4-2014

Multi-Threaded Automatic Integration Using OpenMP and CUDA

Assaf

MULTI-THREADED AUTOMATIC INTEGRATION USING OPENMP AND CUDA

by

Rida Assaf

A thesis submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Master of Science
Computer Science
Western Michigan University
April 2014

Thesis Committee:

Elise de Doncker, Ph.D., Chair
John Kapenga , Ph.D.
Zijiang Yang, Ph.D.

MULTI-THREADED AUTOMATIC INTEGRATION USING OPENMP AND CUDA

Rida Assaf, M.S.

Western Michigan University, 2014

Problems in many areas give rise to computationally expensive integrals that beg the need of efficient techniques to solve them, e.g., in computational finance for the modeling of cash flows; for the computation of Feynman loop integrals in high energy physics; and in stochastic geometry with applications to computer graphics.

We demonstrate feasible numerical approaches in the framework of the PARINT multivariate integration package. The parallel environment is provided by the cluster of the High Performance Computational Science (HPCS) laboratory, with 22 (16- or 32-core) nodes, NVIDIA GPUs, and Intel Xeon Phi coprocessors.

Monte Carlo integration is implemented in CUDA C and can utilize dynamic parallelism on the K20 GPUs, which enables the CUDA kernel to launch child kernels (recursively or iteratively), which will be used to evaluate chunks of sample points. Roundoff error guards are necessary in view of the large computation.

An accurate computation of Feynman loop integrals is achieved with iterated integration using one-dimensional modules from the QUADPACK package, where the function evaluations are performed with multi-threading using OpenMP. The parallel performance is assessed for various types of integrals.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Prof. Elise de Doncker, for her critical eye and mentoring. I would also like to thank Prof. John Kapenga for his various contributions to this work, in addition to all the faculty members that supported me throughout my graduate studies, like Prof. Zijiang Yang and Prof. Steve Carr. Thank you for your continuous support. I am also grateful to all my friends that have been accompanying me on this journey. Thank you for always being there.

Special gratitude goes to my family. To my sister Mariam, my mother Najat, and my father Mahmoud, thank you for always believing in me.

Hold on to your dreams,

Unleash your imagination.

Dare to go on a different route,

And don't follow the nation.

Imagine a new world,

Saved by the new generation.

Singing songs of love,

Aiming for higher education.

Rida Assaf

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Problems in computational geometry, computational physics, computational finance, and other fields give rise to computationally expensive integrals. This work will address various parallel programming techniques and platforms used for multivariate numerical integration.

The platforms used cover the spectrum of modern parallel programming systems, including shared-memory systems and OpenMP, where the workload is distributed among threads to be executed by the cores of a processor. OpenMP is useful in cases where the program is to be parallelized incrementally. The parallelization is based on adding compiler directives (pragmas) to an existing code. Furthermore we discuss and present applications of GPGPU programming using CUDA, which proves to be very efficient for the computationally expensive problems under consideration. Although harder to program, this platform can lead to excellent parallel performance as measured by speedup of the program. The speedup (sequential time divided by parallel time of the execution) is mainly enabled by the large number of cores which is about 2500 in the recent Kepler K20 cards. In this work we do not further discuss the platform of distributed memory systems and MPI (Message Passing Interface). This platform differs from the shared memory system platform in that the workload is divided among separate processes (which may be on multiple processors), and each process may also fork a number of threads, resulting in a hybrid parallel programming platform. We are targeting the latter in ongoing projects. Each of these platforms suits best certain techniques. Algorithms that implement recursion for re-

gion partitioning are well suited for nested multi-threading on an OpenMP platform. Other techniques such as Monte Carlo simulations may be better suited for GPGPU programming, where the number of operations is huge but lightweight, which fits with the large number of lightweight cores (ALUs) available on GPU cards.

An *automatic* numerical integration algorithm is set up as a *black-box* to which the user specifies the dimension, integrand function and integration domain, a limit on the number of function evaluations for termination, a tolerance for the error and possibly other parameters. The method may be *non-adaptive* if it evaluates a fixed sequence of integration rules until a termination criterion is satisfied; or *adaptive*, where the course of the computations is governed by the behavior of the problem at hand.

The black-box algorithm yields an integral approximation

$$Q \approx I = \int_{\mathcal{D}} f(x)\, dx$$

and an absolute error estimate E, where it attempts to satisfy a criterion of the form

$$|Q - I| \leq E \leq \text{ tolerated error}$$

within the allowed number of function evaluations, or flags an error condition if the limit has been reached.

Adaptive methods are generally recommended for low to moderate dimensional problems (say, up to dimension 12), whereas Monte Carlo and number-theoretic type methods (lattice rules, Quasi-Monte Carlo Methods (QMC)) can be used in higher dimensions. Apart from the adaptive and QMC algorithms described below, we will also include re-

sults from a (sequential) crude Monte-Carlo algorithm (RCRUDE from the package MVNPACK [35]), with simple antithetic variates and a uniform (0,1) random number generator from [38].

This work will address applications of OpenMP for iterated adaptive integration methods, and applications of CUDA programming for Monte Carlo integration. A *Monte Carlo* method approximates the expected value of a stochastic process by sampling, i.e., by performing function evaluations over a large set of random points, and returns the sample average of the evaluations as the end result. The functions treated here are usually not smooth and may have singularities within the integration domain, which results in the generation of large sets of evaluation points. The rapidly evolving CUDA environment is well suited for numerical integration of high dimensional integrals, in particular by Monte Carlo or quasi-Monte Carlo methods. With some care, near peak performance can be obtained on important applications. A basis for efficient numerical integration using CUDA kernels is presented, showing several ways to leverage CUDA features and provide automatic error control. This framework allows easy extensions to multiple GPUs, clusters and clouds for addressing problems that were impractical to attack in the past.

In this thesis we examine multivariate numerical integration approaches including adaptive partitioning of the domain, quasi-Monte Carlo (QMC) and Monte Carlo (MC) techniques, which have been implemented for *automatic integration* in sequential and parallel packages [2, 34, 20, 40, 35].

These methods are at the basis of the ParInt package (developed further from ParInt 1.0 ©1999 by E. de Doncker, A. Gupta, A. Genz, R. Zanny). Over the years our interest in

these techniques for automatic numerical integration has been sparked by applications in various fields, including many in statistics (e.g., multivariate normal, t-distributions), finite elements (e.g., automotive simulations), computational finance (financial derivatives, e.g., collateral mortgage obligations, mortgage backed security problems), high energy physics (e.g., interaction cross sections), computational chemistry (e.g., Fock matrix representing the electronic structure of an atom or molecule), and computational geometry.

We discuss different features of CUDA programming along with a parallel Monte Carlo implementation for GPUs in Chapter 2, including DYNAMIC PARALLELISM which was introduced recently with cards that support compute capability 3.0 and higher.

In Chapter 3 we address a class of applications arising in (stochastic) computational geometry. For example, the "cube tetrahedron picking" problem leads to a 12-dimensional integral for the expected volume of a (random) tetrahedron inside the cube, and has important applications for tesselations, in modeling and computer graphics. This application uses the Monte Carlo technique described in Chapter 2.

For problems in high-energy physics (Feynman loop integrals) of low dimensionality (so far for $N \leq 6$), we derived a C implementation of a multi-core algorithm to evaluate the integral as an "iterated" or "repeated" integral over the given product region. Currently this method performs the outer two integrals in parallel, where the associated function evaluations (consisting of the inner integrals) are assigned to multiple threads. This is described in Chapter 4.

CHAPTER 2

Monte Carlo Integration

In this chapter we will assume that the integrand f is a single-valued function, and the integration domain $\mathcal{D}$ is the d-dimensional unit hypercube, $\mathcal{D} = [0,1]^d$. For N uniform random points $\mathbf{x}_i$, the Monte Carlo approximation,

$$Qf = \bar{f} = \frac{1}{N}\sum_{i=1}^{N} f_i, \tag{2.1}$$

with $f_i = f(\mathbf{x}_i)$, is suitable for moderate to high dimensions d, or an irregular integrand behavior or integration domain. In the latter case, the domain can be enclosed in a cube, where the integrand is set to zero outside of it. The main elements of MC methods are the error bounding and the underlying *Pseudo-Random Number Generators (PRNGs)*, which are discussed in Sections 2.1 and 2.2, respectively. In particular, Section 2.2 covers the implementation and use of parallel PRNGs in CUDA.

In a CUDA program, part of the code is executed by the CPU, and the other part is executed by the GPU. Whereas CPU code is usually implemented in functions, the part of the code that runs on the GPU is given in a *kernel* (a GPU function). Part of the responsibilities of the CPU (as the Central Processing Unit) is managing data transfers between the main memory (RAM) and the GPU memory, and in addition, it is responsible for the invocation of GPU kernels. Once the GPU kernel is invoked the control is transferred to the GPU to execute the kernel, and the CPU resumes its point of execution at the return of the kernel function.

It is worth mentioning that the number of blocks per grid and the number of threads per block, in addition to any other parameters the kernel might need, are passed by the CPU in the kernel invocation. A typical invocation of a kernel with the header

```
__global__ void dotProduct(float* x, float* y)
```

that computes the dot product of two vectors $x \cdot y$ will be of the form:

```
dotProduct<<<blocksPerGrid, threadsPerBlock>>>(x, y)
```

where *blocksPerGrid* and *threadsPerBlock* are variables representing the number of blocks per grid and the number of threads per block, respectively, in the launched kernel. Note that the keyword *__global__* indicates a GPU kernel that can be invoked by the CPU and is executed by the GPU. The keyword *__device__* on the other hand indicates a function that is executed by the GPU and can only be invoked by a GPU kernel (a *__global__* or a *__device__* function).

Section 2.4 describes *dynamic parallelism*, which became available with CUDA-5.0, for GPU cards supporting compute capability 3.0 and up. The recursive or "vertical" version in Section 2.4.1, and the iterative or "horizontal" version in Section 2.4.2 are utilized for an efficient GPU implementation of automatic integration in Section 2.4.3, which alleviates problems with memory restrictions and kernel launch overhead (see also [18]).. In Appendix C we include results showing speedups and accuracy for a Feynman loop integral arising in high energy physics [39, 28, 18]. We cover problems from computational geometry in [13], shown in Chapter 3 as applications to the Monte Carlo methods discussed in this chapter.

2.1 MC Convergence

In the following we give a derivation of the error estimate for the one-dimensional case [12]; a similar formulation can be given for $d > 1$.

Let $x_1, x_2, \ldots$ be random variables drawn from a probability distribution with density function $\mu(x)$, $\int_{-\infty}^{\infty} \mu(x)\ dx = 1$, and assume that the *expected value* of f, $\mathcal{I} = \int_{-\infty}^{\infty} f(x)\ \mu(x)\ dx$ exists. For example, $x_1, x_2, \ldots$ may be selected at random from a uniform random distribution in $[0, 1]$, $\mu(x) = 1$ for $0 \leq x \leq 1$, and 0 otherwise.

According to the Central Limit Theorem, the *variance*

$$\sigma^2 = \int_{-\infty}^{\infty} (f(x) - \mathcal{I})^2\ \mu(x)\ dx = \int_{-\infty}^{\infty} f^2(x)\ \mu(x)\ dx - \mathcal{I}^2 \tag{2.2}$$

allows for an integration error bound in terms of the probability

$$\mathcal{P}\left(Ef \leq \frac{\lambda\sigma}{\sqrt{N}}\right) = PI\ +\ \mathcal{O}(\frac{1}{\sqrt{N}}) \tag{2.3}$$

where PI represents the *confidence level* $PI = \frac{1}{\sqrt{2\pi}} \int_{-\lambda}^{\lambda} e^{-x^2/2}\ dx$ as a function of λ. For example, $PI = 50\%$ for $\lambda = 0.6745$; $PI = 90\%$ for $\lambda = 1.645$; $PI = 95\%$ for $\lambda = 1.96$; $PI = 99\%$ for $\lambda = 2.576$. In other words we can state that, e.g., with probability or confidence level of 99% the error $E \leq 2.567\sigma/\sqrt{N}$. The behavior of the error bound in (2.3) represents the $1/\sqrt{N}$ law of MC integration. Note that (2.3) assumes that σ exists, which relies on the integrand being square-integrable, even though that is not required for the convergence of the MC method [12].

To calculate the error estimate we employ the sample variance estimate

$$\begin{aligned}\bar{\sigma}^2 = \frac{1}{N-1}\sum_{i=1}^{N}(f_i - \bar{f})^2 &= \frac{1}{N-1}(\sum_{i=1}^{N} f_i^2 - 2\sum_{i=1}^{N} f_i\bar{f} + N\bar{f}^2) \\ &= \frac{1}{N-1}(\sum_{i=1}^{N} f_i^2 - 2N\bar{f}^2 + N\bar{f}^2) \\ &= \frac{1}{N-1}(\sum_{i=1}^{N} f_i^2 - N\bar{f}^2). \end{aligned} \tag{2.4}$$

We also make use of simple *antithetic variates*, which is known as a variance reduction method, where for each sample point $\mathbf{x}$ in (2.1), an evaluation is also performed at the point $\mathbf{x}^* = \mathbf{1} - \mathbf{x}$ (symmetric with respect to the centroid of the cube) [35]. Thus (2.1) becomes

$$Qf = \frac{1}{N}\sum_{i=1}^{N}\frac{f_i + f_i^*}{2}, \tag{2.5}$$

and we update the variance estimate of (2.4) as

$$\bar{\sigma}^2 = \frac{1}{N-1}\left(\sum_{i=1}^{N}\left(\frac{f_i + f_i^*}{2}\right)^2 - N\left(\frac{\sum_{i=1}^{N}(f_i + f_i^*)}{2N}\right)^2\right).$$

In view of (2.3) we can then use

$$E \leq \frac{\lambda}{\sqrt{N(N-1)}}\left(\sum_{i=1}^{N}\left(\frac{f_i + f_i^*}{2}\right)^2 - N\left(\frac{\sum_{i=1}^{N}(f_i + f_i^*)}{2N}\right)^2\right)^{\frac{1}{2}} \tag{2.6}$$

as an error estimate or bound (under certain conditions).

