Monte Carlo Simulations on Xeon Phi: Offload and Native Mode

Bryar Shareef
Western Michigan University, bryarmustafa@gmail.com

Follow this and additional works at: http://scholarworks.wmich.edu/masters_theses
Part of the Computer Sciences Commons

Recommended Citation

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
Monte Carlo Simulations on Xeon Phi: Offload and Native Mode

by

Bryar Shareef

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

Thesis Committee:
Elise de Doncker, Ph.D., Chair
Ajay Gupta, Ph.D
John Kapenga, Ph.D
In high performance computing, Monte Carlo methods are widely used to solve problems in various areas of computational physics, finance, mathematics, electrical engineering and many other fields. We have designed Monte Carlo methods to compute Feynman loop integrals in high energy physics, and to solve problems in stochastic geometry with applications to computer graphics, such as the tetrahedron picking problem leading to 12 dimensional integrals.

The Intel Xeon Phi is a coprocessor based on a Many Integrated Core (MIC) architecture to gain extreme performance. We have used two different modes, ”offload” and ”native”, to implement the simulations. In offload mode, the main program resides on the host system and the functions are executed on MIC. In native mode, the program is fully executed on MIC.

In this thesis, we compare the performance of our applications running on Intel Xeon Phi, in terms of time and speedup, with the sequential execution on the CPU. The comparison results between the different modes are then shown further in the thesis. In addition, the applications are designed in both single and double precision to show the difference with respect to time.
Acknowledgements

I would like to thank all the people who have helped me in the journey of my graduate studies at Western Michigan University. I would like to express my deepest gratitude to my advisor and Professor, Dr. Elise de Doncker, for her continuous support in my Master’s thesis and research work. Also I appreciate her patience, motivation, enthusiasm, and immense knowledge about the course and field of study.

Furthermore, I would like to give special thanks to my thesis committee: Dr. Ajay Gupta and Dr. John Kapenga, for their encouragement, support, guidance and insightful comments on the thesis.

Finally, I would like to convey my thanks to my Parents, wife and two lovely daughters, Lya and Lenya, for encouraging me in all of my pursuits and for inspiring me to follow my dreams.

Bryar Shareef
## Contents

Acknowledgements ii  
List of Tables v  
List of Figures vi

### 1 Introduction

1.1 Description of MIC 1
1.2 Application Modes 3
  1.2.1 Offload 3
  1.2.2 Native 4
  1.2.3 Symmetric Execution 5
1.3 Intel Compilers 6
1.4 OpenMP 8
1.5 MKL Library 8
  1.5.1 MKL on CPU 9
    1.5.1.1 Automatic Offload 9
    1.5.1.2 Compiler Assisted Offload 9
  1.5.2 MKL on MIC 10
  1.5.3 Random Number Generator 10
    1.5.3.1 Vector Statistical Library (VSL) 12
1.6 Performance Metrics 15

### 2 Monte Carlo Integration 17

2.1 Monte Carlo Integration 17
2.2 VSL Supported Parallel Computation 20
  2.2.1 Splitting Blocks 20
  2.2.2 Leapfrog 22
  2.2.3 Using Different Parameter Sets (BRNG sets) 22
2.3 Kahan Compensation/Summation 23
# 3 Applications

3.1 Monte Carlo Integration ............................................. 27
3.2 High Energy Physics .............................................. 27
3.3 Computational Geometry ......................................... 30
3.4 Cube-Tetrahedron $E[V_4(C^3)]$ ................................ 30
3.5 The Calculation .................................................... 31

# 4 Results and Discussion

4.1 System Architecture ............................................... 33
4.2 Feynman Loop Integrals Application ............................. 34
  4.2.1 Crossed Diagram Feynman Loop Integral .................. 34
    4.2.1.1 Offload .............................................. 34
    4.2.1.2 Native ............................................... 36
    4.2.1.3 Comparison ......................................... 39
  4.2.2 Ladder Diagram Feynman Loop Integral .................... 41
    4.2.2.1 Offload .............................................. 41
    4.2.2.2 Native ............................................... 43
    4.2.2.3 Comparison ......................................... 46
4.3 Cube-Tetrahedron Picking Expected Value $E[V_4(C^3)]$ ...... 48
  4.3.1 Offload ....................................................... 48
  4.3.2 Native ....................................................... 50
  4.3.3 Comparison .................................................. 52

# 5 Conclusion and Future Work

5.1 Thesis Summary .................................................... 55
5.2 Conclusions ........................................................ 56
5.3 Future Work ........................................................ 57
  5.3.1 Mode of Programming ........................................ 57
  5.3.2 Trends in Technology ....................................... 60

References ................................. 61
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Offload times and speedup results for crossed diagram Feynman loop integral</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Native times and speedup results for 2-loop Feynman loop integral (crossed box diagram)</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Times and speedup results for the 2-loop crossed diagram Feynman loop function integral</td>
<td>40</td>
</tr>
<tr>
<td>4.4</td>
<td>Offload times and speedup results for the ladder diagram integral</td>
<td>42</td>
</tr>
<tr>
<td>4.5</td>
<td>Native times and speedup results for the ladder diagram integral</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>Times and speedup results for the ladder diagram integral</td>
<td>47</td>
</tr>
<tr>
<td>4.7</td>
<td>Times and speedup results of native and offload versions for the cube-tetrahedron picking expected value $E[V_4(C^3)]$</td>
<td>49</td>
</tr>
<tr>
<td>4.8</td>
<td>Accuracy, times and the speedup results of the native version for the cube-tetrahedron expected value $E[V_4(C^3)]$</td>
<td>51</td>
</tr>
<tr>
<td>4.9</td>
<td>Time and speedup results for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points N</td>
<td>53</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Intel Xeon Phi card</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Intel Xeon Phi coprocessor cores architecture</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Offload compute model: P1() and P2() are functions executed on Xeon Phi</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Native compute model</td>
<td>5</td>
</tr>
<tr>
<td>1.5</td>
<td>Symmetric compute model</td>
<td>6</td>
</tr>
<tr>
<td>1.6</td>
<td>Pseudo-random number: MCG59</td>
<td>11</td>
</tr>
<tr>
<td>1.7</td>
<td>Quasi-random number: Sobol</td>
<td>12</td>
</tr>
<tr>
<td>1.8</td>
<td>Speedup</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>Area under the curve</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>Two dimensional function</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Block splitting</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Splitting number of points (N)</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Block splitting method</td>
<td>21</td>
</tr>
<tr>
<td>2.6</td>
<td>Leapfrog example, the number of subsequences $P = 3$.</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>Example of different parameter sets with MT2203</td>
<td>23</td>
</tr>
<tr>
<td>2.8</td>
<td>Naive recursive summation</td>
<td>24</td>
</tr>
<tr>
<td>2.9</td>
<td>Kahan summation</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>2-loop box ladder diagram, 2-loop box crossed diagram$^1$</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Cube-tetrahedron picking</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>The speedup of the offload results for the 2-loop crossed diagram</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Feynman loop integral as a function of the number of points N</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>Serial and parallel time of the offload results for the 2-loop crossed</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>diagram Feynman loop integral as a function of the number of points N</td>
<td>36</td>
</tr>
<tr>
<td>4.4</td>
<td>Single and double precision parallel time of the offload results for the</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>2-loop crossed diagram Feynman loop integral as a function of the number of</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>points N</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>Native speedup results for the 2-loop crossed diagram Feynman loop integral</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>as a function of the number of points N</td>
<td>38</td>
</tr>
</tbody>
</table>
4.6 Native parallel single and double precision time for the 2-loop crossed diagram Feynman loop integral as a function of the number of points N

4.7 Comparison between the speedup of native and offload versions for the 2-loop crossed Feynman loop integral as a function of the number of points N

4.8 Comparison between the time of the native and offload versions for the 2-loop crossed Feynman loop integral as a function of the number of points N

4.9 Speedup of the offload results for the 2-loop ladder diagram integral as a function of the number of points N

4.10 Serial and parallel time of the offload results for the 2-loop ladder diagram integral as a function of the number of points N

4.11 Single and double precision parallel time of the offload results for the 2-loop ladder integral as a function of the number of points N

4.12 Native speedup results for the 2-loop ladder diagram integral as a function of the number of points N

4.13 Native serial and parallel time for the 2-loop ladder diagram integral as a function of the number of points N

4.14 Single and double precision parallel time for the 2-loop ladder diagram integral as a function of the number of points N

4.15 Comparison between the speedup of native and offload versions for the 2-loop ladder diagram integral as a function of the number of points N

4.16 Comparison between the time of the native and offload results for the 2-loop Ladder integral as a function of the number of points N

4.17 Speedup of the offload version for the cube-tetrahedron expected value $E[V_4(C^3)]$

4.18 The sequential and parallel time results of the offload version for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points N

4.19 Speedup results of the native version for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points N

4.20 Times of the native and offload versions for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points N

4.21 Comparison between the speedup of native and offload versions for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points N

4.22 Comparison between the time of the native and offload versions for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points N

5.1 Symmetric mode using MPI between nodes

5.2 Symmetric hybrid OpenMP+MPI with offload
Chapter 1

Introduction

1.1 Description of MIC

The Intel Xeon Phi coprocessor adheres to the MIC architecture, which is the Intel Many Integrated Core architecture comprised of many Intel Central Processing Unit cores on a single chip. The Intel Xeon Phi is compatible with the x86 technology and consists of up to 61 cores, which are based on the first Intel Pentium (P54C) generation from back in 1995 [1]. The physical memory size is 6, 8, or 16GB, and can be accessed with an aggregate memory bandwidth of 352GB/s [2]. Each core has 64KB L1 cache including 32KB L1 data cache and 32KB L1 instruction cache, and 512KB L2 cache interconnected by a ring bus. Furthermore, there are eight memory controllers each supporting two channels, thus up to 16 GDDR5 memory elements are supported, Figure 1.1 shows 16 GDDR5s on MIC. MIC cores contain a 512-bit wide vector unit and the MIC has 32 vector registers where each vector unit is capable of packing 16 32-bit elements or 8 64-bit elements of floating point or integer values, which help to achieve high performance [3].

