8R_3$			$16R_4$		
Evals.	Res.	E_a	Time (s)	Res.	E_a	Time (s)	Res.	E_a	Time (s)
$200M$	35.1053	1.574E+00	17.0	35.1331	1.940E+00	5.6	35.5057	3.084E+00	3.7
$500M$	35.1690	9.880E-01	40.4	35.2478	1.199E+00	13.6	35.2835	1.973E+00	8.7
$1B$	35.2320	6.779E-01	79.8	35.2886	8.161E-01	27.1	35.2432	1.382E+00	17.0
$5B$	35.3118	2.642E-01	424.4	35.3094	3.158E-01	130.8	35.2618	5.827E-01	85.6
$10B$	35.3097	1.771E-01	753.9	35.3071	2.100E-01	260.9	35.2786	3.915E-01	164.1
$12B$	35.3078	1.598E-01	916.9	35.3046	1.888E-01	304.3	35.2817	3.520E-01	195.8

From the results in Table 4.11, the $2R$ strategy yields the more accurate results based on the absolute error estimates. A speedup value of 27.7 was obtained for $2R$ with maximum

evaluations set at $500M$. The results produced by the $8R_3$ subdivision strategy were reasonably close to those from the $2R$ strategy but with higher absolute error estimates than $2R$ for corresponding maximum evaluations. However, the execution times for $8R_3$ are significantly less than those for $2R$ for similar error estimates (irrespective of the number of evaluations), and a significantly higher speedup of 173.8 was achieved for $8R_3$ with $500M$ evaluations. The $16R_4$ strategy achieves a speedup of 130.7 at $500M$ evaluations and has even lower execution times than the $8R_3$ strategy. The absolute error estimates are higher than those for both $2R$ and $8R_3$, but as can also be seen for the other two strategies, there is convergence with the absolute error estimates decreasing as the maximum function evaluations increase.

CHAPTER 5

DISCUSSION AND CONCLUSION

Results from ParAdapt, our solution for multivariate numerical integration on GPUs (using the adaptive strategy), have been presented in this dissertation. The adaptive algorithm is particularly effective because as the subregions become smaller in size (following subdivisions), the points in the integration rule achieve more coverage of the integration region, resulting in improved accuracy. The adaptive partitioning creates hot spots and succeeds in concentrating the integration efforts in the vicinity of singularities and other irregular integrand behavior.

One of the main over-arching goals of this research, as addressed in the problem statement in Chapter 1, was to increase the upper threshold on the integral dimension for the adaptive algorithm, by about 10 to 12. Historically, the adaptive strategy has been used in the approximation of integrals of low to moderate dimension (say, 10 to 12). Our solution ParAdapt stores the points and weights on the GPU and, in higher dimensions, the memory usage for storage of required array values is significant. However, results included for the first sample problem in Chapter 4 show that ParAdapt was able to successfully provide approximations to an integral with dimension as high as 24. This goal was therefore achieved, and ParAdapt actually doubles the value of the upper threshold on the dimension.

The GPU mapping and the simultaneous reductions of the results for multiple regions on the GPU form important building blocks in the ParAdapt methods. These may be applicable to different parallel applications.

ParAdapt execution times for each of the sample problems presented, for varying numbers of evaluations, point to the scalability of the algorithms as the total work increases:

the increase in the execution time from a specific of number of evaluations to another was found to be proportional to the corresponding increase of the number of evaluations. This indicates that the work on the GPU dominated the computation throughout a large number of subdivisions.

The results of the sample problems in Chapter 4 illustrate that ParAdapt handles various types of integration difficulties as a general purpose solution. These include singular behavior in the interior of the domain (Sample 1), corner singularity (Sample 2), peak (Sample 3) and boundary singularities (Sample 4).

Sample 4 is a problem that is well-suited for integration by lattice rules after a transformation that alleviates the boundary singularities. However, the Bayesian integral of Sample 3 cannot be treated by a lattice rule, in view of the large peak of the integrand in the domain.

ParInt handles Sample 3, but using significant resources (MPI on 64 or 128 processes). The ParAdapt results compare favorably with those of ParInt with respect to accuracy and time.