There are a number of additional variance reduction techniques that can be applied to Monte Carlo integration in addition to antithetic variates, such as control variates, importance sampling and stratified sampling. These can be included in a numerical integration package, such as PARINT, using the framework presented here, but are not the focus of the chapter and will not be discussed here.

2.2 Pseudo-Random Number Generators (PRNGs)

A *pseudo-random number generator (PRNG)* is a deterministic procedure to generate a sequence of numbers that behaves like a random sequence and can be tested for adherence to relevant statistical properties. Whereas function sampling for Monte Carlo integration is considered *embarrassingly parallel*, some applications will involve large sets of evaluations, thus requiring efficient high quality random number streams on various multilevel architectures containing distributed nodes, multicore processors and many-core accelerators. In particular, good pseudo-random sequences are required for multiple threads or processes, and a much larger period of the sequence may be needed in view of much intensified sampling and larger problems. Choosing new PRNGs may also necessitate validation of streams of random numbers generated by different methods being used together.

In this work we focus on two different PRNGs, mainly the CURAND library and the Philox generators in the RANDOM123 library.

2.2.1 Use of CURAND

With the NVIDIA CURAND approach, the CPU initializes a PRNG and invokes a GPU function to generate the sequence directly in the GPU memory, where it is kept for subsequent use. This avoids the overhead of having the CPU generate the sequence and move it from main memory to the GPU memory. A potential drawback with this approach arises, however, in that GPU memory is limited (e.g., to a maximum of about 5GB on the Tesla K20 cards [43]), so that the number of points needed to get the desired accuracy may exceed the available memory. In that case one may be tempted to solve the problem at hand

by doing several kernel invocations, but that is undesirable because of the kernel launch overhead and the overhead of the wait for CURAND to complete the generation of the sequence.

A CUDA C program section allocating space for `Ndim` $= N * Dim$ floats in an array `numbers` on the device, and calling the CURAND functions is shown below. This section executes on the CPU before the GPU kernel is launched.

```
//  allocate Ndim floats on device in array numbers
cudaMalloc ((void **) &numbers, Ndim * sizeof(float));
//  create pseudo-random number generator
curandCreateGenerator (&gen, CURAND_RNG_PSEUDO_DEFAULT);
//  set seed
curandSetPseudoRandomGeneratorSeed (gen, 1234ULL);
//  generate ndim random numbers on device
curandGenerateUniform (gen, numbers, Ndim);
```

Note that there are variations to the function *curandGenerateUniform* such as *curandGenerateUniformDouble*, which generates points in double precision.

2.2.2 Use of RANDOM123

As opposed to the static generation of the pseudo-random sequence at the beginning of the run by CURAND, the Philox type CBRNG of RANDOM123 allows generating the numbers on the GPU as needed, requires little memory and reduces the number of accesses to global memory. Furthermore it produces 2^{64} or more unique parallel streams of random numbers, each with a period of at least 2^{128}, and is found faster than CURAND on a single NVIDIA GPU [53].

For the CBRNG a statement of the form

```
result = CBRNGname(counter, key)
```

returns a (deterministic) value of *result* as a function of *key* and *counter*; i.e., using the same (*counter*, *key*) combination will always return the same *result*. The RANDOM123 library is implemented entirely in header files [52]. So all what is needed to start using the library is to *#include* it in the program source files, and guide the compiler to find its header files that are unpacked from the downloaded package.

Unlike CURAND, the RANDOM123 library requires no work by the CPU before the kernel launch. When the GPU kernel is launched, the counter and key of the Philox PRNG are initialized as follows:

```
philox4x32_key_t k = {{tid, 0xdecafbad}};
philox4x32_ctr_t c = {{0, 0xf00dcafe, 0xdeadbeef, 0xbeeff00d}};
```

where the use of the global thread index *tid*, which is unique to the thread, ensures the generation of unique random numbers by each thread.

Next, the contents of the counter can be set by accessing the constant array *v*:

```
c.v[0] = ... ; /* some loop-dependent application variable */
```

and executing

```
philox4x32_ctr_t r = philox4x32(c, k);
```

This generates 4 x 32 bit random numbers stored in an array in *r*, which will be unique as long as *c* and *k* are not reused. One way to access the random numbers is:

```
float current_value = u01fixedpt_open_open_32_24(r.v[0]);
```

This returns the first random number in the array; to access the second one *r.v[0]* is replaced by *r.v[1]*, and so on for the third and the fourth numbers.

Note that *u01fixedpt_open_open* is a utility C function that converts 32- or 64-bit random integers to uniformly distributed random values. The 32_24 suffix refers to the conversion of 32-bit integers to floats determined with a 24-bit mantissa (24 is replaced by 53 for doubles); *_open_open* indicates that the output range is open at both ends.

2.3 MC CUDA Kernel

2.3.1 Methods

The Monte Carlo approximation is obtained in (2.1) as the average of the function values on a set of N uniformly distributed random points. In general, an integration rule approximation with function evaluations $f_i = f(\mathbf{x}_i)$ and weights $w_i,\ 1 \leq i \leq N$ can be considered as the dot product $\mathbf{w} \cdot \mathbf{f} = \sum_{i=1}^{N} w_i f_i$. The MC "rule" has constant weights $w_i = \frac{1}{N},\ 1 \leq i \leq N$, but its accumulation can be accomplished in a CUDA kernel with a similar structure as a dot product [54].

For simplicity we first assume that the points have been generated by the CURAND PRNG library and are available on the GPU device. The GPU kernel *MonteCarlo* shown below evaluates the sum of function values and the sum of squares needed for the computation of the integral and variance estimates using antithetic variates according to (2.5) and (2.6), respectively. The kernel can be launched as:

```
//Launch the GPU Kernel
MonteCarlo<<<blocksPerGrid, threadsPerBlock>>>
              (numbers, dev_partial_q, dev_partial_e, N);
```

The function evaluation results are accumulated as follows:

(i) Each thread on the GPU performs a small number of function evaluations.

(ii) All threads in a block reduce their results to the first thread in the block.

(iii) The first thread in the block stores the reduced result in a global array on the device.

(iv) The global array is copied back to the CPU for a final reduction, to sum up the partial results computed by the blocks.

(v) The sum of the per-block partial results divided by N is the final result.

Thus the parameter *dev_partial_q* that is passed to the kernel *MonteCarlo* is stored in the GPU's global memory and will hold the per-block partial results. The computations for *dev_partial_e* are performed in a similar way. The integrand function f is implemented as a CUDA *device function* that takes a *point* as a parameter (represented as a *struct* containing the random coordinates in an array of dimension *dim*), and returns the function evaluation at that *point*.

The purpose of the *cache* arrays in the kernel is to utilize the GPU shared memory whose access is much faster than that of the GPU global memory where the array *numbers* is stored. The *cache* arrays are shared by all threads within the same block only.

In the kernel note that *threadIdx.x* represents the thread's index with respect to all indices within the current block, *blockIdx.x* gives the block's index with respect to all the blocks in the grid, and *blockDim.x* is the dimension of each block, which is the total number of threads in each block. Therefore, the global index of the thread with respect to *all* forked threads in *all* blocks is computed as *threadIdx.x* + *blockIdx.x* * *blockDim.x*.

Furthermore, as the number of evaluations may be much larger than the total number of threads, the while loop *while(tid* $< N$*)* allows assigning evaluations to the threads in a round-robin fashion; i.e., each thread performs evaluations starting with its index and going up in increments of *blockDim.x* * *gridDim.x*. Within the while loop we use *Kahan summation* as a roundoff error guard for the accumulations, which are performed in single (float) precision. Implementing the kernel entirely in double precision may not be desirable because:

- It requires Compute Capability 2.0 or higher, which few NVIDIA desktop video cards currently support.
- It may require twice the compute time and memory.
- It may not be needed to obtain the requested accuracy.

```
struct point {
        float coordinates[dim];
};

__global__ void MonteCarlo(float *numbers, float *partial_q,
                           float *partial_e, int N) {

    // Make use of the GPU shared memory
    __shared__ float cache[threadsPerBlock];
    __shared__ float error_cache[threadsPerBlock];

    // tid holds thread's global index with respect to
    // all GPU threads
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    point current_point;
    point second_point; // Point for antithetic variates
    float eval, evaluations = 0, qk = 0, y, t;
    float sum_of_squares = 0, qksq = 0, ysq, tsq;

    while (tid < N) {
        int i = 0;
        int count = 0;
        int tdim = tid * dim;
        float temp = 0;
        for(i = tdim; i < tdim + dim; i++) {
            // Set point coordinates with random numbers
            temp = numbers[i];
            current_point.coordinates[count] = temp;
            second_point.coordinates[count] = 1 - temp;
            count++;
        }
        // Call the CUDA device function
        eval = (f(current_point) + f(second_point));
        // Kahan summation for: evaluations += eval;
        y = eval - qk;
        t = evaluations + y;
        qk = (t - evaluations) - y;
        evaluations = t;

        // Kahan summation for: sum_of_squares += eval*eval;
        ysq = eval*eval - qksq;
```

```
        tsq = sum_of_squares + ysq;
        qksq = (tsq - sum_of_squares) - ysq;
        sum_of_squares = tsq;

        // Let each thread do a number of evaluations
        tid += blockDim.x * gridDim.x;
    }

    // Store the value obtained by each thread
    cache[threadIdx.x] = evaluations;
    error_cache[cacheIndex] = sum_of_squares;

    // Synchronize threads in this block
    __syncthreads();

    // For reductions, threadsPerBlock must be a power of 2
    // because of the following code. At the end of the
    // following while loop the first thread of the block
    // will hold the sum  of the results obtained by all
    // the threads in the block
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i) {
            cache[cacheIndex] += cache[cacheIndex + i];
            error_cache[cacheIndex]
                        += error_cache[cacheIndex + i];
        }
        __syncthreads();
        i /= 2;
    }

    // If this is the first thread of this block
    if (threadIdx.x == 0) {
        // Store the results of this block in the global arrays
        partial_q[blockIdx.x] = cache[0];
        partial_e[blockIdx.x] = error_cache[0];
    }
}
```

After the kernel finishes executing, the control returns to the CPU, which copies the contents of *partial_results* from the GPU to a local array declared on the CPU, by the following statements:

```
// Copy the partial results from the GPU back to the CPU
cudaMemcpy(partial_q, dev_partial_q,
           blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(partial_e, dev_partial_e,
```

```
        blocksPerGrid*sizeof(float), cudaMemcpyDeviceToHost);
```

Here *partial_q* and *partial_e* are the local arrays that hold the partial results, and the number of bytes transferred is *blocksPerGrid * sizeof(float)* since each block generates a partial result. As shown below, the partial results are summed by the CPU, which returns the integral approximation *final_result*, and an error estimate *error_est* according to (2.6).

```
// Sum up the partial results
double q = 0, e = 0, final_result, error_est;

for (int i = 0; i < blocksPerGrid; i++) {
    q += partial_q[i];
    e += partial_e[i];
}

final_result = q/(2*N);
error_est = sqrt(fabs((e/4 - q*q/(4*N))/(N-1)/N));
```

2.3.2 Numerical validation

One issue associated with the need for large numbers of evaluations is the occurrence of roundoff error. Another problem, specifically when using CURAND for generating the random numbers, is the restricted global memory on the GPU. A solution is not obtained by multiple kernel launches from the CPU in view of the kernel launch overhead. However, the memory restriction problem can be alleviated by using a RANDOM123 PRNG to generate the random numbers as needed on the GPU.