In addition, 61 cores can support 244 threads because each core can generate 4 threads at a time. The programmer can use 1, 2, 3, or 4 threads per core at a time.
MIC has its own Linux operating system and the programmer can write code that runs on the coprocessor, or use the offload mode where the CPU can participate. C, C++, and Fortran are supported on the Xeon Phi with some extensions that help parallelism in programs and APIs such as OpenMP, MPI, Cilk, and TBB. The major advantage of MIC is that it can run the code and the above mentioned APIs without significant changes to the programs. On the list of the top 500 supercomputers of the 75 systems currently using accelerator/coprocessor technology, there are 25 systems with Intel MIC-Xeon Phi technology. Tianhe-2 (the number one fastest supercomputer) and Stampede (the seventh) use Intel Xeon Phi coprocessor [4]. Figure 1.2 and 1.1 illustrate the Intel Xeon Phi architecture.

![Figure 1.1: Intel Xeon Phi card](image)
1.2 Application Modes

MIC has three different programming modes. Programmers can use MIC as a separate node or as a coprocessor. Figures 1.3-5 depict the offload, native and symmetric compute models application modes [1].

1.2.1 Offload

Offload is a heterogeneous programming mode. The CPU manages the exchange of data between the host (CPU side) and MIC (device side) and specifies a portion of the code to execute on MIC. This mode is very common for using MIC to parallelize applications. Here the CPU is primary and the MIC is secondary. MIC is a coprocessor that cooperates with the CPU to speed up the program. It can be used for partial parallelism of a serial program. When the CPU reaches a designated parallel region it sends it to run on the MIC. The CPU part may or may not be further parallel, e.g., using OpenMP or MPI. Use of non-shared
memory between the CPU and the Xeon Phi may be challenging. In offload mode, memory management can be implemented in two ways. The first is a virtual shared memory model that depends on a third system level runtime support mechanism to maintain data coherency between the host and coprocessor shared memory address space. The second option is data marshalling, where the compiler runtime generates and sends a data buffer over to the coprocessor, controlled by parameters provided in the application [1]. Using the offload mode is similar to compiler directives in OpenMP: the code segments to be run on the MIC card are specified with offload pragmas. The compiler runtime copies data from and to the host and the coprocessor around the offload block specified by the pragmas.

Figure 1.3: Offload compute model: P1() and P2() are functions executed on Xeon Phi

1.2.2 Native

In native mode a program executes entirely on the Intel Xeon Phi coprocessor. The on-card operation mode is supported by the MIC's operating system, which is based on a Linux operating system. The programmer can transfer programs and
data to the MIC card manually by using scp (secure copy), and then executing the program on the device with ssh (secure shell). Native mode needs minimal code modifications. The required properties of an application to run in native mode are: data parallelism, use of parallel algorithms, and application scalability. Non transferring data between the CPU and MIC in native mode has the advantage that it saves time by avoiding need for communications and transmissions [3]. The option –mmic must be added for compilation of the native program. This compiles the code to be executed on the MIC only. However, it requires shared libraries such as libiomp5.so are needed.

![Native compute model](image)

**Figure 1.4: Native compute model**

### 1.2.3 Symmetric Execution

In this mode the Intel Xeon Phi and the CPU are used as two separate nodes of a cluster in a heterogeneous environment. These communicate through message passing interface (MPI). We have not implemented this mode in our applications because so far the company that provided our cluster (Penguin Computing) does not support MPI for Xeon Phi on more than one node.
1.3 Intel Compilers

A compiler is a program that translates source code written in a computer programming language (high level) into another computer programming language (machine code). The Intel Xeon Phi is an Intel device and thus supports Intel Compilers. The Intel C++ compiler, also known as icc or icpc, is a group of C and C++ compilers from Intel available for OS X, Linux, Windows and Intel based Android devices. The Intel compiler is used on all Intel-based devices for C and C++ [1].

The compiler compiles the program, links all compiled objects and produces an executable file. To do so it invokes all the options specified in a configuration file and searches for header files in known locations. It sets 16 bytes as the strictest alignment constraint for structures. It also displays error and warning messages. It uses American National Standards Institute (ANSI) with extensions, and performs standard optimizations on the Operating Systems that support all the characters in
Unicode (i.e., multi-byte) format. The compiler processes all file names containing these characters [1].

Furthermore, we can use Intel compilers (icc, icpc, or ifort) on the command line. When the compiler is invoked with icc, it builds C source files using all necessary C libraries and C include files, and also links the object files.

If a C++ source file is used with icc, then it is compiled as a C++ file. Furthermore when the compiler is invoked with icpc it builds C++ source files using all necessary C++ libraries and C++ include files.

The Intel Composer XE tools and the SDK suite are also available for developing on the Intel Xeon Phi platform. The bundle includes the C/C++ and Fortran compilers. Its related runtime libraries like OpenMP, Pthreads, etc., include debugging tools and the Math Kernel Library (MKL). This suite can be used for applications that are compatible with Intel Xeon processors and Xeon Phi coprocessors. It also supports several parallel programming models for Intel Xeon Phi such as Intel Cilk Plus, Intel Threading Building Block (TBB), OpenMP and Pthreads [1]. The compiler can build both offload and cross-compiled code for Intel Xeon Phi, and can be invoked from the command line or used in the Eclipse development environment.

The syntax to compile a C/C++/Fortran file is as follows:

<compiler name> [options] file1 [file2]

Where:
<compiler name> is one of the compiler names like icc, icpc, ifort mentioned above;
[options] are options that are passed to the compiler to control code-generation, optimization, output file name, type and path. If no options are specified, the compiler applies default options such as O2 for default optimization [1].

For example, for compiling a C program:
icc -mkl -openmp -mmic filename.c -o executable_name
for compiling a C++ program:
icpc -mkl -openmp -mmic filename.cpp -o executable_name

1.4 OpenMP

OpenMP stands for Open Multi Processing. This is an API for parallel programming on a shared memory model that works with C/C++ and Fortran. Data can be shared, and private. Shared data can be accessed by all threads whereas private data is private to a particular thread. OpenMP makes it very simple to program parallel applications because it is not a new programming language. It consists of compiler directives, runtime libraries, and environment variables that influence run-time behavior and can be added to a sequential program [5]. Tim Mattson from Intel in his online lecture says “The reason of developing OpenMP is that a group of computer scientists and programmers thought that writing high parallel applications using Pthreads would be very hard”. When the program is executed, it starts from the main program sequentially until it reaches OpenMP directives, which make the program parallel and create a team of threads; thus the sequential program waits until all the threads are finished with their work. Basically, this applies the mechanism of fork-join in order to parallelize the application. In native mode on the Intel Xeon Phi, OpenMP can be used without any changes with Fortran, C, and C++. In our simulations we use OpenMP with C in the native and offload mode versions [3][6][5].

1.5 MKL Library

MKL, the Math Kernel Library, is a very powerful computational math library, which helps to accelerate the performance of the application and gain effective
speedup. It consist of highly optimized math functions and is used for engineering, science, financial and other applications. The Math Kernel Library resides on the CPU as well as the MIC [7].

1.5.1 MKL on CPU

MKL is an Intel library that includes optimized routines and functions for Intel CPUs or compatible Intel CPU based devices that support multi-threading. Intel MKL has many function groups but the most useful library for Monte Carlo simulation is VSL (Vector Statistical Library), which produces pseudo-random and quasi-random number sequences with a variety of options [7].

1.5.1.1 Automatic Offload

Another powerful feature of the Intel MKL library is automatic offload. This enables Intel MKL library routines benefit from the Intel Xeon Phi coprocessor transparently and automatically, without any changes in the code. The only step the programmer should take is enabling automatic offload, which can be done by exporting the variable ($MKL\_MIC\_ENABLE = 1$) [7].