It is our goal to implement further refinements in ParAdapt to improve memory usage, accuracy of results, and program efficiency (in terms of execution time). Details of future work planned are included in the next section.

5.1. Future Work

As was mentioned earlier, memory usage becomes significant as the integral dimension increases. Currently, all points and weights are stored on the GPU to be utilized in the integrand evaluations and in the error estimation process. The integration rules used in ParAdapt are fully symmetrical [37] and we are considering the possibility of generating the symmetrical sets on the GPU rather than storing them on the device. The effects of this on the execution time will be assessed after implementation of this enhancement. Further

with respect to the integration rule, so far we used ADAPT and its degree 7 rule because of simplicity. It would be possible to incorporate the degree 9 rule of DCUHRE even though it uses more points.

A feature of CUDA that may be considered for future developments is "dynamic parallelism" [52], [53], referring to a scenario in which a parent CUDA kernel can call a child CUDA kernel and, without any involvement from the CPU, consume the output from the child kernel.

Another feature of CUDA that we are considering exploring is "unified memory" [54], [55]. From the documentation provided by NVIDIA, it appears that the gains in execution time in ParAdapt may not be significant in our current environment. However, it may be worth exploring for porting to new GPUs communicating over a fast bus.

BIBLIOGRAPHY

[1] E. de Doncker, F. Yuasa, K. Kato, T. Ishikawa, J. Kapenga, and O. Olagbemi, "Regularization with numerical extrapolation for finite and uv-divergent multi-loop integrals", *Computer Physics Communications*, vol. 224, pp. 164–185, 2018. [Online]. Available: \\https://doi.org/10.1016/j.cpc.2017.11.001.

[2] O. E. Olagbemi, E. de Doncker, and F. Yuasa, "Workshop on large scale computational physics - LSCP 2016", *Procedia Computer Science*, vol. 80, pp. 1416–1417, 2016.

[3] K. Kato, E. de Doncker, T. Ishikawa, J. Kapenga, O. Olagbemi, and F. Yuasa, "High performance and increased precision techniques for Feynman loop integrals", *J. Physics: Conf. Series (JPCS), IOP Series*, vol. 762, 2016, 012070, iopscience.iop.org/article/10.1088/1742-6596/762/1/012070.

[4] E. de Doncker, F. Yuasa, J. Kapenga, and O. Olagbemi, "Scalable software for multivariate integration on hybrid platforms", vol. 640, 2015, doi:10.1088/1742-6596/640/1/012062.

[5] E de Doncker and A. Almulihi, "Adaptive task partitioning for Bayesian applications", in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2016, pp. 572 –577. [Online]. Available: https://ieeexplore.ieee.org/document/7881407.

[6] H. S. Li, J. F. Dempsey, and H. E. Romeijin, "A Fourier analysis on the optimal grid size for discrete proton beam dose calculation", *Med. Phys.*, vol. 33, no. 9, pp. 3508–3518, 2006.

[7] S. Li, E. de Doncker, K. Kaugars, and H. Li, "A fast integration method and its application in a medical physics problem", *Springer Lecture Notes in Computer Science (LNCS)*, vol. 3984, pp. 789–797, 2006.

[8] S. Li, K. Kaugars, and E. de Doncker, "Grid based numerical integration and visualization", in *International Conference of Computational Intelligence and Multimedia Applications (ICCIMA)*, CDROM Proceedings, 2005.

[9] ——, "Distributed adaptive multivariate function visualization", *International Journal of Computational Intelligence and Applications (IJCIA)*, vol. 6, no. 2, pp. 273–288, 2006.

[10] S. Li, "Online support for multivariate integration", PhD thesis, Western Michigan University, 2005.

[11] J. Li and E. de Doncker, "Dynamic visualization of computations on the internet", in *Lecture Notes in Computer Science*, vol. 1593, Springer-Verlag, 1999, pp. 360–369.

[12] K. Mullen, D. M. Ennis, E. de Doncker, and J. A. Kapenga, "Multivariate models for the triangular and duo-trio methods", *Biometrics*, vol. 44, pp. 1169–1175, 1988.

[13] A. C. Genz, *Adapt*. [Online]. Available: http://http://www.math.wsu.edu/faculty/genz/software/fort77/adapt.f, (accessed: 08.01.2019).