Figure 2.1 plots (on $\log_{10}$ scale) the absolute error (blue/diamonds curve) and estimated error (red/rectangles curve) of the MC approximation, as a function of ($\log_{10}$ of) the number of points N, for the integration of the function $f(\mathbf{x}) = 3x_0^2$ over the 12D unit cube. Thus

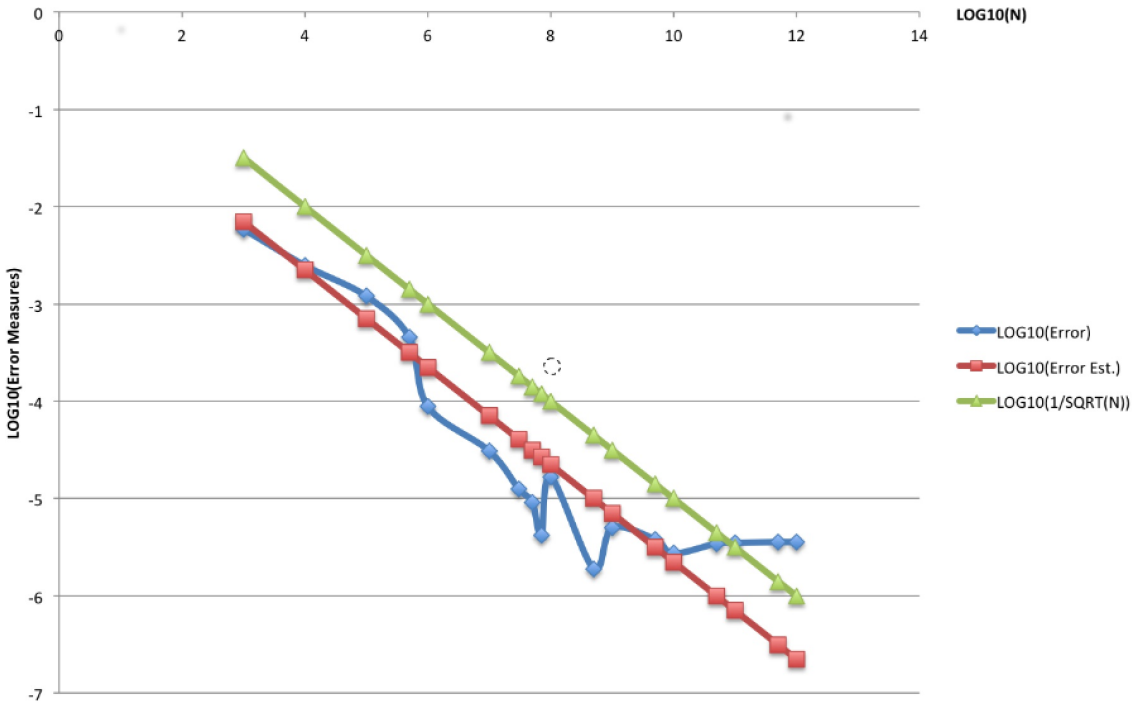

Figure 2.1 Error behavior for MC approximation using Kahan summation. Shown are: $\log_{10}$(Error), $\log_{10}$(Error Estimate) and $\log_{10}(1/\sqrt{N})$ as a function of $\log_{10} N$.

the length of the pseudo-random sequence used is $12N$ for each integration. The estimated error is based on (2.6) with $\lambda = 1$. The function $\log_{10}(1/\sqrt{N}) = -\frac{1}{2}\log_{10} N$ is also plotted (green/triangles curve). The error adheres closely to the $1/\sqrt{N}$ behavior through $N = 10^{10}$; then is stagnant or increases slightly through $N = 10^{12}$.

We used a version of the kernel in Section 2.3.1, employing the Kahan summation technique for the accumulations in single precision. For comparison we also ran the program with Kahan summation replaced by using doubles for the summation variables (only). The accuracies as well as the times were very similar. When plotted, the results were in fact indistinguishable from those of Figure 2.1.

2.4 Dynamic Parallelism

An important feature with NVIDIA's release of CUDA-5.0, that works on GPU cards supporting compute capability 3.0 and up is *Dynamic Parallelism*. We will further refer to it as *DP*.

From the NVIDIA Techbrief on *DP*, *"additional parallelism can be exposed to the GPU's hardware schedulers and load balancers dynamically, adapting in response to data-driven decisions or workloads. Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism can be more transparently expressed."* [44].

Consequently, *DP* may eliminate the need for additional kernel launches, and thus eliminates the extra kernel launch overheads that would be incurred. Basically a parent kernel can now invoke a child kernel, without CPU intervention, keeping the control completely on the GPU and eliminating the overhead of switching control between the GPU and the CPU between kernel calls. With respect to the implementation of MC, *DP* enables *chunking*, i.e., instead of launching one kernel with a huge number of points, the CPU can launch one kernel with a smaller number of points, which in turn can launch child kernels each with their chunk of a smaller number of points.

Furthermore, *DP* allows the number of forked threads to exceed the usual limit, since threads running within the parent grid can invoke a child grid, so now instead of having each thread take on an amount of work in the parent kernel, work can be off-loaded to the child kernels. A child grid needs to complete before the parent grid is considered complete.

2.4.1 Vertical (Recursive) Dynamic Parallelism

To make use of this, *Dynamic Parallelism* techniques were applied to the kernel using RANDOM123. So now instead of letting the kernel launched by the CPU do all the work, this kernel does part of the work, say a chunk of *limit* = 200 million points, and then launches another kernel to do another chunk of the work, until the total number N is reached. This type of control is displayed in the parent kernel below.

```
__global__ void MonteCarlo(float *partial_results, int runs) {

    // Make use of the GPU shared memory
    __shared__ float cache[threadsPerBlock];

    // tid holds thread's global index w.r.t to all GPU threads
    int tid = threadIdx.x + blockIdx.x * blockDim.x;

    float current_point;
    float second_point;
    float evaluations;

    //limit can be any number
    while (tid < limit) {

        Generate the current point using Random123;
        Get a second point as 1 - current_point;
        // Call the CUDA device function to evaluate it at
        //  the current points using Kahan summation and/or
        //  double accumulators
        evaluations = evaluations + f(current_point) +
                      f(second_point);
        // Let each thread do a number of evaluations
        tid += blockDim.x * gridDim.x;
    }
    // Store the value obtained by each thread
    cache[threadIdx.x] = evaluations;

    // Synchronize threads in this block
    __syncthreads();

    // For reductions, threadsPerBlock must be a power of 2
```

```
    // because of the following code. At the end of the
    // following while loop the first thread of the block
    // will hold the sum of the results obtained by all
    // the threads in the block

    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i) {
            cache[cacheIndex] += cache[cacheIndex + i];
        }
        __syncthreads();
        i /= 2;
    }
    // If this is the first thread of this block
    if (threadIdx.x == 0) {
        // Store the block's result in the global array
        partial_results[kernel_index * gridDim.x + blockIdx.x]
                  = cache[0];
    }
    // Recompute the original tid
    tid = threadIdx.x + blockIdx.x * blockDim.x;
    if(tid == 0) // the first thread of the current kernel
    {
        if(runs > 0) // and not all partitions have been run
        {
            MonteCarlo<<<gridDim.x,threadsPerBlock>>>
                   (partial_results, runs-1);
        }
    }
}
```

Here *kernel_index* is the index of the current kernel, which is used in this case to indicate the global index of the current block with respect to all blocks of all launched kernels to store the block result in the correct position of the global array. The variable *runs* stores the number of kernels that remain to be launched, based on the number of chunks, computed by the CPU as *N/limit* and passed as a parameter to the kernel, where it is decremented and passed on to the subsequent kernel launched.

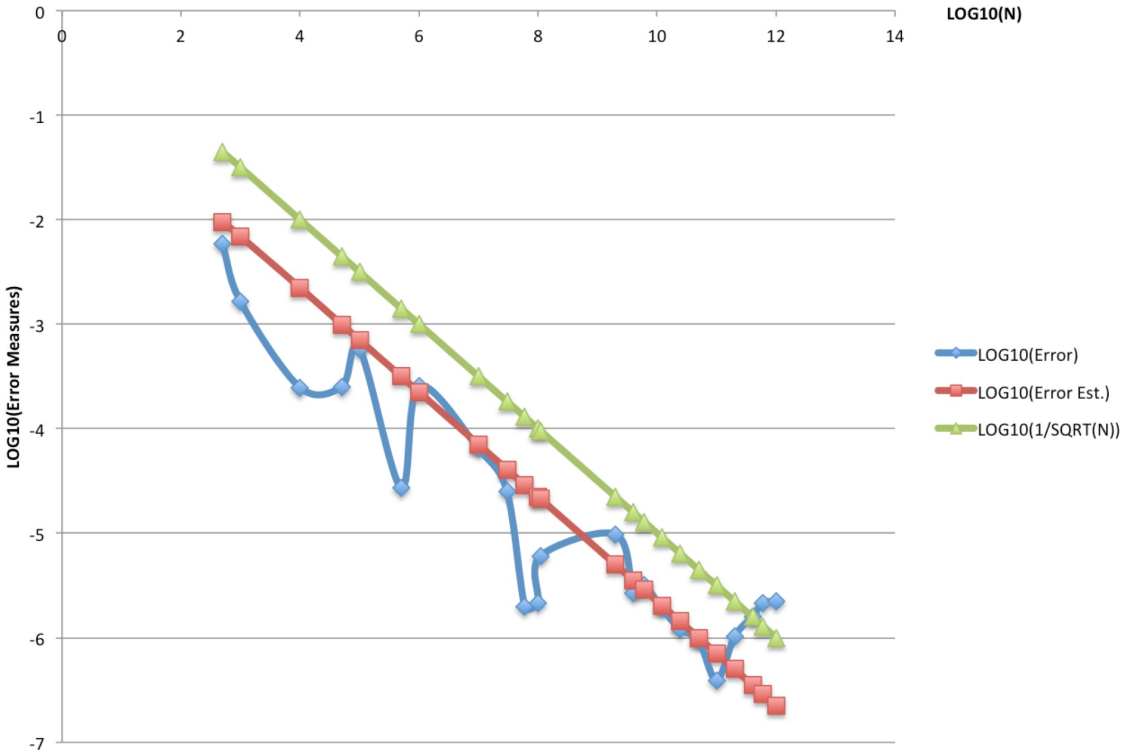

Figure 2.2 Error behavior using *horizontal DP* and *chunking*. The plot shows (for integration of $f(\mathbf{x}) = 3x_0^2$ over the 12D unit cube): $\log_{10}$(Error), $\log_{10}$(Error Estimate) and $\log_{10}(1/\sqrt{N})$ as a function of $\log_{10} N$.

A drawback of vertical *DP* where the kernels are launched in depth one at a time, is the maximum recursion depth, which currently limits the number of recursive launches to 24 and may not be sufficient for some applications.

2.4.2 Horizontal (Iterative) Dynamic Parallelism

We implemented another approach where kernel calls are performed breadth-wise, thereby creating *DP* in a horizontal/iterative way. The first kernel invokes all other necessary kernels, and the runtime takes care of scheduling them over the resources. The differences in the control, from the kernel of Section 2.4.1, are shown in the section below.

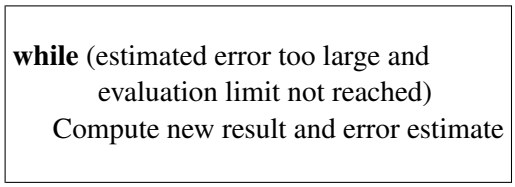

Figure 2.3 Automatic integration meta-algorithm

```
// Recompute the original tid
tid = threadIdx.x + blockIdx.x * blockDim.x;
if(tid == 0) // The first thread of this kernel
{
    while(runs > 0) // not all partitions have run
    {
        MonteCarlo<<<gridDim.x,threadsPerBlock>>>
                  (partial_results, 0);
    }

}
```

In this approach the *if* is replaced by a *while*, since all the kernel launches take place in the beginning. The variable *runs* is passed as 0 to all subsequent kernels, to ensure that no more kernels are launched than required.

Figure 2.2 gives the accuracy and estimated error obtained with horizontal *DP* using a chunk size of 200M $= 2 \times 10^8$ for $2 \times 10^9 \leq N \leq 4 \times 10^{11}$, and a chunk size of 1B $= 10^9$ for $6 \times 10^{11} \leq N \leq 10^{12} =$ 1T. No dynamic parallelism was used for $N \leq$ 200M. This version of the kernel used double precision accumulators. For the legend description see that of Figure 2.1. Figure 2.2 shows good accuracy through N in excess of 10^{11} points.

2.4.3 Automatic Integration with Dynamic Parallelism

Automatic integration was introduced in Chapter 1. We use the *Dynamic Parallelism* approach above to launch successive MC kernels on the GPU, and add a kernel (*metaMC*)

implementing the controller of the meta-algorithm. The controller is launched from the CPU and runs in one block using one thread on the GPU. It sequences the MC kernel calls which perform chunks of evaluations, until either the prescribed accuracy has been attained, or the maximum number of chunks has been computed. Pseudo-code for the *metaMC* controller is given below.

```
__global__ void metaMC(float *partial_results,
              float *partial_errors, long long int N,
              float sequence_factor, float tolerance,
              float *gpu_results, int limit)
{
    //Initialize all variables
    float partial_r = 0.0;
    float partial_e = 0.0;
    float final_result = 0.0;
    float final_error = 0.0;
    float error = tolerance+100.0; // just to enter loop
    long long int totalN = 0;
    float count = 1.0;

    while(error > tolerance && count <= limit)
    {
        MonteCarlo<<<blocksPerGrid, threadsPerBlock>>>
              (partial_results, partial_errors, N);
        // Wait for the kernel to return
        cudaDeviceSynchronize();

        //Keep track of the total number of points used
        // multiplied by 2 because of antithetic variates
        totalN += 2.0 * N;

        //Compute the value of N for the next kernel call
        N = N * sequence_factor;

        // Sum up the partial results returned by the
        // MonteCarlo kernel to the old values
        for (int i = 0; i < blocksPerGrid ;  i++) {
            partial_r += partial_results[i];
```

```
            partial_e += partial_errors[i];
        }
        //Compute the final result and error estimate
        final_result = partial_results/(totalN);
        final_error = sqrt(fabs((partial_errors/4.0 -
              partial_results*partial_results/(4.0*totalN))/
              (totalN-1.0)/totalN));

        // Set error to the correct value to be compared
        // with the tolerance next
        error = final_error;

        count++;
    }
    // Store the final results and the number of times
    // MonteCarlo was run
    gpu_results[0] = count;
    gpu_results[1] = final_result;
    gpu_results[2] = final_error;
}
```

2.5 Conclusions and the Future

It is clear that CUDA processors can be very effective in numerical integration. Important high-dimensional integrals that were previously impractical to evaluate can now be dealt with routinely, and the development of reliable open source software to provide this support will be available shortly. This is possible because of three factors:

- the advances in CUDA hardware and software by NVIDIA,
- the advances in efficient pseudo-random number generation for CUDA, and
- analysis and application of error and roundoff controls.