1.5.1.2 Compiler Assisted Offload

This model takes advantage of the offload pragmas and the Intel compiler to transfer offloaded data to and from the device. In the offload region the programmer should determine the input and output data for the MIC card. The compiler links the MKL library on the MIC card, which enables the compiler to use the needed libraries during run time. The advantage of using compiler assisted offload is data persistence because it helps to reuse the data on the coprocessor instead of
transferring data more often. Some compiler pragmas and directives can manage compiler assisted offload [7].

1.5.2 MKL on MIC

Provision of the Math Kernel Library on MIC makes the Intel Xeon Phi coprocessor much more flexible for the programmer. Automatic offload is not possible in this case.

1.5.3 Random Number Generator

Random number sequences have been used in science for many years, such as in mathematics, physics, finance and computer science. Using an appropriate random number generator is a prerequisite for the success of the Monte Carlo method. One distinguishes true random numbers that can be found in physical phenomena and non-true pseudo-random numbers, which are produced by an algorithm. With a True-Random Number Generator (TRNG), we can truly generate random numbers from the physical world, for example, by measuring sources of atmospheric noise. This can be used for practical purposes and cannot be predetermined or guessed. In the computer era, producing pseudo-random numbers is easier and faster. A Pseudo-Random Number Generator (PRNG) is a substitute for a true-random number generator. Even though a pseudo-random number sequence may resemble a true-random sequence, it is initialized by a seed of the PRNG and will repeat after a certain period. PRNGs are useful in computations for testing and solving problems in such areas as physics, finance, and mathematics because of the speed in the generation and reproducibility of the sequences.
A Quasi-Random Number Generator (QRNG), produces low discrepancy and highly uniform sequences at a fast rate. The uniformity of the sequence is measured by means of its discrepancy, and provides an advantage over PRNG sequences for some applications.

The period of the Basic Random Number Generator (BRNG) (which may be PRNG or QRNG) is one of the most critical features that affects its quality. For instance, two of the BRNGs that are used for the simulations discussed later on are MCG59 and Sobol. MCG59 is used in Pseudo-Monte Carlo simulation, and has a period of about $2^{57}$. Figure 1.6 shows an example obtained with 4000 random numbers of MCG59, which takes seconds to generate. Sobol is used for Quasi-Monte Carlo simulations, and has a period of about $2^{32}$. These BRNGs are useful because of the speed, efficiency, and small memory requirements for keeping the generator state. Figure 1.7 gives an example of a sequence of 4000 quasi-random Sobol numbers.
1.5.3.1 Vector Statistical Library (VSL)

The Vector Statistical Library (VSL) is a useful library of the Math Kernel Library (MKL) for producing vectors of pseudo- and quasi-random numbers. VSL contains generators conforming to various types of statistical probability distributions to improve the performance of programs based on highly optimized BRNGs (Basic Random Number Generators), such as uniform, normal (Gaussian), exponential, or Poisson distributions, for different architectures of Intel processors. VSL is suitable for building applications in parallel because it is a thread-safe library, with different configurations for parallel computations.

In addition, VSL can split the original sequence using three sub-sequence techniques; block splitting, the leapfrog method, and BRNG sets [8] (see Chapter 2 for details). The Vector Statistical Library (VSL) consists of the following sub-routines [8]:
• Stream Initialization Subroutine. A specific abstract source of random numbers is a random stream. A particular stream can generate a random number sequence by connecting a stream to a particular BRNG and selecting particular initial values, so the random sequence can be determined in advance. Every random number stream in VSL is identified by a universal stream state descriptor. The allocated memory space contains information on the BRNG and its current state. Additionally it holds specific information for the status of the leapfrog and/or skip-ahead methods.

• Vector Statistical Library(VSL): can be created and initialized in two ways [8].

• Available random number generator as include:
  – for C/C++, VSLNewStream (stream, BRNG, seed)
  – for Fortran, VSLNewStreamEx (stream, BRNG, n, params)

• Integer Output Generation Subroutine. Some properties of testing results of BRNGs and some other information about VSL pseudo-, quasi-, and non-deterministic (true) random number are generated [8].
  – MCG31m1 is a pseudo-random generator, and is described as a 31-bit multiplicative congruential generator.
  – MCG59 is a pseudo-random generator, and is described as a 59-bit multiplicative generator.
  – WH is a pseudo-random generator, and is described as a set of 273 Wichmann-Hill combined multiplicative congruential generators.
  – NIEDERREITER (with Antonov-Saleev modification) is a quasi-random generator, described as an abstract source of random numbers.
  – Sobol is based on a 32-bit Gray code based quasi-random number generator. Paul Bratley and Bennett Fox implemented a Sobol quasi-random sequence generator in 1988 [8]. Within VSL Sobol produces
low-discrepancy sequences with period $2^{32}$. Furthermore, during initialization Sobol allows for user-defined parameters (direction numbers or initial direction numbers and primitive polynomials), and quasi-random number generation can be achieved in any dimension. Quasi random vectors support dimensions from 1 to 40 as a default [8], and are generated according to:

$$x_n = x_{n-1} + v_c$$

$$u_n = x_n / 2^{32}$$

where the value of $c$ is equal to the rightmost zero bit in $n - 1$, and $x_n$ is to the $s$-dimensional vector of 32-bit components [8]. In addition, the $s$-dimensional vectors is calculated during random stream initialization, $v_i, i = 1, 32$

The vector $u_n$ is the generator output normalized to the unit hypercube $(0, 1)^s$ [8].

- NON-DETERMINISTIC (true-random) is a non-deterministic generator and available for the late CPUs such as AVX, which are used in cryptographic applications, gambling and other games of chance [8].

- Single Precision Floating-Point Random Number Vector Generation Subroutine. This method creates a real arithmetic vector that is uniformly distributed in $[a, b]$. The two modes of the VSL random number generation are accurate and fast. The specifying values of the method parameters can be initialized during the call of the distribution generator. For instance, generating accurate single precision floating-point number according to uniform distribution [8] can be done in the C language using:

$$\text{status} = \text{vsRngUniform} (\text{ACCURATE VSL METHOD}, \text{stream}, n, r, a, b);$$
Double Precision Floating-Point Random Number Vector Generation Subroutine. The function produces a real arithmetic vector that is uniformly distributed in \([a, b]\). A double precision floating-point random number sequence in the C language can be generated as follows:[8]

\[
\text{status} = \text{vdRngUniform} (\text{ACCURATE VSL METHOD, stream, n, r, a, b});
\]

### 1.6 Performance Metrics

Execution time and memory usage are the major performance metrics in serial and parallel programming. The execution time justifies a parallel implementation of the Monte Carlo method, since the sequential version is very slow. A Monte Carlo approximation for an integral converges (slowly) as \(\frac{1}{\sqrt{N}}\) where \(N\) is the number of points used. A question that comes to mind is how much faster the application can be run by a parallel version compared to the sequential (serial) version. The speedup ratio is given by \([9][6]\)

\[
\text{Speedup} = \frac{\text{SerialExecutionTime}}{\text{ParallelExecutionTime}}
\]

or

\[
\eta = \frac{T_s}{T_p}
\]

where \(T_s\) is the serial execution time and \(T_p\) is the parallel execution time.

The parallel efficiency\([9][6]\) is another important metric in parallel computing, defined as:

\[
E = \frac{\eta}{P}
\]

where \(\eta\) is the speedup and \(P\) is the number of processors or cores (processes or threads) used in the parallel program. If \(\eta = P\) the speedup is called “ideal” or “linear”; if \(\eta > P\) it is called “superlinear”, Figure\ref{fig:performance} shows the speedup with
number of processes. In addition, Work and Cost have been defined as metrics in parallel computing [6].

Figure 1.8: Speedup
Chapter 2

Monte Carlo Integration

2.1 Monte Carlo Integration

Monte Carlo integration is a powerful method for approximating difficult integrals by random sampling. It derives its name from the city of world’s popular casinos, “Monte Carlo”, because for generating random numbers the roulette wheel was used as the simplest mechanical device [10][11]. Monte Carlo simulation is very useful especially where closed analytical solutions are not available.

The integral of a one-dimensional function $f(x)$ from $a$ to $b$ is written as:

$$\int_{a}^{b} f(x)dx$$

and is represented by the indicated area in Figure 2.1
The Monte approximation for one-dimensional is written as:

$$\int_0^1 f(x)dx \approx \frac{1}{N} \sum_{j=1}^{N} f(x_j)$$

Where the $x_j \in [0, 1]$ are drawn from a uniform distribution.