[14] R Zanny, E de Doncker, K Kaugars, and L Cucos, PARINT 1.2 USER'S MANUAL, 2002. [Online]. Available: https://cs.wmich.edu/parint/, (accessed: 09.07.2019).

[15] P. van Dooren and L. de Ridder, "An adaptive algorithm for numerical integration over an n-dimensional rectangular region: Algorithm 006", *Journal of Computational and Applied Mathematics*, vol. 2, no. 3, pp. 207–217, 1976. [Online]. Available: https://doi.org/10.1016/0771-050X(76)90005-X.

[16] J. Berntsen, T. O. Espelid, and A. C. Genz, "An adaptive algorithm for the approximate calculation of multiple integrals", *ACM Transactions on Mathematical Software*, vol. 17, no. 4, pp. 437–451, 1991. [Online]. Available: https://dl.acm.org/citation.cfm?doid=210232.210233.

[17] J. Berntsen, T. O. Espelid, and A. Genz, "Algorithm 698: DCUHRE-an adaptive multidimensional integration routine for a vector of integrals", *ACM Trans. Math. Softw.*, vol. 17, pp. 452–456, 1991, http: //www.sci.wsu.edu/math/faculty/genz/homepage.

[18] NVIDIA, *Nvidia high performance computing - CUDA zone*. [Online]. Available: https://developer.nvidia.com/cuda-zone, (accessed: 08.31.2019).

[19] P. J. Davis and P. Rabinowitz, *Methods of Numerical Integration*. Orlando, Florida: Academic Press, Inc., 1984.

[20] R. Schürer, "A comparison between (quasi-)monte carlo and cubature rule based methods for solving high-dimensional integration problems", *Mathematics and Computer in Simulation*, vol. 62, pp. 509–517, 3-6 2003. [Online]. Available: https://doi.org/10.1016/S0378-4754(02)00250-1.

[21] A. H. Stroud, *Approximate Calculation of Multiple Integrals*. Englewood Cliffs, New Jersey: Prentice Hall, 1971.

[22] A. C. Genz and A. A. Malik, "Remarks on algorithm 006: An adaptive algorithm for numerical integration over an n-dimensional rectangular region", *Journal of Computational and Applied Mathematics*, vol. 6, no. 4, pp. 295–302, 1980. [Online]. Available: https://doi.org/10.1016/0771-050X(80)90039-X.

[23] A. Genz, "Testing multidimensional integration routines", in *Tools, Methods and Languages for Scientific and Engineering Computation*, 1984, pp. 81–94.

[24] H. D. Shapiro, "Increasing robustness in global adaptive quadrature through interval selection heuristics", *ACM Transactions on Mathematical Software*, vol. 10, no. 2, pp. 117–139, 1984.

[25] H. O'Hara and F. J. Smith, "The evaluation of definite integrals by interval subdivision", *The Computer Journal*, vol. 12, no. 2, pp. 179–182, 1969. [Online]. Available: https://doi.org/10.1093/comjnl/12.2.179.

[26] R. Cranley and T. N. L. Patterson, "On the automatic numerical evaluation of definite integrals", *The Computer Journal*, vol. 14, no. 2, pp. 189–198, 1971. [Online]. Available: https://doi.org/10.1093/comjnl/14.2.189.

[27] G. E. Forsythe, M. A. Malcolm, and C. B. Moler, *Computer Methods for Mathematical Computations*. Englewood Cliffs, New Jersey: Prentice-Hall, 1977.

[28] L. F. Shampine and J. R. C. Allen, *Numerical Computing: An Introduction*. Philadelphia: Saunders, 1973.

[29] E. de Doncker, F. Yuasa, K. Kato, T. Ishikawa, J. Kapenga, and O. Olagbemi, "Regularization with numerical extrapolation for finite and UV-divergent multi-loop integrals", *Computer Physics Communications*, vol. 224, pp. 164–185, 2018, https://doi.org/10.1016/j.cpc.2017.11.001.

[30] O. Olagbemi and E. de Doncker, *Scalable algorithms for multivariate integration with* PARADAPT *and CUDA (accepted)*.

[31] 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.