Rapid advances are happening on all three of these areas. NVIDIA GPUs are planned and being released with more cores, memory and features. CUDA 6.0, now available in beta, has features for coordinating 8 GPUs on a single host. Effective pseudo-random and

hopefully new quasi-random (low-discrepancy sequence) number generation on CUDA continues to evolve. Finally, general frameworks for CUDA implementation on workstations. clusters and clouds have been demonstrated and are evolving.

Applications of Monte Carlo integration to solve problems in stochastic geometry will be given in Chapter 3, after a brief description of the Quasi-Monte Carlo method outlined in the next section.

2.6 Quasi-Monte Carlo

The Quasi-Monte Carlo (QMC) method [8] samples the integration function at the (N) points of a regular lattice, which are modified by a random vector. Let the sequence of randomized (Korobov) lattice rule approximations be denoted by

$$K_N(\boldsymbol{\beta}) = \frac{1}{N}\sum_{i=1}^{N} f(\{\frac{i}{N}\mathbf{v} + \boldsymbol{\beta}\}),$$

where $\mathbf{v}$ is the lattice generator vector and $\boldsymbol{\beta}$ is a uniformly distributed random vector. By computing the rule for q random $\boldsymbol{\beta}$ vectors, the integral can be approximated as

$$I \approx \bar{K}_N = \frac{1}{q}\sum_{j=1}^{q} K_N(\boldsymbol{\beta}_j), \tag{2.7}$$

which allows for a standard error estimation

$$E_N^2 = \frac{1}{q(q-1)}\sum_{j=1}^{q}(K_N(\boldsymbol{\beta}_j) - \bar{K}_N)^2.$$

The program DKBVRC from the package MVNDST [34] is at the basis of the parallel QMC method in PARINT [20]. The algorithm calculates the $\bar{K}_N$ values of (2.7) for successively

larger numbers N of integration points until either an answer is found to the user-specified accuracy or the function count limit is reached.

CHAPTER 3

Stochastic Geometry and Monte Carlo Applications

A family of problems, to determine the expected d-dimensional (dD) volume $E[V_n(K)]$ of the polyhedron formed by n points $X_1, \ldots, X_n$, uniformly distributed in the interior of a convex body K, is considered in [64], and

$$E[V_n(K)] = \frac{1}{|K|} \int_K \cdots \int_K \text{conv}(X_1, \ldots, X_n) \, \frac{dX_1}{|K|} \cdots \frac{dX_n}{|K|}, \tag{3.1}$$

where $|K|$ denotes the d-dimensional volume of K and the integrand function is the volume $\text{conv}(X_1, \ldots, X_n)$ of the convex hull generated by the n points. For some special cases we give a parallel implementation using Monte Carlo integration in [13].

J. J. Sylvester considered the plane case for a random triangle T in a convex set K and posed the problem to determine the shape of K for which the expected value $E[X]$ is minimal or maximal, for the variable $X = \text{area}(T)/\text{area}(K)$ (see, e.g., [50]). Note that the convex body K which maximizes the expected volume of a random d-dimensional simplex, is unknown for $d \geq 3$.

As a direct application in $\mathbb{R}^3$, the mean volume of a tetrahedron whose vertices are uniformly distributed on a circular domain (*cap*) of the unit sphere is an essential element in the analysis of Poisson-Vonoroi and Delaunay tessellations [36]. Tessellations induce a partition into space-filling random polyhedra, and are used in such areas as molecular modeling, material science, pattern recognition and statistical data analysis. Tetrahedral meshes in general are found in applications including computer graphics, computer-aided

design, robotics [48], electromagnetic simulations [11] and medical imaging [56, 32, 63].

Efron [31] determined the probability with which N points chosen at random in a convex region, are the vertices of a re-entrant polyhedron, where the last r selected points lie inside the convex hull of the first $N - r$ points. For example with $r = 1$, the probability that $N = 5$ points chosen at random in a convex region form the vertices of a re-entrant polyhedron (as related to (3.1)) is $N \times E[V_4(K)]$.

Results of the $E[V_n(K)]$ problem have been derived which cover specific cases for $d \geq 3$. For example, the problem to evaluate the expected volume of a random polytope in the interior of a sphere (ball), was considered early on in [5, 1]. Tetrahedron picking in a cube (*cube tetrahedron picking*) was handled, e.g., in [64, 50]. For tessellations on the surface of a sphere, $\mathcal{S}^2$ in 3D, (*sphere-tetrahedron picking*) see [36]. The problem was solved for d-dimensional simplices, with $d+1$ random vertices on $\mathcal{S}^{d-1}$ in $d \geq 2$ dimensions [41]. The solution of the problem for tessellations within a tetrahedron (*tetrahedron-tetrahedron picking*) [6, 64, 49] refutes a conjecture from 1991 that the expected volume would be a rational number [9, 57]. This as well as *cube-* and *octahedron-tetrahedron picking* contributes to the results with respect to tessellations in non-spherical convex bodies. A recurrence relation was established for $E[V_n(K)]$ with respect to arbitrary K in [4]. Overall, closed form solutions have been restricted to special cases and difficult to obtain.

In the problem of tessellations on a spherical surface (*sphere-tetrahedron picking*), the vertices of the tetrahedron are random points on the unit spherical surface $\mathcal{S}^2$. The integral (3.1) is 8-dimensional as it involves the integration for each of the four vertices over $K = \mathcal{S}^2$ (which is 2-dimensional). $\mathcal{S}^2$ can be described in parameter form by

$(x = \sqrt{1-u^2}\cos(\theta),\ y = \sqrt{1-u^2}\sin(\theta),\ z = u)$, so that the integration is over $u \in [-1,1]$ and $\theta \in [0,2\pi]$. As suggested in [57], the dimension can be reduced from 8 to 5 by fixing the first vertex (x_1, y_1, z_1) at $(0,0,1)$ and taking the second (x_2, y_2, z_2) in the xz-plane ($y = 0$) without loss of generality. The first vertex is thus obtained by $u_1 = 1$ and the second by $\theta_2 = 0$, so that the integration over the first sphere is removed and the second integration becomes one-dimensional. The integral obtained in this form corresponds to $\bar{V}(\pi)$ of [36],

$$\bar{V}(\pi) = \frac{\int_{-1}^{1} du_2 \int_{-1}^{1} du_3 \int_{-1}^{1} du_4 \int_{0}^{2\pi} d\theta_3 \int_{0}^{2\pi} d\theta_4 \, |V_4|}{\int_{-1}^{1} du_2 \int_{-1}^{1} du_3 \int_{-1}^{1} du_4 \int_{0}^{2\pi} d\theta_3 \int_{0}^{2\pi} d\theta_4 \, 1}, \tag{3.2}$$

where $|V_4|$ is the volume of the tetrahedron, given by the determinant

$$V_4 = \frac{1}{6}\begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix}.$$

The integrand function for the 5D integral coded as a device function (to be called from the CUDA kernel) is given in Appendix A. The integration intervals $[-1,1]$ and $[0,2\pi]$ are transformed to $[0,1]$. The implementation for the 8D integral can be given with dim $= 8$ and where the first and second vertex are computed similarly to the third and fourth.

Note that (3.2) corresponds to the special case of the expected volume $\bar{V}(\alpha)$ for $\alpha = \pi$ in [36], of the spherical *cap*

$$\mathcal{S}^2(\alpha) = \{\ s(\phi,\vartheta) \mid ((\sin(\phi)\sin(\vartheta), \cos(\phi)\sin(\vartheta), \cos(\vartheta));\ -\pi \le \phi \le \pi,\ 0 \le \vartheta \le \alpha\ \}.$$

Thus the representation of $\mathcal{S}^2$ above corresponds with $u = \cos(\vartheta)$ and $\alpha = \pi$. The expected volume of a random tetrahedron with vertices on $\mathcal{S}^2$ is given by $\bar{V}(\pi) = \frac{4\pi}{105} \approx 0.11968$.

As another case, we obtain the integral

$$E[V_4(T^3)] = 6^4 \int_0^1 dx_1 \int_0^{1-x_1} dy_1 \int_0^{1-x_1-y_1} dz_1 \int_0^1 dx_2 \int_0^{1-x_2} dy_2 \int_0^{1-x_2-y_2} dz_2$$
$$\int_0^1 dx_3 \int_0^{1-x_3} dy_3 \int_0^{1-x_3-y_3} dz_3 \int_0^1 dx_4 \int_0^{1-x_4} dy_4 \int_0^{1-x_4-y_4} dz_4 \, |V_4|$$

for the expected volume of a random tetrahedron within the 3D unit simplex. The integral is transformed as shown in the code of the device function in Appendix B, so all variables range over $[0, 1]$. The exact value is given by $E[V_4(T_3)] = \frac{13}{720} - \frac{\pi^2}{15015} \approx 0.017398$ (see [6, 64]).

For cube-tetrahedron picking, $E[V_4(C_3)]$ is given as an integral over the 12D unit cube, $[0, 1]^{12}$,

$$E[V_4(C^3)] = \prod_{i=1}^{4} \left[\int_0^1 dx_i \int_0^1 dy_i \int_0^1 dz_i \right] \, |V_4|, \tag{3.3}$$

and equals $E[V_4(T^3)] = \frac{3977}{21600} - \frac{\pi^2}{2160} \approx 0.0138428$ (see [64]).

3.1 Numerical Accuracy and Parallel Performance

Table 3.1 compares the programs DCUHRE [2], DKBVRC [34] and RCRUDE [35] for the integral (3.3) of the *cube-tetrahedron* problem. Note that the problem is difficult in view of the absolute value function of the determinant integrated in 12 dimensions. The first column of Table 3.1 gives the number of integrand evaluations allowed. DKBVRC usually stays well under that, since it computes a sequence of rules each with a fixed number of points N_i at level i, and $N_i << N_j$ for $i < j$.

The parameter *key* of DCUHRE is set to 4, which selects a cubature rule of polynomial degree 7. DCUHRE starts the test with good accuracy. However, the result is only consid-

# EVALS (M)	PROGRAM	RESULT	ABS. ERR.	ABS. ERR. Est.	TIME (S)
0.1	DCUHRE	1.381719e-02	3.36e-06	1.13e-02	0.3978e-02
	DKVBRC	1.394575e-02	1.03e-04	1.48e-04	0.5185e-02
	RCRUDE	1.363727e-02	2.06e-04	1.85e-04	0.2427e-01
1	DCUHRE	1.373211e-02	1.11e-04	6.96e-03	0.4372e-01
	DKVBRC	1.384961e-02	6.84e-06	3.67e-05	0.6687e-01
	RCRUDE	1.380387e-02	3.89e-05	5.90e-05	0.1214e-01
10	DCUHRE	1.373192e-02	1.11e-04	3.32e-03	0.4391e+00
	DKVBRC	1.384268e-02	9.37e-08	5.93e-06	0.8048e+00
	RCRUDE	1.384701e-02	4.24e-06	1.87e-05	0.4403e+00
30	DCUHRE	1.374961e-02	9.32e-05	2.36e-03	0.1323e+01
	DKVBRC	1.384218e-02	5.97e-07	4.13e-06	0.2729e+01
	RCRUDE	1.384282e-02	4.11e-08	1.08e-05	0.7280e+01
50	DCUHRE	1.381719e-02	2.56e-05	2.22e-03	0.2206e+01
	DKVBRC	1.384249e-02	2.88e-07	2.18e-06	0.4112e+01
	RCRUDE	1.384290e-02	1.27e-07	8.36e-06	0.1381e+02
100	DCUHRE	1.380235e-02	4.04e-05	1.85e-03	0.4410e+01
	DKVBRC	1.382480e-02	2.37e-08	1.28e-06	0.9232e+01
	RCRUDE	1.384405e-02	1.27e-06	5.91e-06	0.2427e+02
200	DCUHRE	1.380291e-02	3.99e-05	1.57e-03	0.8827e+01
	DKVBRC	1.384280e-02	2.37e-08	1.28e-06	0.9229e+01
	RCRUDE	1.384281e-02	3.50e-08	4.18e-06	0.4863e+02
	Exact:	1.384278e-02			

Table 3.1 Comparison for *cube-tetrahedron* problem ($E[V_4(C^3)]$).

ered as good as indicated by the estimated error, which is high and decreases only slowly. The (absolute) error and error estimate are listed in the fourth and fifth column, respectively. DKBVRC performs very well for this problem, with respect to achieved accuracy and reliability. The accuracy of RCRUDE starts out low, but does increase considerably, at the cost of computation time. These programs are written in Fortran (double precision). A version of the gfortran compiler was used, and the programs were run on a MacBook Pro with 3.06 GHz Intel Core 2 Duo processor and 8GB of memory. Elapsed execution times obtained with *etime* are given in seconds.