A multidimensional integral over the d-dimensional unit cube can be written as [12]

$$\mathcal{I} = \int_D f(\bar{x})d\bar{x}$$

where $D = \{(x_1, x_2, ..., x_d) \mid 0 \leq x_i \leq 1, \text{ for } 1 \leq i \leq d\}$

In Figure 2.2 (for $d = 2$) the integral corresponds to the volume contained between the surface of the function $f(x_1, x_2)$ and the (unit square) domain. The multidimensional MC approximation over the d-dimensional unit cube $D$ as:

$$\int_D f(\bar{x})d\bar{x} \approx \frac{1}{N} \sum_{j=1}^{N} f(x_1^j, x_2^j, ..., x_d^j)$$

Figure 2.1: Area under the curve
We developed parallel Monte Carlo integration on the MIC for solving the problems explained later in chapter 3. We implemented two versions for each problem, and the differences between these are explored. As explained in chapter 1, the first version uses offload mode, where the C language is used with some special directives and libraries. In offload mode, the most compute intensive parts run on MIC and some small parts are done by the CPU. The corresponding functions deployed loops or other parts of the program are transferred to the MIC card. The second version for each problem uses native mode, which means that the program is executed completely on the MIC card and it is programmed in the C language with OpenMP. For a native run, the necessary libraries have to transferred to the MIC. The copy is made in advance because, while running on MIC natively, it is not possible to use any libraries on the CPU. Furthermore, for both versions the compilation is done on the CPU.
2.2 VSL Supported Parallel Computation

2.2.1 Splitting Blocks

The most challenging issue in parallel computing is how to balance the load close to perfect by. In CUDA programming for GPUs, a global thread id is the determined by the thread id, block id, and block size. The CUDA runtime performs the underlying data splitting between blocks and threads, so the programmer does not need to know how it works. A kernel is a part of the program that runs on the GPU. Two variables are passed to the kernel between triple angle brackets, for example \( \langle\langle B, 10 \rangle\rangle \) indicates that B blocks are used with 10 threads per block [13].

Splitting data into blocks and assigning blocks to threads is done manually for the Xeon Phi. One of the powerful and suitable technique for Monte Carlo simulation is block-splitting. This method is also known as skipping-ahead and involves splitting the sequence into B non-overlapping blocks [8]. The block-splitting method needs the sequence to be very long and the length needs to be known in advance. This requires some more computation in the definition part of the program. The code in Figure 2.3 represents block-splitting in our Monte Carlo simulation:

```c
vslStreamStatePtr stream;
vslNewStream(&stream, VSL_BRNG_MCG59, RANDSEED);
for(i=0; i<num_of_threads; i++){
    int nskip = n_per_thread * i;
    int status = vslSkipAheadStream( stream, nskip );
    for( int j=0; j < num_of_blocks; j++ )
    {
        vsRngUniform( VSL_RNG_METHOD_UNIFORM_STD, 
                      stream, (dimension*(block_size)), (float*)random, 0.0f, 1.0f);
    }
}
```

Figure 2.3: Block splitting
Figure 2.4 shows the code that we use splitting a sequence of length $N$ into blocks and blocks into threads. For example, for $N = 2048$, $num\_of\_threads = 4$ and $block\_size = 128$. The result is as follow: the number of random numbers per thread $= n\_per\_thread = 512$, $n\_tail = 0$, $num\_of\_blocks = 16$, and $block\_tail = 0$. In this case we do not have a reminder $n\_tail$ or $block\_tail$, but in general the reminders may be no-zeros. Thus by using the skip-ahead technique we can assign each thread 4 different non-overlapping blocks. Figure 2.5 depicts the how block splitting.

```c
int num\_of\_blocks, block\_tail;
n\_pthread = N/num\_of\_threads;
n\_tail = N-n\_per\_thread * num\_of\_threads;

num\_of\_blocks = n\_per\_thread/block\_size;
b\_tail = n\_per\_thread - num\_of\_blocks * block\_size;
```

**Figure 2.4:** Splitting number of points ($N$)

using the code in Figure 2.3 for this example, thread 0 has $i = 0$, $nskip = 0$; thread 1 has $i = 1$, $nskip = 512$. Thus thread 0 processes the first four blocks. Since thread 2 has $i = 2$, $nskip = 1024$, thread 1 processes the second set of 4 blocks, etc.

![Block splitting method](image)

**Figure 2.5:** Block splitting method
2.2.2 Leapfrog

The leapfrog method [14][15] splits the sequence into a number $P$ of disjoint subsequences. Various $P$ can be used and must be chosen carefully according to the data at hand. For example in Figure 2.6 (for $P = 3$) the first, second, and third elements of the first subsequence are shown. The difference between the leapfrog and the skip-ahead method is that the leapfrog method splits the original sequence $x_1, x_2, x_3, \ldots$ into $P$ subsequences so that the first subsequence corresponds to $x_1, x_{P+1}, x_{3P+1}, \ldots$ the second corresponds to $x_2, x_{P+2}, x_{2P+2}, x_{3P+2}, \ldots$ and, finally, the $P^{th}$ to $x_P, x_{2P}, x_{3P}, \ldots$. The leapfrog function in VSL is called as follows:

\[
\text{vslLeapfrogStream}(\text{stream}, \text{thread\_num}, \text{num\_of\_threads})
\]

![Figure 2.6: Leapfrog example, the number of subsequences $P = 3$.]

2.2.3 Using Different Parameter Sets (BRNG sets)

Different parameters can be used for any stream, leading to independent random number sequences. For example, the generator MT2203 can produce 6024 independent basic random number generator (BRNGs) [14][15]. Figure 2.7 shows an example for MT2203.
2.3 Kahan Compensation/Summation

Floating point arithmetic is available in almost every programming language. The Kahan compensation or summation algorithm [16][14][17][18][12] was suggested back in 1965 by Kahan and later analyzed by Kahan and Knuth. It is a very effective and simple method to minimize the loss of accuracy from the accumulation of roundoff error in a summation. When double precision is used, the result is close to the floating point precision result obtained with Kahan summation.

As in [12], first consider $S_N$, the exact value of the sum in

$$S_N = \sum_{i=1}^{N} f_i,$$

and let $\hat{S}_N$ be the computed value of the sum; then $E_N$ is the (absolute) error, where

$$E_N = S_N - \hat{S}_N.$$

The interest here is in the relative error $E_N/S_N$, that is undefined when $S_N = 0$. When $S_N = 0$, a bound on the absolute error can still be sought [12]. A bound on the relative error indicates how many digits of precision can be lost due to roundoff. Under reasonable assumptions on the integrand, if the integral value is $I$, the expected value of $S_N$ is $NI$. So if the integral is not 0, the value of $|S_N|$ is...
```c
float sum = 0.0;
int i;
for(i = 0; i < N; i++) {
    sum += f[i];
}
return sum;
```

**Figure 2.8:** Naive recursive summation

expected to be non-zero and increase up with $N$. In what follows it is assumed that $S_N \neq 0$. One way to accumulate the sum is naive summation as shown in Figure 2.8.

This is a member of a class of summation algorithms called recursive summation, where the sum is initialized and accumulated. As another example of a recursive summation method, the data could be sorted and then summed. It is clear that naive summation can be performed as an online method, but sorting first cannot.

Letting $e$ be the accuracy of the naive recursive summation and assuming $N \ast e < 1$, Equation 2.1 below is a bound on the relative error in $\hat{S}_N$ for naive summation,

\[
\frac{|E_N|}{|S_N|} \leq ((N - 1)e + \mathcal{O}(e^2)) \frac{\sum_{i=1}^{N} |f_i|}{|\sum_{i=1}^{N} f_i|},
\]

(2.1)

In 2.1 the factor

\[
\kappa_s = \frac{\sum_{i=1}^{N} |f_i|}{|\sum_{i=1}^{N} f_i|}
\]

(2.2)

represents a condition number for the summation process. It is known that any summation method that uses a fixed precision and takes time independent of the data (it can be dependent on $N$) will have a factor $\kappa_s$ in its condition number.

A condition number of a process is a measure of the sensitivity of the result to the input. Note that $\kappa_s = 1$ if all the $f_i$ are positive; thus the summation is well conditioned. $\kappa_s$ can be very large if there is significant cancellation in the summation [12].
For positive $f_i$ the naive summation using doubles for $N = 10^{14}$ results in a relative error bound of $10^{-2}$ or 2 digits of precision. If $\kappa_x = 10^5$ there might be serious problems. Note though that Equation (2.2) gives the worst case situation in terms of the accumulation of the rounding errors.
Again assuming \( N \times e < 1 \), the relative error bound for Kahan summation \([12]\) is given by (2.3)

\[
\frac{|E_N|}{|S_N|} \leq (2e + O(Ne^2)) \frac{\sum_{i=1}^{n} |f_i|}{\sum_{i=1}^{n} f_i}
\]  

(2.3)

```c
float y, temp, c, sum;
int j;
for (j=1; j<N; j++) {
    y = X[j] - c;
    temp = sum + y;
    c = (temp - sum) - y;
    sum = temp;
}
return sum;
```

**Figure 2.9:** Kahan summation
Chapter 3

Applications

3.1 Monte Carlo Integration

We developed Monte Carlo simulation for solving some problems covered later in this chapter, and we have implemented two versions for each problem. The first version is the offload mode discussed in section 1.2.1, where the C language is used with some special directives, and libraries needed on the MIC. The second version corresponds to the native mode, which means that the program on the MIC card and (implemented in the C language with OpenMP) is executed completely on the MIC card. For a native run, we have to transfer dependent libraries to the MIC. The copy is made in advance because while running on MIC natively it is not possible to use any libraries on the CPU. Furthermore, for both versions the compilation is performed on the CPU.