[32] R. Cools and A. Haegemans, "Algorithm 824:Cubpack: A package for automatic cubature, framework description", *ACM Transactions on Mathematical Software*, vol. 29, no. 3, pp. 287–296, 2003.

[33] T. Hahn, "Cuba - a library for multidimensional numerical integration", *Computer Physics Communications*, vol. 176, pp. 712–713, 2007, https://doi.org/10.1016/j.cpc.2007.03.006.

[34] M. A. Malcolm and R. B. Simpson, "Local versus global strategies for adaptive quadrature", *ACM Transactions on Mathematical Software*, vol. 1, no. 2, pp. 129–146, 1975. [Online]. Available: https://dl.acm.org/citation.cfm?doid=355637.355640.

[35] R. Cools and P. Rabinowitz, "Monomial cubature rules since 'stroud': A compilation", *J. Comp. Appl. Math.*, vol. 48, pp. 309–326, 1993.

[36] R. Cools, "Monomial cubature rules since 'stroud': A compilation - part 2", *J. Comp. Appl. Math.*, vol. 112, pp. 21–27, 1999.

[37] A. C. Genz and A. A. Malik, "An imbedded family of fully symmetric numerical integration rules", *SIAM Journal on Numerical Analysis*, vol. 20, no. 3, pp. 580–588, 1980. [Online]. Available: https://www.jstor.org/stable/2157273.

[38] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, *A Fortran-to-C Converter*. [Online]. Available: https://www.netlib.org/f2c/, (accessed: 08.02.2019).

[39] E de Doncker, K Kaugars, L Cucos, and R Zanny, "Current status of the parint package for parallel multivariate integration", in *Proceedings of Computational Particle Physics Symposium (CPP 2001)*, 2002, pp. 110 –119.

[40] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Pearson, 2010.

[41] *Center for high performance computing and big data*. [Online]. Available: https://cs.wmich.edu/hpcs/, (accessed: 08.08.2019).

[42] Penguin Computing, *Expanding access to full-performance HPC through the cloud*. [Online]. Available: https://www.penguincomputing.com/pod-hpc-cloud/, (accessed: 08.08.2019).

[43] *NVIDIA Tesla family specification comparison*. [Online]. Available: https://www.anandtech.com/show/7521/nvidia-launches-tesla-k40, (accessed: 08.08.2019).

[44] J. Sanders and E. Kandrot, *CUDA by Example*. Boston, Massachusetts: Pearson Education, 2010.

[45] *CUDA toolkit documentation*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#nvcc-command-options, (accessed: 09.07.2019).

[46] M. Evans and T Swartz, "Methods for approximating integrals in statistics with special emphasis on bayesian integration problems", *Statistical Science*, vol. 10, no. 3, pp. 254–272, 1995. [Online]. Available: https://projecteuclid.org/euclid.ss/1177009938.

[47] MPI, http://www-unix.mcs.anl.gov/mpi/index.html.

[48] 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*, Springer, (To appear) 2019.

[49] A. Sidi, "A new variable transformation for numerical integration", in *International Series of Numerical Mathematics*, vol. 112, Birkhäuser, Basel, 1993, pp. 359–373. [Online]. Available: https://doi.org/10.1007/978-3-0348-6338-4_27.

[50] ——, "Extension of a class of periodizing transformations for numerical integration", *Mathematics of Computation*, vol. 75, no. 253, pp. 327–343, 1995. [Online]. Available: https://projecteuclid.org/euclid.ss/1177009938.

[51] T. Binoth and G. Heinrich, "Numerical evaluation of multi-loop integrals by sector decomposition", *Nucl. Phys. B*, vol. 680, p. 375, 2004, hep-ph/0305234v1.

[52] *Dynamic parallelism in CUDA*. [Online]. Available: http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf, (accessed: 10.19.2019).

[53] *CUDA dynamic parallelism API and principles*. [Online]. Available: https://devblogs.nvidia.com/cuda-dynamic-parallelism-api-principles/, (accessed: 10.19.2019).

[54] *Unified memory in CUDA 6*. [Online]. Available: https://devblogs.nvidia.com/unified-memory-in-cuda-6/, (accessed: 10.19.2019).

[55] *Unified memory for CUDA beginners*. [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/, (accessed: 10.19.2019).