In Table 3.2 we give results from a parallel Monte Carlo implementation, run on a cluster node with dual Intel Xeon E5-2670, 2.6GHz CPUs and 128GB of memory; and a M2090 GPU with 512 CUDA cores, 1.3GHz GPU clock rate and 5375 MB of global memory. The number of parallel blocks, and threads per block launched in the GPU runs was set to 16 and 512, respectively, for the tests described below. The problems addressed are $E[V_4(\mathcal{S}^2)],\ E[V_4(C^3)]$ and $E[V_4(T^3)]$, coded in CUDA C using single precision (float). For the timings in Table 3.2 we list the parallel time, and the speedup relative to a sequential computation which uses *erand48* for random number generation. The GPU computation is timed via the *cudaEventElapsedTime* mechanism. The speedup is the sequential time divided by the parallel time.

It emerges that for the small problem size of 100,000 sample points, the parallelization is not warranted. However, for larger sample sizes the speedups increase to values between 100 and 200, and up to 245 for the different problems. Indeed, as the number of samples increases by a factor of 10^3 (from 100,000 to 100 million), the sequential time increases by

a factor of 10^3 but the parallel time increases only by a factor of about 15. Note that the sequential times are comparable to those of the sequential program RCRUDE in Table 3.1 for the *cube-tetrahedron* problem.

# EV. (M)	PROBLEM	RESULT	ABS. ERR.	SEQ. TIME (s)	PAR. TIME (s)	SPEEDUP
0.1	$E[V_4(\mathcal{S}^2)]$	1.196144e-01	6.53e-05	2.9787e-02	1.0117e-02	2.9442e+00
	$E[V_4(C^3)]$	1.383333e-02	3.57e-05	2.1131e-02	9.8808e-03	2.1385e+00
	$E[V_4(T^3)]$	1.743181e-02	3.36e-05	2.6249e-02	9.9260e-03	2.6445e+00
1	$E[V_4(\mathcal{S}^2)]$	1.194591e-01	2.21e-04	2.9330e-01	1.0690e-02	2.7437e+01
	$E[V_4(C^3)]$	1.381283e-02	2.99e-05	2.0992e-01	1.1233e-02	1.8687e+01
	$E[V_4(T^3)]$	1.734215e-02	5.61e-05	2.5701e-01	1.1108e-02	2.3139e+01
10	$E[V_4(\mathcal{S}^2)]$	1.196825e-01	2.74e-06	2.9632e+00	2.0068e-02	1.4374e+02
	$E[V_4(C^3)]$	1.384527e-02	2.50e-06	2.0518e+00	2.3818e-02	8.6146e+01
	$E[V_4(T^3)]$	1.740328e-02	5.04e-06	2.5688e+00	2.4165e-02	1.0630e+02
30	$E[V_4(\mathcal{S}^2)]$	1.196864e-01	6.65e-06	8.8007e+00	4.2466e-02	2.0724e+02
	$E[V_4(C^3)]$	1.384298e-02	2.06e-07	6.1744e+00	5.1855e-02	1.1905e+02
	$E[V_4(T^3)]$	1.740763e-02	9.39e-06	7.7047e+00	5.2874e-02	1.4572e+02
50	$E[V_4(\mathcal{S}^2)]$	1.196707e-01	9.03e-06	1.3848e+01	6.4795e-02	2.1372e+02
	$E[V_4(C^3)]$	1.384206e-02	7.11e-07	1.1376e+01	8.0151e-02	1.4193e+02
	$E[V_4(T^3)]$	1.739901e-02	7.76e-07	1.2827e+01	8.1835e-02	1.5675e+02
100	$E[V_4(\mathcal{S}^2)]$	1.196731e-01	6.65e-06	2.9287e+01	1.1908e-01	2.4594e+02
	$E[V_4(C^3)]$	1.384286e-02	8.66e-08	2.0372e+01	1.5005e-01	1.3577e+02
	$E[V_4(T^3)]$	1.740003e-02	1.79e-06	2.5617e+01	1.5321e-01	1.6719e+02

Table 3.2 GPU results for $E[V_4(\mathcal{S}^2)]$, $E[V_4(C^3)]$ and $E[V_4(T^3)]$.

CHAPTER 4

Iterated Numerical Integration

4.1 Adaptive Region Partitioning

Many integration problems feature varying function behavior across the integration domain (e.g., singularities or peaks). An adaptive method based on region partitioning executes a number of iterations where, in each iteration, a subregion is selected, subdivided, and the total integral and error estimates are updated accordingly. Region selection is based on the local error estimate which is obtained together with the local integral approximation over each region. By intensive partitioning in the vicinity of singularities or other hot spots, it is the goal to focus on the difficult areas and sample the integrand adaptively as needed. The integration rules of the algorithm DCUHRE [2] were chosen for the adaptive methods in the PARINT parallel integration package [20, 40].

We treat Feynman loop integrals in [29] using iterated adaptive numerical integration (described in Section 4.4.1 below). Feynman loop integrals appear in higher order corrections of interaction cross section calculations in perturbative quantum field theory. The integrals are computationally intensive especially in view of singularities which may occur within the integration domain. For the treatment of threshold and infrared singularities, techniques were developed using iterated (repeated) adaptive integration and extrapolation (see, e.g., [14, 30, 16, 26]). We describe a shared memory parallelization and its application to one- and two-loop problems, by multi-threading in the outer integrations of the iterated integral [25]. The implementation is layered over OpenMP and retains the adaptive

procedure of the sequential method exactly. We give performance results for loop integrals associated with various types of diagrams including one-loop box, pentagon and two-loop vertex diagrams.

4.2 Introduction to Feynman Loop Integrals

Feynman loop integrals are generally affected by non-integrable singularities, through vanishing denominators in the interior and/or at the boundaries of the integration domain. In order to handle singularities inside the domain, the value for the integral is calculated by introducing a parameter $i\delta$ in the integrand denominator, effectively moving the singularity into the complex plane, and by taking the limit of the integral as $\delta \to 0$. We make use of numerical iterated integration and a numerical procedure for convergence acceleration or extrapolation of a sequence of integral values as δ decreases.

A loop integral for a diagram with L loops and N propagators is given in Feynman parameter space as

$$I \;=\; \frac{\Gamma\left(N-\frac{nL}{2}\right)}{(4\pi)^{nL/2}}(-1)^N \int_0^1 \cdots \int_0^1 \prod_{j=1}^{N} dx_j\, \delta(1-\sum x_j) \frac{C^{N-n(L+1)/2}}{(D-i\,\delta\, C)^{N-nL/2}}, \tag{4.1}$$

where C and D are polynomials in $x_1, x_2, \ldots, x_N$, determined by the topology of the corresponding diagram and the physical parameters. Loop integrals are generally divergent when denominators vanish in the integration region. We assume I does not suffer from other divergences, such as infrared (IR) divergence, and exists in the limit as $\delta \to 0$. To calculate the limit numerically [21], we generate a sequence of $I = I(\delta)$ for (geometrically) decreasing values of δ, and apply convergence acceleration or extrapolation to the limit with

the ε-algorithm [55, 58].

The basic method was implemented in DCM (*Direct Computational Method*) for loop integrals [26]. The technique can achieve high accuracies compared to, e.g., Monte Carlo integration, and it has been shown to successfully handle some singular problems which are problematic for other adaptive multivariate integration programs. The obtained accuracy helps controlling the accumulation of numerical error in large-scale calculations, where the precision of the end-results is fundamental for comparisons between theoretical and experimental results.

In previous work [25] a technique was introduced utilizing multi-threading for a parallel computation of the integrals. In [29] we add a parallelization on possibly multiple levels of the iterated integral, and focus further on testing the application to loop integrals. The implementation is layered over OpenMP [47] and retains the adaptive procedure of the sequential method exactly. Thus, each parallel execution performs the same subdivisions and function evaluations as the corresponding sequential run and can be regenerated; which is an important issue in the verification and debugging of parallel programs, and for avoiding useless parallel work. The parallelization is with respect to the function evaluations in the iterated method, where each function evaluation is itself an integral (except at the lowest level). This is important to guarantee a large enough granularity for the parallel procedure.

Concepts underlying automatic adaptive integration are reviewed in Section 4.3 below. Their incorporation within an extrapolation strategy as implemented in DCM is outlined in Section 4.4. Section 4.4.1 describes our parallelization of iterated integration for a multi-threaded environment. Applications to the computation of Feynman loop integrals are

```
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 into subregions
    Evaluate new subregions and update results
    Insert new subregions into priority queue
```

Figure 4.1 Adaptive Integration Meta-Algorithm

given in Section 4.5, with timing results using the OpenMP Application Program Interface (API).

4.3 Automatic Adaptive Integration

A versatile type of algorithm to implement the black-box approach performs an adaptive partitioning of the integration region as shown in Figure 4.1. At each step, a region is subdivided, integral and error estimates are computed over the subregions, and the overall result and error estimate are updated. Various strategies are possible for the selection of the region to be subdivided at each step. A *global adaptive* strategy maintains a priority queue on the subregion collection (e.g., a linked list or a heap), keyed with the local error estimates of the subregions, and selects the region with the highest (absolute) error estimate for subdivision at each step. As a result of the region selection in adaptive procedures, sample points tend to be concentrated in the vicinity of irregular behavior such as singularities, discontinuities, peaks or ridges and troughs of the integrand function.

Examples of adaptive integrators include the 1D adaptive programs of the QUAD-

PACK [51] package, the program DCUHRE [3] for multivariate integration over a cube, and programs in CUBPACK [7] for adaptive integration over cubes and simplices.

The local integral (over a subregion/interval) is approximated by a (cubature/quadrature) rule which is a linear combination of function values, of the form $\sum_{k=1}^{K} w_k f(x_1^{(k)}, x_2^{(k)}, \ldots, x_N^{(k)})$. By evaluating more than one rule over the subregion, a local error estimate can be obtained as a function of the difference between local integral approximations. For example, the (1D) program DQAGE of the QUADPACK package uses a pair of approximations, given by an r-point Gauss rule and the interlacing $(2r+1)$-point Kronrod rule. The QUADPACK user has the option of choosing one of the pairs with $r = 7, 10, 15, 20, 25$ or 30. The Kronrod rule supplies the local integral approximation, and the Gauss rule (together with the Kronrod rule) serves to obtain the local error estimate. The r-point Gauss rule is of polynomial degree of accuracy $2r-1$ (i.e., it integrates all polynomials of degrees $0, \cdots, 2r-1$ exactly, and there are polynomials of degree $2r$ which are not integrated exactly). The Kronrod rule is of polynomial degree $3r+1$ if r is even, and of degree $3r+2$ when r is odd (because of symmetry). The multivariate program DCUHRE applies an embedded sequence of cubature rules [2] for the local integral and error approximations over a subregion.

4.4 Integration and Extrapolation Strategy

The infinitesimal parameter $i\delta$ in the denominator of (4.1) prevents the integral from diverging and in that sense plays the role of a regulator. We consider the integral as a function of δ and construct a sequence of approximations to $I(\delta)$ for a decreasing sequence of δ, in order to perform an extrapolation to the limit as $\delta \to 0$. Methods for an extrapolation

of the sequence rely on the existence of an asymptotic expansion $I(\delta) \sim I + a_1\varphi_1(\delta) + a_2\varphi_2(\delta) + \cdots$, as $\delta \to 0$. Linear or nonlinear extrapolation methods can be explored under certain conditions on the functions $\varphi_\ell(\delta)$ (see, e.g., [21]). DCM implements a nonlinear extrapolation or convergence acceleration with the ε-algorithm of Wynn [55, 58], which can be applied under more general conditions than linear extrapolation. Note that $\delta = 0$ is valid in cases with unphysical kinematics, that is, where no divergence appears and no extrapolation is required, so the multi-dimensional integration only can be carried out.

For the numerical integration, we use iterated integration with the QUADPACK programs DQAGE or DQAGSE [51]. The extrapolation is applied to a sequence of $I(\delta_j)$ computed for a geometric progression of δ_j, such as $c(1.2^{-j}), j \geq 0$ where c is a constant. We have used DCM for automatic integration without explicit information about the location or nature of the singularity, for various cases of one- and two-loop diagrams including 3-point (*vertex*), 4-point (*box*) [21, 24, 23, 59], 5-point (*pentagon*) and, after reductions, 6-point (*hexagon*) diagrams [17]; as well as two-loop *self-energy* [61], *double box*, *ladder* and *crossed vertex* [22, 15, 14, 62, 16, 60] diagrams with masses. The techniques were further incorporated for semi-automatic calculations of loop integrals with infrared divergences, in [14, 30].