3.2 High Energy Physics

We have developed Monte Carlo simulations using the offload and native modes on MIC [7], for the Feynman integral problem that is used in high energy physics for
calculating the crossed and ladder section of particle interactions. 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 of 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 $\delta \to 0$. In [19] a numerical procedure is proposed for convergence acceleration or extrapolation of a sequence of loop integral values as $\delta$ decreases.

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

$$I = \frac{\Gamma \left(N - \frac{nL}{2}\right)}{(4\pi)^{nL/2}} (-1)^N \int_0^1 \prod_{j=1}^N dx_j \delta(1 - \sum x_j) \frac{C^{N-n(L+1)/2}}{(D - i\delta C)^{N-nL/2}}, \quad (3.1)$$

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. We assume $I$ does not suffer from other divergences, such as infrared (IR) or ultraviolet (UV) divergence. To calculate the limit numerically [19], a sequence of $I = I(\delta)$ is generated for (geometrically) decreasing values of $\delta$, and (linear or non-linear) convergence acceleration or extrapolation to the limit is applied [20][21][7][22][23][24][25][26].

The integration in (3.1) is taken over the $N$-dimensional unit cube. However, as a result of the $\delta$-function one of the $x_j$ can be expressed in terms of the other ones, which reduces the integral dimension to $N-1$ and the domain to the $d = (N-1)$-dimensional unit simplex of the form

$$S_d = \{(x_1, x_2, \ldots, x_d) \in \mathbb{R}^d \mid \sum_{j=1}^d x_j \leq 1 \text{ and } x_j \geq 0\} \quad (3.2)$$

In this thesis, we implemented 2-loop crossed and ladder box diagram Feynman integrals, given in (3.3) for $L=2$ and $n=4$:
\[ \mathcal{I} = (-1)^N \prod_{j=1}^{N-1} \int_0^{1-\sum_{k=1}^{j-1}} dx_j \frac{C^{N-6}(D + i\delta)^{N-4}}{(D^2 + \delta^2)^{N-4}}, \]

\[ = (-1)^N \int_0^1 dx_1 \int_0^{1-x_1} dx_2 \cdots \int_0^{1-x_1-\cdots-x_{N-2}} dx_{N-1} \frac{C^{N-6}(D + i\delta)^{N-4}}{(D^2 + \delta^2)^{N-4}}, \] (3.3)

The results for (3.3) are shown in chapter 4, as obtained from our implementation on the Intel Xeon Phi coprocessor with 61 cores. The corresponding 2-loop ladder and 2-loop crossed box diagrams \cite{27} where \( N = 7 \) are shown in Figure 3.1.

Figure 3.1: 2-loop box ladder diagram, 2-loop box crossed diagram

\footnote{F. Yuasa at the High Energy Accelerator Research Organization (KEK), Tsukuba, Japan.}
3.3 Computational Geometry

3.4 Cube-Tetrahedron $E[V_4(C^3)]$

![Figure 3.2: Cube-tetrahedron picking](image)

We developed the offload and native nodes on MIC, to approximate $E[V_4(C^3)]$ by Monte Carlo integration, using C/C++ with OpenMP. We run the application on a node of the thor cluster with Dual Intel Xeon E5-2670, 2.6GHz CPUs and 128GB of memory, and a 61 Core Xeon Phi card (MIC) with 8GB of memory. For small problems we do not get a good speedup, using either offload or native mode. However the speedup increases up to a certain limit with increasing $N$. The speedup is calculated by dividing the sequential time by parallel time. We obtained the sequential time by using the erand48() random number generator on the CPU.
3.5 The Calculation

In stochastic geometry, the cube-tetrahedron picking problem determines the expected volume of a random tetrahedron in the interior of a cube. The vertices of the tetrahedron are chosen independently and uniformly \([28][29][30]\). In general, the computation of \(E[V_n(K)]\) is a classical problem in stochastic computational geometry, for the expected volume of the convex hull generated by \(n\) independent and uniformly distributed points in the interior of a convex body \(K\). Denoting the \(d\)-dimensional volume of \(K\) by \(|K|\), and the volume of the convex hull of \(n\) points by \(\text{conv}(X_1, ..., X_n)\), this is given by the integral in (3.4)

\[
E[V_n(K)] = \frac{1}{|K|} \int_K \cdots \int_K \text{conv}(X_1, \ldots, X_n) \frac{dX_1}{|K|} \cdots \frac{dX_n}{|K|} \quad (3.4)
\]

(see [28][29][30]). For cube-tetrahedron picking, \(K\) is the 3D unit cube \(C_3\), and \(V_4\) is the tetrahedron with vertices \(X_1, X_2, X_3, X_4\) inside \(C_3\). Then \(E[V_4(C_3)]\) is determined by an integral over the 12 dimensional unit cube, \([0, 1]^{12}\),

\[
E[V_4(C_3)] = \prod_{i=1}^{4} \left[ \int_0^1 dx_i \int_0^1 dy_i \int_0^1 dz_i \right] |V_4|,
\]

which is:

\[
E[V_4(C_3)] = \int_0^1 dx_1 \int_0^1 dy_1 \int_0^1 dz_1 \int_0^1 dx_2 \int_0^1 dy_2 \int_0^1 dz_2 \\
\int_0^1 dx_3 \int_0^1 dy_3 \int_0^1 dz_3 \int_0^1 dx_4 \int_0^1 dy_4 \int_0^1 dz_4 |V_4|.
\]

The volume \(|V_4|\) of the tetrahedron is given by the absolute value of the determinant

\[
\begin{vmatrix}
  x_1 & y_1 & z_1 & 1 \\
  x_2 & y_2 & z_2 & 1 \\
  \frac{1}{3!} & x_3 & y_3 & z_3 & 1 \\
  x_4 & y_4 & z_4 & 1 \\
\end{vmatrix}
\]
In view of the absolute value, the integrand has derivative singularities and the integral is hard to compute numerically. The analytic result is known and is equal to:

\[ E[V_4(C^3)] = \frac{3977}{21600} - \frac{\pi^2}{2160} \approx 0.0138428 \]

(as given in [28][29][30]).
Chapter 4

Results and Discussion

4.1 System Architecture

We ran the applications on a node of the thor cluster with Dual Intel Xeon E5-2670, 2.6GHz CPUs and 128GB of memory, and MIC with 61 cores and around 8 GB of memory and where each core has L1 and L2 caches. The computational work should be large enough in order to dominate the overhead, e.g., from communication between the host and the device. This version was developed using OpenMP with C/C++ and offload directives for the offload mode applications. VSL random number generators from the MKL library with various BRNGs can be used. Here we used the pseudo-random number generator MCG59. The sequential versions call erand48() as the random number generator on the CPU. As expected, the results indicate that the time for double precision is larger than for single precision. This is due to double precision being 64 bit (=8 bytes) floating point, whereas single precision is 32 bit (=4 bytes) floating point (according to IEEE 754). Using less memory plays an important role in the parallel programming performance on Xeon Phi cards. Later in this chapter we will discuss and compare all results from the native and offload modes for the applications. The
speedup in our applications for an increasing number of points (N) increases up to a certain point, this brings the concept of granularity, which is the ratio of the time required for a basic communication operation for the time required for a basic parallel computation.

4.2 Feynman Loop Integrals Application

4.2.1 Crossed Diagram Feynman Loop Integral

4.2.1.1 Offload

The first mode is offload to MIC. As mentioned in previous chapters, part of the program runs on the MIC card in offload mode and part runs on the CPU. Naturally the most time-consuming parts of the program should be run on MIC (if possible). Data-parallel programming requires dividing big data into smaller chunks and splitting the chunks among the cores in order to gain speedup. The results for the 6-dimensional Feynman loop integral (corresponding to the 2-loop box crossed diagram of Figure 3.1) are shown in Table 4.1, where the absolute error, sequential time ($T_s$), parallel time ($T_p$), and the speedup (with respect to $T_p(SP)$) are listed as the number of points N increases (see Figure 4.1). The speedup increases for large N up to a certain limit (beyond 70), after which it levels off (due to parallel overhead). VSL random number generators from the MKL library with various BRNGs (depending on different parameters) can be used. Here we use the pseudo-random number generator MCG59. In Table 4.1 we compare the parallel performance to that of the sequential version (using erand48()). Figure 4.2 plots the serial ($T_s$) and parallel time ($T_p(SP)$) of the offload results for the crossed diagram Feynman loop integrals with respect to N. As N increases the accuracy increases up to a certain point, where roundoff error sets in. In addition, Figure 4.3 depicts the parallel single ($T_p(SP)$) and double precision ($T_p(DP)$) time of the
offload results for the Feynman loop integral with respect to N. The time for single precision is smaller than that for double precision as discussed in section 4.1.

### Table 4.1: Offload times and speedup results for crossed diagram Feynman loop integral

<table>
<thead>
<tr>
<th>N</th>
<th>Result</th>
<th>[Abs., Error(\text{SP})]</th>
<th>Serial</th>
<th>Tp(\text{SP})</th>
<th>speedup(\text{SP})</th>
<th>Tp(\text{DP})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10^4)</td>
<td>7.153306E-2</td>
<td>1.381834E-2</td>
<td>3.4E-1</td>
<td>2.80E+002</td>
<td>1.2E-3</td>
<td>3.24E+002</td>
</tr>
<tr>
<td>(5 \times 10^3)</td>
<td>7.941855E-2</td>
<td>5.932848E-3</td>
<td>1.7E+0</td>
<td>2.81E+002</td>
<td>6.0E-3</td>
<td>3.29E+002</td>
</tr>
<tr>
<td>(10^4)</td>
<td>7.318768E-2</td>
<td>1.216372E-2</td>
<td>3.4E+0</td>
<td>2.34E+002</td>
<td>1.5E-2</td>
<td>3.59E+002</td>
</tr>
<tr>
<td>(5 \times 10^4)</td>
<td>8.272277E-2</td>
<td>2.628628E-3</td>
<td>1.7E+1</td>
<td>2.38E+002</td>
<td>7.1E-2</td>
<td>3.59E+002</td>
</tr>
<tr>
<td>(10^5)</td>
<td>8.404877E-2</td>
<td>1.302628E-3</td>
<td>3.3E+1</td>
<td>2.29E+002</td>
<td>1.4E-1</td>
<td>3.60E+002</td>
</tr>
<tr>
<td>(5 \times 10^5)</td>
<td>8.424752E-2</td>
<td>1.103878E-3</td>
<td>1.6E+2</td>
<td>2.47E+002</td>
<td>6.5E-1</td>
<td>3.59E+002</td>
</tr>
<tr>
<td>(10^6)</td>
<td>8.433348E-2</td>
<td>1.017918E-3</td>
<td>3.3E+2</td>
<td>2.59E+002</td>
<td>1.3E+0</td>
<td>3.85E+002</td>
</tr>
<tr>
<td>(5 \times 10^6)</td>
<td>8.414121E-2</td>
<td>1.210188E-3</td>
<td>1.6E+3</td>
<td>3.54E+002</td>
<td>4.5E+0</td>
<td>3.95E+002</td>
</tr>
<tr>
<td>(10^7)</td>
<td>8.503879E-2</td>
<td>3.126081E-4</td>
<td>3.3E+3</td>
<td>3.88E+002</td>
<td>8.5E+0</td>
<td>4.99E+002</td>
</tr>
<tr>
<td>(5 \times 10^7)</td>
<td>8.519184E-2</td>
<td>1.595581E-4</td>
<td>1.6E+4</td>
<td>5.73E+002</td>
<td>2.8E+1</td>
<td>6.48E+002</td>
</tr>
<tr>
<td>(10^8)</td>
<td>8.533403E-2</td>
<td>1.736815E-5</td>
<td>3.3E+4</td>
<td>7.25E+002</td>
<td>4.6E+1</td>
<td>8.94E+002</td>
</tr>
<tr>
<td>(2 \times 10^8)</td>
<td>8.534404E-2</td>
<td>7.381505E-6</td>
<td>6.5E+4</td>
<td>9.52E+002</td>
<td>6.8E+1</td>
<td>1.19E+003</td>
</tr>
<tr>
<td>(3 \times 10^8)</td>
<td>8.534327E-2</td>
<td>8.128150E-6</td>
<td>9.2E+4</td>
<td>1.24E+003</td>
<td>7.4E+1</td>
<td>1.84E+003</td>
</tr>
</tbody>
</table>

**Figure 4.1:** The speedup of the offload results for the 2-loop crossed diagram Feynman loop integral as a function of the number of points N
4.2.1.2 Native

The second mode is native, which uses only the Xeon Phi coprocessor to execute the program (see chapter 1). We use native mode with OpenmMP in the C language to parallelize the Feynman loop integrals. In section 4.1, we discussed...
the MIC system specifications and random number generator. The results for the 6-dimensional Feynman loop integral (corresponding to the crossed diagram of Figure 3.1) are shown in Table 4.2, where the error, sequential time \( (T_s) \), single \( (T_p(\text{SP})) \) and double \( (T_p(\text{DP})) \) precision parallel time, and the speedup are listed for increasing \( N \).

Figure 4.4 displays the speedup, which increases for large number of points \( N \), as discussed in section 4.1. The serial and parallel time of the native results for the Feynman loop integral as a function of \( N \) are depicted in Figure 4.5; the results are similar to the offload version. The serial time increases considerably and the parallel time increases only slightly. Figure 4.6 shows the parallel single \( (T_p(\text{SP})) \) and double precision \( (T_p(\text{DP})) \) time of the native results for the Feynman loop with respect to \( N \). Single precision consumes less time than double precision (see also section 4.1).

### Table 4.2: Native times and speedup results for 2-loop Feynman loop integral (crossed box diagram)

<table>
<thead>
<tr>
<th>( N )</th>
<th>Result</th>
<th>( \text{Abs.Error(\text{SP})} )</th>
<th>Serial Time</th>
<th>( T_p(\text{SP}) )</th>
<th>speedup</th>
<th>( T_p(\text{DP}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^4 )</td>
<td>7.450869E-2</td>
<td>1.084271E-2</td>
<td>340.0E-3</td>
<td>145.0E-1</td>
<td>234.5E-4</td>
<td>218.9E-1</td>
</tr>
<tr>
<td>( 5 \times 10^3 )</td>
<td>7.928818E-2</td>
<td>6.063218E-3</td>
<td>170.0E-2</td>
<td>103.9E+0</td>
<td>163.7E-4</td>
<td>238.4E+0</td>
</tr>
<tr>
<td>( 10^4 )</td>
<td>8.081384E-2</td>
<td>4.537558E-3</td>
<td>340.0E-2</td>
<td>136.9E+0</td>
<td>248.4E-4</td>
<td>243.9E+0</td>
</tr>
<tr>
<td>( 5 \times 10^4 )</td>
<td>8.531835E-2</td>
<td>3.304815E-5</td>
<td>170.0E-1</td>
<td>204.8E+0</td>
<td>829.9E-4</td>
<td>273.4E+0</td>
</tr>
<tr>
<td>( 10^5 )</td>
<td>8.409959E-2</td>
<td>1.251808E-3</td>
<td>330.0E-1</td>
<td>208.9E+0</td>
<td>157.9E-3</td>
<td>289.4E+0</td>
</tr>
<tr>
<td>( 5 \times 10^5 )</td>
<td>8.550503E-2</td>
<td>-1.536319E-4</td>
<td>160.0E+0</td>
<td>197.6E+0</td>
<td>809.8E-3</td>
<td>278.4E+0</td>
</tr>
<tr>
<td>( 10^6 )</td>
<td>8.587205E-2</td>
<td>-5.206519E-4</td>
<td>330.0E+0</td>
<td>237.5E+0</td>
<td>139.0E-2</td>
<td>290.3E+0</td>
</tr>
<tr>
<td>( 5 \times 10^6 )</td>
<td>8.518797E-2</td>
<td>1.634281E-4</td>
<td>160.0E+1</td>
<td>266.0E+0</td>
<td>601.4E-2</td>
<td>293.4E+0</td>
</tr>
<tr>
<td>( 10^7 )</td>
<td>8.546886E-2</td>
<td>-1.174619E-4</td>
<td>330.0E+1</td>
<td>262.5E+0</td>
<td>125.7E-1</td>
<td>383.5E+0</td>
</tr>
<tr>
<td>( 5 \times 10^7 )</td>
<td>8.548287E-2</td>
<td>-1.314719E-4</td>
<td>160.0E+2</td>
<td>507.2E+0</td>
<td>315.5E-1</td>
<td>601.5E+0</td>
</tr>
<tr>
<td>( 10^8 )</td>
<td>8.542618E-2</td>
<td>-7.478185E-5</td>
<td>330.0E+2</td>
<td>552.4E+0</td>
<td>597.4E-1</td>
<td>685.0E+0</td>
</tr>
<tr>
<td>( 2 \times 10^8 )</td>
<td>8.538732E-2</td>
<td>-3.592185E-5</td>
<td>650.0E+2</td>
<td>710.5E+0</td>
<td>914.9E-1</td>
<td>844.0E+0</td>
</tr>
<tr>
<td>( 3 \times 10^8 )</td>
<td>8.537982E-2</td>
<td>-2.842185E-5</td>
<td>920.0E+2</td>
<td>817.6E+0</td>
<td>112.5E+0</td>
<td>102.4E+1</td>
</tr>
</tbody>
</table>
Figure 4.4: Native speedup results for the 2-loop crossed diagram Feynman loop integral as a function of the number of points N

Figure 4.5: Serial and native parallel time for the 2-loop crossed diagram Feynman loop integral as a function of the number of points N
4.2.1.3 Comparison

Comparisons between offload and native mode speedup and time are shown in Table 4.3 for the 2-loop box crossed diagram. The native mode version for the 2-loop crossed diagram Feynman loop integral spends less time than the offload mode version. The peak speedup in native mode is $1.13 \times 10^2$, and in offload it is $7.45 \times 10^1$, where number of points ($N$) is $3 \times 10^8$ (see Table 4.3 and Figure 4.7). The native and offload modes yield very close accuracy and show a similar behaviour for double and single precision (see section 4.1).