4.4.1 Parallel numerical iterated integration

In this section we apply iterated 1D integration over a finite d-dimensional product region, i.e., the integral I can be written as

$$I = \int_{\alpha_1}^{\beta_1} dx_1 \int_{\alpha_2}^{\beta_2} dx_2 \ldots \int_{\alpha_d}^{\beta_d} dx_d \, f(x_1, x_2, \ldots, x_d), \tag{4.2}$$

and the limits of integration may in general be functions, $\alpha_j = \alpha_j(x_1, x_2, \ldots, x_{j-1})$ and $\beta_j = \beta_j(x_1, x_2, \ldots, x_{j-1})$. The integration over the interval $[\alpha_j, \beta_j]$ will be performed with a 1D adaptive integration code, and different 1D integration programs may be applied in different directions $1 \leq j \leq d$.

If an interval $[a, b]$ arises in the subdivision of $[\alpha_j, \beta_j]$ for $1 \leq j < d$, then the local integral approximation over $[a, b]$ is of the form

$$\int_a^b dx_j \; F(c_1, \ldots, c_{j-1}, x_j) \;\approx\; \sum_{k=1}^{K} w_k F(c_1, \ldots, c_{j-1}, x^{(k)}), \tag{4.3}$$

where the w_k and $x^{(k)}, 1 \leq k < K$, are the weights and abscissae of the local rule scaled to the interval $[a, b]$ and applied in the x_j-direction. For $j = 1$ this is the outer integration direction. The function evaluation

$$F(c_1, \ldots, c_{j-1}, x^{(k)}) = \int_{\alpha_{j+1}}^{\beta_{j+1}} dx_{j+1} \ldots \int_{\alpha_d}^{\beta_d} dx_d \; f(c_1, \ldots, c_{j-1}, x^{(k)}, x_{j+1}, \ldots, x_d), \; 1 \leq k \leq K, \tag{4.4}$$

is itself an integral in the $x_{j+1}, \ldots, x_d$-directions, and is computed by the method(s) for the inner integrations. For $j = d$, (4.4) is the evaluation of the integrand function

$$F(c_1, \ldots, c_{d-1}, x^{(k)}) = f(c_1, \ldots, c_{d-1}, x^{(k)}).$$

Note that the error incurred in the inner integration is subject to an error control condition and will contribute to the overall integration error. Work on the integration error interface is reported in [33, 37, 19].

Subsequently, in Section 4.5, we give results obtained with 1D repeated integration by the program DQAGE from QUADPACK, where the local integration is performed with the

(7, 15)- or the (10, 21)-points Gauss-Kronrod pairs. We will refer to the latter as GK15 and GK21, respectively.

The inner integrals given by (4.4), which appear in the local rule sum of (4.3), are independent and can thus be evaluated in parallel, by multiple threads. In previous work [25, 27] we reported results of this parallelization applied to the outer direction of integration using OpenMP. Important properties of this method include that: 1) the granularity of the parallel integration is large, especially when the inner integrals given by (4.4) are of dimension greater than two; 2) apart from possibly the order of the summation in the local rule evaluation, the parallel calculation is the same as the sequential evaluation.

Nesting of parallel constructs is also feasible, for a parallel evaluation of the rule in multiple coordinate directions of the iterated integral. The current implementation allows for a nested parallelization in the outer and the next to outer level (corresponding to the x_1 and x_2 directions) in the iterated integral (4.2)); this can be extended to more or different levels. This feature is beneficial when the inner integral evaluations at the x_1-level are very time-consuming, so the work can be further distributed to new threads, on systems where a large number of threads are available.

4.5 Results for Feynman Loop Integrals

We implemented the methods of Section 4.4.1 as a multi-threaded application layered over OPENMP [47], and report timing results on: a multi-core Intel workstation with Xeon X5680 3.33GHz, 6-core (12 logical)-core CPU (*"minamivt005"*); the Hitachi SR16000/M1 with POWER7 3.83GHz, 32-core processors at KEK; and the Intel cluster (*"thor"*) at

WMU with Xeon E5-2670 processors, 16 cores per node.

OPENMP was chosen because of the ability to parallelize the function evaluations, and portability, meaning most (if not all) recent computers now support multi-threading on shared memory systems.

The compilers used were: on *minamivt005*, the Intel *ifort* Fortran XE compiler with flag *-openmp*; the f90 compiler with flag *-omp* on SR16000; and the gfortran compiler with flag *-fopenmp* on *thor*. Furthermore we implemented a C version of the package, which is being tested with gcc *-fopenmp*.

In all cases the main loop in the rule routines DQK15 and DQK21 of QUADPACK was parallelized (for the outer two integrations), by means of an OpenMP *omp parallel do* for the Gauss and Kronrod rule evaluations. On comparing loop schedules, we found that the *dynamic* loop schedule outperforms the default *static* schedule in selected test problems, on *minamivt005* and on *thor*. In the *static* schedule, the participating threads are assigned their loop iterations at the beginning of the loop, in a round-robin fashion; thus the assigned iterations are determined only by the thread ID and the total number of iterations. The *dynamic* schedule adjusts the assignments at runtime. The threads can request more work at the completion of their previously assigned iterations, which is beneficial for load balancing in cases where the integrand behavior over the subregion is non-uniform. On SR16000 the *dynamic* schedule is not available, and neither is nested loop scheduling.

In [25] we reported timings for a one-loop non-scalar box problem, $M_4(f, g; \delta)$ from [20]. Presently we give timings for loop integral computations corresponding to the one-loop pentagon, and two-loop ladder vertex diagrams and parameters specified below. The run-

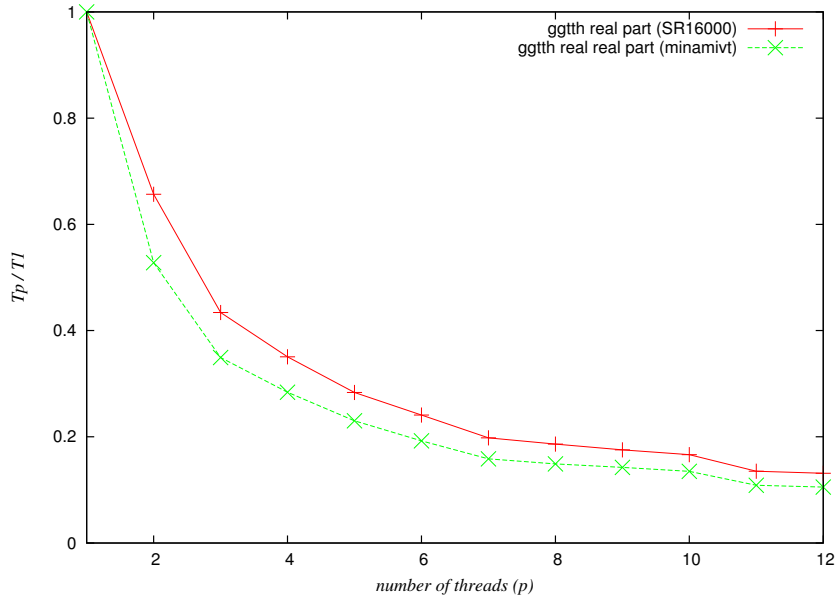

Figure 4.2 ρ_p vs. # threads p on SR16000 and *minamivt005* for $\gamma\gamma \to t\,\bar{t}\,H$ (1-loop pentagon)

ning times are total elapsed times spent in the integration code. The time for the extrapolation is negligible compared to the integration time. Unless otherwise stated, the adaptive integration method uses the GK21 pair in the x_1 direction and the GK15 pair in all other integration directions.

For the one-loop pentagon diagram corresponding to the interaction $\gamma\gamma \to t\,\bar{t}\,H$, the integral (4.1) with $L = 1,\ N = 5,\ n = 4$, reduces to

$$I = -2\int_0^1 dx \int_0^{1-x} dy \int_0^{1-x-y} dz \int_0^{1-x-y-u} du\ \frac{1}{(D - i\delta)^3}$$

via the transformation $x_1 = 1,\ x_2 = 1 - x - y - z - u,\ x_3 = y,\ x_4 = z,\ x_5 = u$ (and omitting the factor $\frac{1}{(4\pi)^{nL/2}}$). Here

$$D = x_1 m_1^2 + x_2 m_2^2 + x_3 m_3^2 + x_4 m_4^2 + x_5 m_5^2 - x_1 x_2 s_1 - x_2 x_3 s_2 - x_3 x_4 s_3$$

$$- x_4 x_5 s_4 - x_5 x_1 s_5 - x_1 x_3 s_{12} - x_2 x_4 s_{23} - x_3 x_5 s_{34} - x_4 x_1 s_{45} - x_5 x_2 s_{51},$$

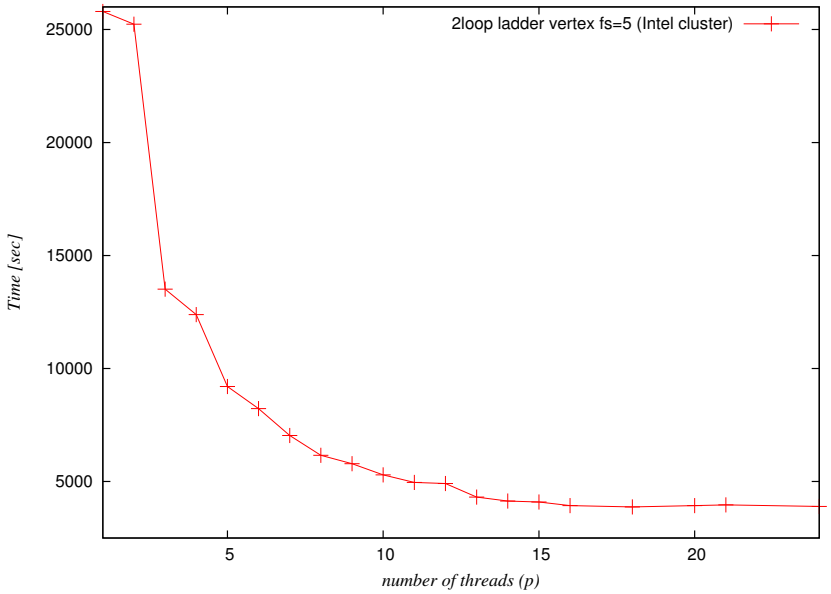

Figure 4.3 Time T_p vs. # threads p on WMU Intel cluster for 2-loop ladder vertex, $k^2/m_t^2 = 5$

where the m_j are masses, the p_j are external momenta, $s_j = p_j^2$ and $s_{ij\ldots} = (p_i + p_j + \ldots)^2$, $0 \leq i, j \leq 5$. Fig. 4.2 displays the running time T_p scaled by the sequential time T_1, i.e., $\rho_p = T_p/T_1$ as a function of the number of threads p, on *minamivt* and SR16000. The sequential time for the integration decreases significantly particularly on the Intel system (from $T_1 = 2726$ sec to $T_{12} = 287$ sec). The extrapolation converges to about 5 digits.

The integral for the two-loop ladder vertex is (4.1) with $N = 6$, $n = 4$, and C, D given

by

$$D = -C(x_1(p_1^2 - m_1^2) + x_2(p_2^2 - m_2^2) - x_3 m_3^2 + x_4(p_1^2 - m_4^2) + x_5(p_2^2 - m_5^2) - x_6 m_6^2)$$

$$+ C_1(x_5^2 p_2^2 + x_4^2 p_1^2 - x_4 x_5(p_3^2 - p_1 * 2 - p_2^2)) + C_2(x_2^2 p_2^2 + x_1^2 p_1^2 - x_1 x_2(p_3^2 - p_1^2 - p_2^2))$$

$$+ 2x_2 x_3 x_5 p_2^2 + 2x_1 x_3 x_4 p_2^2 - x_3(x_2 x_4 + x_1 x_5)(p_3^2 - p_1^2 - p_2^2),$$

$$x_6 = 1 - x_1 - x_2 - x_3 - x_4 - x_5,$$

$$C_1 = x_1 + x_2 + x_3,$$

$$C_2 = 1 - x_1 - x_2,$$

$$C = x_3(1 - x_1 - x_2 - x_3) + (x_1 + x_2)(1 - x_1 - x_2). \tag{4.5}$$

The parameters are $m_1 = m_2 = m_4 = m_5 = m_t$ (= 150 GeV, top quark mass), and $m_3 = m_6 = m_Z$ (= 91.17 GeV, Z-boson mass). Fig. 4.3 gives the running time T_p (on *thor*) for the real part integral (in seconds) as a function of the number of threads p, for parameter $k^2 = 112500$ with $k^2/m_t^2 = 5$ (denoted by *fs* = 5 in the key of Fig. 4.3). It is shown that the parallelization yields a spectacular savings in time for this compute-intensive problem (from $T_1 = 25800$ sec to $T_{24} = 3897$ sec).

Fig. 4.4 gives corresponding times for the case $k^2/m_t^2 = 1$ which is part of the unphysical region for this problem; it is thus obtained without extrapolation by setting $\delta = 0$. The two curves in Fig. 4.4 represent the computations carried out with the GK15 and GK21 Gauss-Kronrod pairs in the x_1 dimension; the pair GK15 is used in the other dimensions. The (5D) integral is computed using the transformation derived in [21].