The times taken in offload and native mode are depicted in Figure 4.8; offload time is larger than native time and the offload mode spends more time for large $N$, because of the overhead in transferring data to the Xeon Phi card.
Table 4.3: Times and speedup results for the 2-loop crossed diagram Feynman loop function integral

<table>
<thead>
<tr>
<th>N</th>
<th>Offload Speedup</th>
<th>Native Speedup</th>
<th>Offload Time</th>
<th>Native Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>1.21E-03</td>
<td>2.34E-02</td>
<td>2.80E+002</td>
<td>1.45E+001</td>
</tr>
<tr>
<td>$5 \times 10^3$</td>
<td>6.04E-03</td>
<td>1.64E-02</td>
<td>2.81E+002</td>
<td>1.04E+002</td>
</tr>
<tr>
<td>$10^4$</td>
<td>1.45E-02</td>
<td>2.48E-02</td>
<td>2.34E+002</td>
<td>1.37E+002</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>7.13E-02</td>
<td>8.30E-02</td>
<td>2.38E+002</td>
<td>2.05E+002</td>
</tr>
<tr>
<td>$10^5$</td>
<td>1.44E-01</td>
<td>1.58E-01</td>
<td>2.29E+002</td>
<td>2.09E+002</td>
</tr>
<tr>
<td>$5 \times 10^5$</td>
<td>6.49E-01</td>
<td>8.10E-01</td>
<td>2.47E+002</td>
<td>1.98E+002</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.28E+00</td>
<td>1.39E+00</td>
<td>2.59E+002</td>
<td>2.37E+002</td>
</tr>
<tr>
<td>$5 \times 10^6$</td>
<td>4.52E+00</td>
<td>6.01E+00</td>
<td>3.54E+002</td>
<td>2.66E+002</td>
</tr>
<tr>
<td>$10^7$</td>
<td>8.51E+00</td>
<td>1.26E+01</td>
<td>3.88E+002</td>
<td>2.62E+002</td>
</tr>
<tr>
<td>$5 \times 10^7$</td>
<td>2.79E+01</td>
<td>3.15E+01</td>
<td>5.73E+002</td>
<td>5.07E+002</td>
</tr>
<tr>
<td>$10^8$</td>
<td>4.55E+01</td>
<td>5.97E+01</td>
<td>7.25E+002</td>
<td>5.52E+002</td>
</tr>
<tr>
<td>$2 \times 10^8$</td>
<td>6.83E+01</td>
<td>9.15E+01</td>
<td>9.52E+002</td>
<td>7.10E+002</td>
</tr>
<tr>
<td>$3 \times 10^8$</td>
<td>7.45E+01</td>
<td>1.13E+02</td>
<td>1.24E+003</td>
<td>8.18E+002</td>
</tr>
</tbody>
</table>

Figure 4.7: Comparison between the speedup of native and offload versions for the 2-loop crossed Feynman loop integral as a function of the number of points N
4.2.2 Ladder Diagram Feynman Loop Integral

4.2.2.1 Offload

The results of the ladder diagram integrals in offload mode are shown in Table 4.4, where the error, sequential time ($T_s$), parallel time ($T_p$) and the speedup (for single precision are listed with respect to $N$. The specification of the hardware and the random number generators for the applications are discussed in section 4.1. Figure 4.9 shows the speedup for an increasing number of points ($N$), which increases up to a certain point (where it levels off).

Figure 4.10 shows the serial and parallel time of the offload results for the ladder diagram integral with respect to the number of points ($N$). The serial time increases considerably for an increasing number of points ($N$); however the parallel time increases only slightly. Furthermore, the parallel single precision ($T_p(SP)$) and double precision ($T_p(DP)$) time of the offload results for the ladder diagram integrals are plotted with respect to the number of points ($N$) (in Figure 4.11).
The double precision time is larger because of the word size of double precision, which is 8 bytes while single precision is 4 bytes (see also section 4.1).

**Table 4.4:** Offload times and speedup results for the ladder diagram integral

<table>
<thead>
<tr>
<th>N</th>
<th>Result</th>
<th>Error</th>
<th>Serial Time</th>
<th>Tp(SP)</th>
<th>Speedup</th>
<th>Tp(DP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10⁴</td>
<td>1.028652E-1</td>
<td>7.755210E-4</td>
<td>3.380E-1</td>
<td>2.812E+2</td>
<td>1.202E-3</td>
<td>2.893E+2</td>
</tr>
<tr>
<td>5 × 10³</td>
<td>1.028712E-1</td>
<td>7.694910E-4</td>
<td>1.683E+0</td>
<td>2.980E+2</td>
<td>5.648E-3</td>
<td>3.099E+2</td>
</tr>
<tr>
<td>10⁴</td>
<td>1.029510E-1</td>
<td>6.897210E-4</td>
<td>3.466E+0</td>
<td>3.780E+2</td>
<td>9.170E-3</td>
<td>3.981E+2</td>
</tr>
<tr>
<td>10⁵</td>
<td>1.030674E-1</td>
<td>5.732910E-4</td>
<td>6.445E+1</td>
<td>3.607E+2</td>
<td>1.787E-1</td>
<td>3.494E+2</td>
</tr>
<tr>
<td>10⁶</td>
<td>1.035787E-1</td>
<td>6.202099E-5</td>
<td>3.337E+2</td>
<td>3.209E+2</td>
<td>1.040E+0</td>
<td>4.039E+2</td>
</tr>
<tr>
<td>5 × 10⁶</td>
<td>1.035670E-1</td>
<td>7.372099E-5</td>
<td>1.673E+3</td>
<td>5.201E+2</td>
<td>3.217E+0</td>
<td>5.839E+2</td>
</tr>
<tr>
<td>10⁷</td>
<td>1.036095E-1</td>
<td>3.117449E-5</td>
<td>3.402E+3</td>
<td>4.192E+2</td>
<td>8.115E+0</td>
<td>5.429E+2</td>
</tr>
<tr>
<td>5 × 10⁷</td>
<td>1.036420E-1</td>
<td>-1.279011E-6</td>
<td>1.667E+4</td>
<td>7.030E+2</td>
<td>2.371E+1</td>
<td>7.928E+2</td>
</tr>
<tr>
<td>10⁸</td>
<td>1.036430E-1</td>
<td>-2.279011E-6</td>
<td>3.330E+4</td>
<td>7.700E+2</td>
<td>4.325E+1</td>
<td>8.433E+2</td>
</tr>
<tr>
<td>2 × 10⁸</td>
<td>1.036559E-1</td>
<td>-1.517901E-5</td>
<td>6.709E+4</td>
<td>9.682E+2</td>
<td>6.930E+1</td>
<td>1.022E+3</td>
</tr>
<tr>
<td>3 × 10⁸</td>
<td>1.035637E-1</td>
<td>7.702099E-5</td>
<td>9.989E+4</td>
<td>1.227E+3</td>
<td>8.141E+1</td>
<td>1.562E+3</td>
</tr>
</tbody>
</table>

**Figure 4.9:** Speedup of the offload results for the 2-loop ladder diagram integral as a function of the number of points N
4.2.2.2 Native

Table 4.5 lists the error, sequential time, parallel time, and the speedup with respect to the number of points (N). The application is run on the Xeon Phi card.
independently, so the data does not need to be transferred to the MIC. Figure 4.12 displays the speedup as a function of the number of points (N). The speedup results of the native version show a familiar pattern where the speedup increases up to a certain point as the number of points (N) increases.

Figure 4.13 depicts the serial and parallel time of the native results for the ladder diagram integrals with respect to N. The serial time ($T_s$) grows considerably, but the parallel time grows only slightly for N. In addition, the parallel time of the single precision version is smaller than for double precision (see also section 4.1).