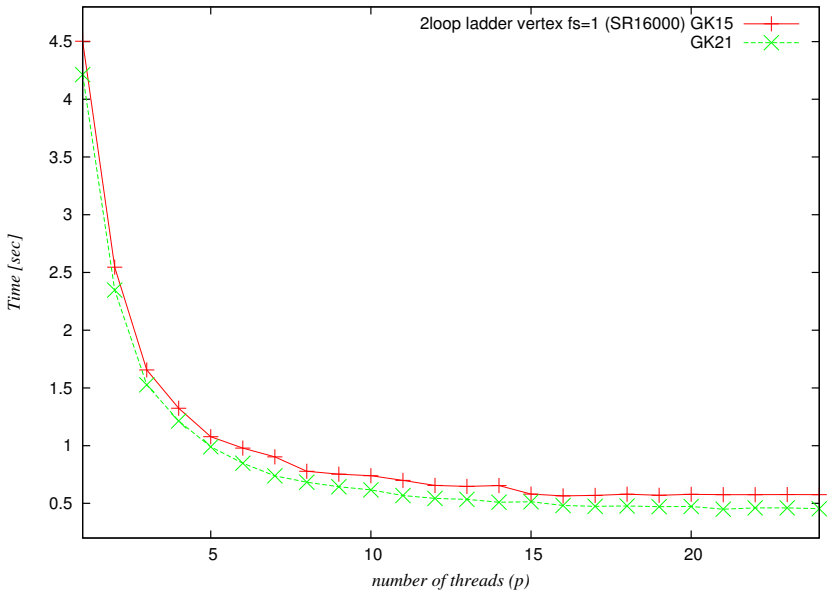

Figure 4.4 Time T_p vs. # threads p on SR16000 for 2-loop ladder vertex, $k^2/m_t^2 = 1$

4.6 Concluding Remarks

In addition to the lowest order or tree level, higher order corrections are required for an accurate theoretical prediction of the cross section of particle interactions, and for checking its agreement with the data observed at colliders. Feynman loop diagrams need to be taken into account, necessitating the calculation of *loop integrals*. While packages based on symbolic integration are available for one-loop integrals, symbolic reductions may lead to large sets of integrals, and analytic integration is not generally possible. The intensive nature of the direct computation performed by DCM motivates a parallelization of the individual problems, in particular for 3D and higher-dimensional integrals.

We have developed a multi-core parallelization on one or more function evaluation levels in the iterated integration procedure. In this paper we give timing results of the method implemented in OPENMP, which show good parallel performance for problems

executed on a Xeon X5680, 6- (12 logical)-core CPU workstation, and furthermore on the (Hitachi) SR16000 system at KEK (Japan), and the Intel cluster (based on Xeon E5-2670 processors with 16 cores per node) at WMU. The Intel cluster at WMU and the SR16000 are suited for multi-threaded computations on each node and message passing between nodes.

Our future work will naturally involve implementations and testing for other and more computational intensive loop integrals.

Future plans include hybrid (distributed shared memory) parallelizations using OpenMP and MPI. This opens the way for more parallelism, since processes can run on multiple nodes, and each can fork a group of threads for function evaluations. Two approaches are being followed in this domain. In one version, the evaluations in the outer dimensions are distributed statically on different processes and each process forks the necessary number of threads to evaluate these points. In another version, more load balancing and dynamic evaluation techniques are being used, having a master-slave parallelism approach, where the master controls the distribution of work to the slave processes, and the slave processes keep requesting more points as needed.

CHAPTER 5

Conclusions

We presented adaptive iterated methods and Monte Carlo (MC) methods as approaches to solve compute-intensive numerical integration problems, and their implementations on modern parallel programming platforms (shared memory systems via OPENMP, and GPGPU programming via CUDA). We tested the adaptive techniques in Feynman loop applications, and the Monte Carlo methods for the challenging stochastic geometry problem to determine the expected volume $E[V_4(K)]$ of a tetrahedron with uniformly distributed vertices in the interior of a cube (C^3), tetrahedron (T^3) and the surface of the 3D sphere ($\mathcal{S}^2$). In the numerical tests we monitored the accuracy and efficiency of the methods as the number of sample points or the complexity of the function increases.

This work is part of efforts to port the PARINT package to a hybrid parallel environment, by adding efficient multi- and many-core capabilities (using OPENMP [47], CUDA [10] and OPENACC [46]), to its distributed (MPI [45, 42]) computations. In a future stage we intend to develop efficient kernels for QMC as well as MC on GPUs and Xeon Phi accelerators. We also plan on extending our applications in stochastic computational geometry, for further approximations of the expected d-dimensional volume $E[V_n(K)]$ of the convex hull of n random points in a convex body.

A CUDA C device function (integrand evaluation) for spherical surface tessellation

```
const int dim = 5;
struct point {
    float coordinates[dim];
};
__device__ float f( point p ) {
    float u2,u3,u4,t3,t4;
    float x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4;
    float d1,d2,d3,d4,f0;
    const float pi2 = atan(1.0)*8;

    u2 = 2*p.coordinates[0]-1;
    u3 = 2*p.coordinates[1]-1;
    u4 = 2*p.coordinates[2]-1;
    t3 = p.coordinates[3]*pi2;
    t4 = p.coordinates[4]*pi2;
    x1 = 0;
    y1 = 0;
    z1 = 1;
    x2 = sqrt(1.0-u2*u2);
    y2 = 0;
    z2 = u2;
    x3 = sqrt(1.0-u3*u3)*cos(t3);
    y3 = sqrt(1.0-u3*u3)*sin(t3);
    z3 = u3;
    x4 = sqrt(1.0-u4*u4)*cos(t4);
    y4 = sqrt(1.0-u4*u4)*sin(t4);
    z4 = u4;

    d1 = x2*y3*z4+y2*z3*x4+z2*x3*y4-z2*y3*x4-x2*z3*y4-y2*x3*z4;
    d2 = x1*y3*z4+y1*z3*x4+z1*x3*y4-z1*y3*x4-x1*z3*y4-y1*x3*z4;
    d3 = x1*y2*z4+y1*z2*x4+z1*x2*y4-z1*y2*x4-x1*z2*y4-y1*x2*z4;
    d4 = x1*y2*z3+y1*z2*x3+z1*x2*y3-z1*y2*x3-x1*z2*y3-y1*x2*z3;

    f0 = d1-d2+d3-d4;
    return fabs(f0)/6.0;
}
```

CUDA C device function (integrand evaluation) for spherical surface tessellation

B CUDA C device function (integrand evaluation) for tessellation in tetrahedron

```
const int dim = 12;
struct point {
    float coordinates[dim];
};
__device__ float f( point p ) {
    float x1,y1,z1,x2,y2,z2,x3,y3,z3,x4,y4,z4;
    float d1,d2,d3,d4,f0;

    x1 = p.coordinates[0];
    y1 = (1.0-x1)*p.coordinates[1];
    z1 = (1.0-x1-y1)*p.coordinates[2];
    x2 = p.coordinates[3];
    y2 = (1.0-x2)*p.coordinates[4];
    z2 = (1.0-x2-y2)*p.coordinates[5];
    x3 = p.coordinates[6];
    y3 = (1.0-x3)*p.coordinates[7];
    z3 = (1.0-x3-y3)*p.coordinates[8];
    x4 = p.coordinates[9];
    y4 = (1.0-x4)*p.coordinates[10];
    z4 = (1.0-x4-y4)*p.coordinates[11];

    d1 = x2*y3*z4+y2*z3*x4+z2*x3*y4-z2*y3*x4-x2*z3*y4-y2*x3*z4;
    d2 = x1*y3*z4+y1*z3*x4+z1*x3*y4-z1*y3*x4-x1*z3*y4-y1*x3*z4;
    d3 = x1*y2*z4+y1*z2*x4+z1*x2*y4-z1*y2*x4-x1*z2*y4-y1*x2*z4;
    d4 = x1*y2*z3+y1*z2*x3+z1*x2*y3-z1*y2*x3-x1*z2*y3-y1*x2*z3;

    f0 = fabs(d1-d2+d3-d4);
    f0 = f0*(1.0-x1)*(1.0-x1-y1)*(1.0-x2)*(1.0-x2-y2);
    f0 = f0*(1.0-x3)*(1.0-x3-y3)*(1.0-x4)*(1.0-x4-y4);
    return f0*1296;
}
```

CUDA C device function (integrand evaluation) for tessellation in tetrahedron

C Times and speedup results for a Feynman loop integral

As an application we treated a Feynman (two-)loop integral, which was previously considered in [39, 28]. This type of problem is important for the calculation of the cross section of particle interactions in high energy physics.

Table 2 lists the integral approximation, absolute error and error estimate, and the par-

allel time of a GPU computation on Kepler K20, using the MC kernel with RANDOM123 as the PRNG, and Kahan summation on the GPU. The results in the bottom part of the table (for $N \geq 4 \times 10^8$) are obtained using the horizontal dynamic parallelism strategy with a chunk size of 200 million. For the smaller values of N in the top part of the table, the chunk size is set equal to N, so no child kernels are launched. For these values of N, the execution time is compared to that of a corresponding sequential calculation where erand48() is called to generate the pseudo-random sequence on the CPU. Speedups of near full peak performance are observed. Note that the error decreases to 9.9e-08 at $N = 4 \times 10^{10}$. The integrand function is given below.

N	RESULT	ABS. ERR.	ERR. EST.	SEQ. TIME (ms)	PAR. TIME (ms)	SPEEDUP
10^3	7.725290e-02	8.1e-03	1.1e-02	3.4e-01	2.5e-01	1.37e+00
5×10^3	8.380065e-02	1.6e-03	4.8e-03	1.7e+00	2.5e-01	6.64e+00
10^4	8.317693e-02	2.2e-03	3.2e-03	3.4e+00	2.6e-01	1.31e+01
5×10^4	8.428590e-02	1.1e-03	1.6e-03	1.7e+01	2.8e-01	5.92e+01
10^5	8.421736e-02	1.1e-03	1.1e-03	3.3e+01	3.2e-01	1.04e+02
5×10^5	8.467325e-02	6.8e-04	5.3e-04	1.6e+02	6.1e-01	2.70e+02
10^6	8.497359e-02	3.8e-04	3.9e-04	3.3e+02	9.7e-01	3.38e+02
5×10^6	8.516875e-02	1.8e-04	1.8e-04	1.6e+03	3.6e+00	4.49e+02
10^7	8.535279e-02	1.4e-06	1.4e-04	3.3e+03	6.9e+00	4.71e+02
5×10^7	8.528537e-02	6.6e-05	5.9e-05	1.6e+04	3.3e+01	4.89e+02
10^8	8.534221e-02	9.2e-06	4.2e-05	3.3e+04	6.6e+01	4.95e+02
2×10^8	8.533084e-02	2.1e-05	3.0e-05	6.5e+04	1.3e+02	4.94e+02
4×10^8	8.532608e-02	2.5e-05	2.1e-05		2.6e+02	
10^9	8.531650e-02	3.5e-05	1.3e-05		6.6e+02	
2×10^9	8.533230e-02	1.9e-05	9.6e-06		1.3e+03	
4×10^9	8.534547e-02	5.9e-06	6.8e-06		2.6e+03	
10^{10}	8.535301e-02	1.6e-06	4.3e-06		6.6e+03	
4×10^{10}	8.535130e-02	9.9e-08	2.2e-06		2.6e+04	
10^{11}	8.534716e-02	4.2e-06	1.4e-06		6.6e+04	

Table 5.1 Times and speedup results for a Feynman loop integral.

```
__device__ float f(point p) {

        float x1,x2,x3,x4,x5,x6,x7;
        float dd,cc,dd3,t1,f0;

        x1 = p.coordinates[0];
        x2 = (1.0-x1)*p.coordinates[1];
        x3 = (1.0-x1-x2)*p.coordinates[2];
        x4 = (1.0-x1-x2-x3)*p.coordinates[3];
        x5 = (1.0-x1-x2-x3-x4)*p.coordinates[4];
        x6 = (1.0-x1-x2-x3-x4-x5)*p.coordinates[5];
        x7 = 1.0-x1-x2-x3-x4-x5-x6;

        t1 = x1+x2+x3;
        cc =(x1+x2+x3+x4+x5)*(x1+x2+x3+x6+x7) - t1*t1;
        dd = cc*(x1+x2+x3+x4+x5+x6+x7)
          -((x1*x2*x4 + x1*x2*x5 + x1*x2*x6 + x1*x2*x7
          + x1*x5*x6 + x2*x4*x7 - x3*x4*x6)
          + (x3*(-x4*x6 + x5*x7))
          + (x3*(x1*x4 + x1*x5 + x1*x6 + x1*x7 + x4*x6 + x4*x7))
          + (x3*(x2*x4 + x2*x5 + x2*x6 + x2*x7 + x4*x6 + x5*x6))
          + (x1*x4*x5 + x1*x5*x7 + x2*x4*x5 + x2*x4*x6
          + x3*x4*x5 + x3*x4*x6 + x4*x5*x6 + x4*x5*x7)
          + (x1*x4*x6 + x1*x6*x7 + x2*x5*x7 + x2*x6*x7
          + x3*x4*x6 + x3*x6*x7 + x4*x6*x7 + x5*x6*x7));
        dd3 = dd*dd*dd;
        if(dd3 == 0) return(0);

        f0 = 2.0*cc/dd3
             *(1.0-x1)*(1.0-x1-x2)*(1.0-x1-x2-x3)
             *(1.0-x1-x2-x3-x4)*(1.0-x1-x2-x3-x4-x5);
        return f0;
}
```

BIBLIOGRAPHY

1. AFFENTRANGER, F. The expected volume of a random polytope in a ball. *J. Microscopy 151* (1988), 277–287.
2. BERNTSEN, J., ESPELID, T. O., AND GENZ, A. An adaptive algorithm for the approximate calculation of multiple integrals. *ACM Trans. Math. Softw. 17* (1991), 437–451.
3. BERNTSEN, J., ESPELID, T. O., AND GENZ, A. Algorithm 698: DCUHRE-an adaptive multidimensional integration routine for a vector of integrals. *ACM Trans. Math. Softw. 17* (1991), 452–456.
4. BUCHTA, C. Distribution-independent properties of the convex hull of random points. *J. Theor. Prob. 3* (1990), 387393.
5. BUCHTA, C., AND MÜLLER, J. Random polytopes in a ball. *J. Appl. Prob. 21* (1984), 753–762.
6. BUCHTA, C., AND REITZNER, M. The convex hull of random points in a tetrahedron: Solution of Blaschke's problem and more general results. *J. Reine Angew. Math 536* (2001), 1–29.
7. COOLS, R., AND HAEGEMANS, A. Algorithm 824:Cubpack: A package for automatic cubature, framework description. *ACM Transactions on Mathematical Software 29*, 3 (2003), 287–296.
8. CRANLEY, R., AND PATTERSON, T. N. L. Randomization of number theoretic methods for multiple integration. *SIAM J. Numer. Anal. 13* (1976), 904–914.
9. CROFT, H. T., FALCONER, K. J., AND GUY, R. K. *Unsolved Problems in Geometry*. Springer-Verlag, New York, 1991. B5. Random Polygons and Polyhedra.
10. CUDA. NVIDIA Developer Zone, https://developer.nvidia.com/category/zone/cuda-zone.
11. DARDENNE, J., SIAUVE, N., VALETTE, S., PROST, R., AND BURAIS, N. Impact of tetrahedral mesh quality for electromagnetic and thermal simulations. COMPUMAG, *Florianopolis: Brazil* (2009), 1044–1045.
12. DAVIS, P. J., AND RABINOWITZ, P. *Methods of Numerical Integration*. Academic Press, New York, 1975.
13. DE DONCKER, E., AND ASSAF, R. Gpu integral computations in stochastic geometry. *Lecture Notes in Computer Science 7972* (2013), 129–139. VII Workshop Comp. Geometry and Applics. (CGA).
14. DE DONCKER, E., FUJIMOTO, J., HAMAGUCHI, N., ISHIKAWA, T., KURIHARA, Y., LJUCOVIC, M., SHIMIZU, Y., AND YUASA, F. Extrapolation algorithms for infrared divergent integrals, 2010. arXiv:hep-ph/1110.3587; PoS (CPP2010)011.

15. DE DONCKER, E., FUJIMOTO, J., HAMAGUCHI, N., ISHIKAWA, T., KURIHARA, Y., SHIMIZU, Y., AND YUASA, F. Transformation, reduction and extrapolation techniques for Feynman loop integrals. *Springer Lecture Notes in Computer Science (LNCS) 6017* (2010), 139–154.

16. DE DONCKER, E., FUJIMOTO, J., HAMAGUCHI, N., ISHIKAWA, T., KURIHARA, Y., SHIMIZU, Y., AND YUASA, F. Quadpack computation of Feynman loop integrals. *Journal of Computational Science (JoCS) 3*, 3 (2011), 102–112.

17. DE DONCKER, E., FUJIMOTO, J., KURIHARA, Y., HAMAGUCHI, N., ISHIKAWA, T., SHIMIZU, Y., AND YUASA, F. Recursive box and vertex integrations for the one-loop hexagon reduction in the physical region. In *XIV Adv. Comp. and Anal. Tech. in Phys. Res.* (2010). PoS (ACAT10) 073.

18. DE DONCKER, E., KAPENGA, J., AND ASSAF, R. Monte carlo automatic integration with dynamic parallelism in CUDA. In *Numerical Computations with GPUs* (2014), V. Kindratenko, Ed., Springer.

19. DE DONCKER, E., AND KAUGARS, K. Dimensional recursion for multivariate adaptive integration. *Procedia Computer Science 1* (2010), 117–124.

20. DE DONCKER, E., KAUGARS, K., CUCOS, L., AND ZANNY, R. Current status of the ParInt package for parallel multivariate integration. In *Proc. of Computational Particle Physics Symposium (CPP 2001)* (2001), pp. 110–119.

21. DE DONCKER, E., SHIMIZU, Y., FUJIMOTO, J., AND YUASA, F. Computation of loop integrals using extrapolation. *Computer Physics Communications 159* (2004), 145–156.

22. DE DONCKER, E., SHIMIZU, Y., FUJIMOTO, J., AND YUASA, F. Numerical computation of a non-planar two-loop vertex diagram. In *LoopFest V, Stanford Linear Accelerator Center* (2006). http://www-conf.slac.stanford.edu/loopfestv/proc/present/DEDONCKER.pdf.

23. DE DONCKER, E., SHIMIZU, Y., FUJIMOTO, J., AND YUASA, F. Computation of Feynman loop integrals. *PAMM - Wiley InterScience Journal 7*, 1 (2007).

24. DE DONCKER, E., SHIMIZU, Y., FUJIMOTO, J., YUASA, F., CUCOS, L., AND VAN VOORST, J. Loop integration results using numerical extrapolation for a non-scalar integral. *Nuclear Instruments and Methods in Physics Research A 539* (2004), 269–273. hep-ph/0405098.

25. DE DONCKER, E., AND YUASA, F. Parallel computation of Feynman loop integrals. In *IUPAP C20 Conference on Computational Physics (CCP 2011)* (2012), vol. 402, The Journal of Physics: Conf. Series Dec. 2012. doi:10.1088/1742-6596/454/1/012082.

26. DE DONCKER, E., AND YUASA, F. Toward automatic regularization for Feynman loop integrals in perturbative quantum field theory. *Measurements in Quantum Mechanics* (2012), ISBN 978–953–51–0058–4.

27. DE DONCKER, E., AND YUASA, F. Adaptive control in multi-threaded iterated integration. *Journal of Physics: Conf. Series 410* (2013), 012047. doi:10.1088/1742-6596/410/1/012047.

28. DE DONCKER, E., AND YUASA, F. Distributed and multi-core computation of 2-loop integrals, 2013. Accepted.

29. DE DONCKER, E., YUASA, F., AND ASSAF, R. Multi-threaded adaptive extrapolation procedure for feynman loop integrals in the physical region. *Journal of Physics: Conf. Ser. 454*, 012082 (2013). doi:10.1088/1742-6596/454/1/012082.

30. DE DONCKER, E., YUASA, F., AND KURIHARA, Y. Regularization of IR-divergent loop integrals. *Journal of Physics: Conf. Ser. 368*, 012060 (2012).

31. EFRON, B. The convex hull of a random set of points. *Biometrica 52* (1965), 331–343.

32. FEDOROV, A., AND CHRISOCHOIDES, N. Tetrahedral mesh generation for non-rigid registration of brain mri: Analysis of the requirements and evaluation of solutions. In *Proceedings of the 17th International Meshing Roundtable* (2008), R. V. Garimella, Ed., Springer Berlin Heidelberg. DOI 10.1007/978-3-540-87921-3.

33. FRITSCH, F. N., KAHANER, D. K., AND LYNESS, J. N. Double integration using one-dimensional adaptive quadrature routines: A software interface problem. *ACM TOMS 7*, 1 (1981), 46–75.

34. GENZ, A. MVNDST, 1998. http://www.math.wsu.edu/faculty/genz/software/fort77/mvndstpack.f.

35. GENZ, A. MVNPACK, 2010. http://www.math.wsu.edu/faculty/genz/software/fort77/mvnpack.f.

36. HEINRICH, L., KÖRNER, MEHLHORN, N., AND MUCHE, L. Numerical and analytical computation of some second-order characteristics of spacial Poisson-Vonoroi tesselations. *Statistics 31* (1998), 235–259.

37. KAHANER, D., MOLER, C., AND NASH, S. *Numerical Methods and Software*. Prentice Hall, 1988.

38. L' EQUYER, P. Combined multiple recursive random number generators. *Operations Research 44* (1996), 816–822.

39. LAPORTA, S. High-precision calculation of multi-loop Feynman integrals by difference equations. *Int. J. Mod. Phys. A 15* (2000), 5087–5159. arXiv:hep-ph/0102033v1.

40. LI, S., KAUGARS, K., AND DE DONCKER, E. Distributed adaptive multivariate function visualization. *International Journal of Computational Intelligence and Applications (IJCIA) 6*, 2 (2006), 273–288.

41. MILES, R. E. Isotropic random simplices. *Adv. Appl. Probability 3* (1971), 353–382.

42. MPI. http://www-unix.mcs.anl.gov/mpi/index.html.

43. NVIDIA. Tesla Product Literature, http://www.nvidia.com/object/tesla_product_literature.html.

44. NVIDIA. http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf.

45. OPEN-MPI. http://www.open-mpi.org.

46. OPENACC. http://www.openacc-standard.org.

47. OPENMP. http://www.openmp.org.

48. PACH, J., AND SHARIR, M. *Combinatorial Geometry and Its Algorithmic Applications: The Alcala Lectures*. Amer Mathematical Society, 2009. ISBN-10: 0821846914, ISBN-13: 978-0821846919.

49. PHILIP, J. The average volume of a random tetrahedron in a tetrahedron. *TRITA MAT 06 MA 02* (2006). http://www.math.kth.se/˜johanph/ev.pdf.

50. PHILIP, J. The expected volume of a random tetrahedron in a cube, 2007. http://www.math.kth.se/˜johanph/ETC.pdf.

51. PIESSENS, R., DE DONCKER, E., ÜBERHUBER, C. W., AND KAHANER, D. K. *QUADPACK, A Subroutine Package for Automatic Integration*, vol. 1 of *Springer Series in Computational Mathematics*. Springer-Verlag, 1983.

52. SALMON, J. K., AND MORAES, M. A. Random123: a library of counter-based random number generators. http://deshawresearch.com/resources_random123.html, and http://www.thesalmons.org/john/random123/releases/1.06/docs, Random123-1.06 Documentation.

53. SALMON, J. K., MORAES, M. A., DROR, R. O., AND SHAW, D. E. Parallel random numbers: As easy as 1, 2, 3. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)* (2011). Best Paper Award.

54. SANDERS, J., AND KANDROT, E. *CUDA by Example - An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2011. ISBN: 978-0-13-138768-3.

55. SHANKS, D. Non-linear transformations of divergent and slowly convergent sequences. *J. Math. and Phys. 34* (1955), 1–42.

56. SITEK, A., HUESMAN, R. H., AND GULLBERG, G. T. Tomographic reconstruction using an adaptive tetrahedral mesh defined by a point cloud. *IEEE Transactions on Medical Imaging 25*, 9 (2006), 1172–1179.

57. WEISSTEIN, E. W. Sphere tetrahedron picking. In *MathWorld – A Wolfram Web Resource* (2013). http://mathworld.wolfram.com/topics/RandomPointPicking.html.

58. WYNN, P. On a device for computing the $e_m(s_n)$ transformation. *Mathematical Tables and Aids to Computing 10* (1956), 91–96.

59. YUASA, F., DE DONCKER, E., FUJIMOTO, J., HAMAGUCHI, N., ISHIKAWA, T., AND SHIMIZU, Y. Precise numerical results of IR-vertex and box integration with extrapolation. In *XI Adv. Comp. and Anal. Tech. in Phys. Res.* (2007). PoS (ACAT07) 087, arXiv:0709.0777v2 [hep-ph].

60. YUASA, F., DE DONCKER, E., HAMAGUCHI, N., ISHIKAWA, T., KATO, K., KURIHARA, Y., AND SHIMIZU, Y. Numerical computation of two-loop box diagrams with masses. *Journal Computer Physics Communications 183* (2012), 2136–2144.

61. YUASA, F., ISHIKAWA, T., FUJIMOTO, J., HAMAGUCHI, N., DE DONCKER, E., AND SHIMIZU, Y. Numerical evaluation of Feynman integrals by a direct computation method. In *XII Adv. Comp. and Anal. Tech. in Phys. Res.* (2008). PoS (ACAT08) 122; arXiv:0904.2823.

62. YUASA, F., ISHIKAWA, T., KURIHARA, Y., FUJIMOTO, J., SHIMIZU, Y., HAMAGUCHI, N., DE DONCKER, E., AND KATO, K. Numerical approach to calculation of Feynman loop integrals, 2010. arXiv:1109.4213v1 [hep-ph]; PoS (CPP2010)011.

63. ZHANG, Y., WANG, W., LIANG, X., BAZILEVS, Y., HSU, M.-C., KVAMSDAL, T., BREKKEN, R., AND ISAKSEN, J. High-fidelity tetrahedral mesh generation from medical imaging data for fluid-structure interaction analysis of cerebral aneurysms. *Computer Modeling in Engineering & Sciences (CMES)*, 2 (2009), 131–148.

64. ZINANI, A. The expected volume of a tetrahedron whose vertices are chosen at random in the interior of a cube. *Monatsh. Math. 139* (2003), 341–348. DOI 10.1007/s00605-002-0531-y.