**Table 4.5: Native times and speedup results for the ladder diagram integral**

| N    | Result         | $|Abs. Error|$ | Serial Time | $Tp(SP)$ | Speedup  | $Tp(DP)$ |
|------|----------------|----------------|-------------|----------|----------|----------|
| 5 $\times$ 10$^3$ | 1.029499E-1    | 6.908210E-4    | 1.683E+0    | 3.235E+2 | 5.203E-3 | 3.289E+2 |
| 10$^4$ | 1.029430E-1    | 6.977210E-4    | 3.466E+0    | 3.752E+2 | 9.239E-3 | 3.873E+2 |
| 5 $\times$ 10$^4$ | 1.029811E-1    | 6.595988E-4    | 2.367E+1    | 3.954E+2 | 5.987E-2 | 3.989E+2 |
| 10$^5$ | 1.035002E-1    | 1.405210E-4    | 6.445E+1    | 3.726E+2 | 1.730E-1 | 3.895E+2 |
| 5 $\times$ 10$^5$ | 1.036002E-1    | 4.052099E-5    | 1.669E+2    | 3.631E+2 | 4.597E-1 | 3.883E+2 |
| 10$^6$ | 1.036789E-1    | -3.817901E-5   | 3.337E+2    | 3.310E+2 | 1.008E+0 | 3.894E+2 |
| 5 $\times$ 10$^6$ | 1.036857E-1    | -4.497901E-5   | 3.466E+3    | 4.501E+2 | 3.717E+0 | 5.309E+2 |
| 10$^7$ | 1.035737E-1    | 6.502099E-5    | 3.402E+3    | 4.761E+2 | 7.146E+0 | 6.192E+2 |
| 5 $\times$ 10$^7$ | 1.035690E-1    | 7.172099E-5    | 6.030E+2    | 2.764E+1 | 8.103E+2 |
| 2 $\times$ 10$^8$ | 1.036445E-1    | -3.779011E-6   | 6.709E+4    | 8.591E+2 | 7.809E+1 | 9.450E+2 |
| 3 $\times$ 10$^8$ | 1.036299E-1    | 1.082099E-5    | 9.989E+4    | 1.036E+3 | 9.643E+1 | 1.178E+3 |
Figure 4.12: Native speedup results for the 2-loop ladder diagram integral as a function of the number of points $N$.

Figure 4.13: Native serial and parallel time for the 2-loop ladder diagram integral as a function of the number of points $N$. 

45
4.2.2.3 Comparison

The results in Table 4.6 show comparisons between the offload and native mode in terms of the speedup and time. Figure 4.7 shows that the speedup gained in native mode for the ladder diagram loop integral is larger than in offload. The peak speedup in offload is $8.14E+01$, while in native mode it is $9.64E+01$, where the number of points ($N$) is $3 \times 10^8$. The native version has better speedup as it does not copy data from the host to the device and vice-versa. Another aspect that can be discussed is that the parallel efficiency depends on the nature of the application. For instance, if we have a longer sequential part in the program, the native version may be slower than offload. The cores on the Xeon Phi card are very simple and slow, thus running sequential by on the card would be significantly slower than running on the CPU. The differences between native and offload mode parallel time and speedup are shown in Figure 4.16. It emerges that offload version is slower than native mode especially for large $N$. 

![Figure 4.14: Single and double precision parallel time for the 2-loop ladder diagram integral as a function of the number of points $N$](image)
### Table 4.6: Times and speedup results for the ladder diagram integral

<table>
<thead>
<tr>
<th>N</th>
<th>Native Time</th>
<th>Offload Time</th>
<th>Native Speedup</th>
<th>Offload Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>2.79237E+002</td>
<td>2.81232E+002</td>
<td>1.21E-03</td>
<td>1.20E-03</td>
</tr>
<tr>
<td>$5 \times 10^3$</td>
<td>3.23475E+002</td>
<td>2.97980E+002</td>
<td>5.20E-03</td>
<td>5.65E-03</td>
</tr>
<tr>
<td>$10^4$</td>
<td>3.75168E+002</td>
<td>3.77981E+002</td>
<td>9.24E-03</td>
<td>9.17E-03</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>3.95355E+002</td>
<td>4.45355E+002</td>
<td>5.99E-02</td>
<td>5.31E-02</td>
</tr>
<tr>
<td>$10^5$</td>
<td>3.72571E+002</td>
<td>3.60653E+002</td>
<td>1.73E-01</td>
<td>1.79E-01</td>
</tr>
<tr>
<td>$5 \times 10^5$</td>
<td>3.63097E+002</td>
<td>3.39089E+002</td>
<td>4.60E-01</td>
<td>4.92E-01</td>
</tr>
<tr>
<td>$10^6$</td>
<td>3.31001E+002</td>
<td>3.20908E+002</td>
<td>1.01E+00</td>
<td>1.04E+00</td>
</tr>
<tr>
<td>$5 \times 10^6$</td>
<td>4.50105E+002</td>
<td>5.20105E+002</td>
<td>3.72E+00</td>
<td>3.22E+00</td>
</tr>
<tr>
<td>$10^7$</td>
<td>4.76104E+002</td>
<td>4.19219E+002</td>
<td>7.15E+00</td>
<td>8.12E+00</td>
</tr>
<tr>
<td>$5 \times 10^7$</td>
<td>6.03021E+002</td>
<td>7.03021E+002</td>
<td>2.76E+01</td>
<td>2.37E+01</td>
</tr>
<tr>
<td>$10^8$</td>
<td>7.03021E+002</td>
<td>7.70021E+002</td>
<td>4.74E+01</td>
<td>4.32E+01</td>
</tr>
<tr>
<td>$2 \times 10^8$</td>
<td>8.59127E+002</td>
<td>9.68177E+002</td>
<td>7.81E+01</td>
<td>6.93E+01</td>
</tr>
<tr>
<td>$3 \times 10^8$</td>
<td>1.03587E+003</td>
<td>1.22700E+003</td>
<td>9.64E+01</td>
<td>8.14E+01</td>
</tr>
</tbody>
</table>

**Figure 4.15:** Comparison between the speedup of native and offload versions for the 2-loop ladder diagram integral as a function of the number of points $N$.
4.3 Cube-Tetrahedron Picking Expected Value

\[ E[V_4(C^3)] \]

4.3.1 Offload

We implemented Monte Carlo integration to solve cube-tetrahedron picking problems in the offload mode on the Intel Xeon Phi card. In section 4.1, we discussed the hardware system, programming environment and the random number generators. Table 4.7 shows the results and performance of the expected value calculation of the cube-tetrahedron picking problem \( E[V_4(C^3)] \) leading to 12-dimensional integrals. Figure 4.17 shows the speedup graph as a function of the number of points (N), using the 61 cores of the Xeon Phi. Offload directives were used to transfer data and functions to the Intel Xeon Phi card.

The sequential and parallel time for the cube-tetrahedron picking expected value \( E[V_4(C^3)] \) is depicted in Figure 4.18. The highest sequential time is 8.777445E+4 for \( 3 \times 10^8 \) number of points (N).
Table 4.7: Times and speedup results of native and offload versions for the cube-tetrahedron picking expected value $E[V_4(C^3)]$

<table>
<thead>
<tr>
<th>N</th>
<th>Result</th>
<th>Abs.Error</th>
<th>Serial Time</th>
<th>Parallel Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^4$</td>
<td>1.352597E-2</td>
<td>3.168058E-4</td>
<td>2.940000E-1</td>
<td>2.79237E+002</td>
<td>1.05E-003</td>
</tr>
<tr>
<td>$5 \times 10^3$</td>
<td>1.348075E-2</td>
<td>3.620258E-4</td>
<td>1.401000E+0</td>
<td>2.93475E+002</td>
<td>4.77E-003</td>
</tr>
<tr>
<td>$10^4$</td>
<td>1.363150E-2</td>
<td>2.112758E-4</td>
<td>2.841799E+2</td>
<td>3.75168E+002</td>
<td>7.57E-001</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>1.365884E-2</td>
<td>1.839358E-4</td>
<td>3.596680E+2</td>
<td>4.45355E+002</td>
<td>8.08E-001</td>
</tr>
<tr>
<td>$10^5$</td>
<td>1.377253E-2</td>
<td>7.024575E-5</td>
<td>3.751681E+2</td>
<td>3.75168E+002</td>
<td>1.00E+000</td>
</tr>
<tr>
<td>$5 \times 10^5$</td>
<td>1.382837E-2</td>
<td>1.440575E-5</td>
<td>1.311230E+2</td>
<td>3.63097E+002</td>
<td>3.61E-001</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.383437E-2</td>
<td>8.405751E-6</td>
<td>2.559350E+2</td>
<td>3.44713E+002</td>
<td>7.42E-001</td>
</tr>
<tr>
<td>$5 \times 10^6$</td>
<td>1.383580E-2</td>
<td>6.975751E-6</td>
<td>1.303061E+3</td>
<td>5.20105E+002</td>
<td>2.51E+000</td>
</tr>
<tr>
<td>$10^7$</td>
<td>1.383633E-2</td>
<td>6.445751E-6</td>
<td>2.602076E+3</td>
<td>5.99219E+002</td>
<td>4.34E+000</td>
</tr>
<tr>
<td>$5 \times 10^7$</td>
<td>1.383788E-2</td>
<td>4.895751E-6</td>
<td>1.302058E+4</td>
<td>1.13002E+003</td>
<td>1.15E+001</td>
</tr>
<tr>
<td>$10^8$</td>
<td>1.384252E-2</td>
<td>2.557507E-7</td>
<td>2.606409E+4</td>
<td>1.47002E+003</td>
<td>1.77E+001</td>
</tr>
<tr>
<td>$2 \times 10^8$</td>
<td>1.384250E-2</td>
<td>2.757507E-7</td>
<td>5.391147E+4</td>
<td>2.08969E+003</td>
<td>2.58E+001</td>
</tr>
<tr>
<td>$3 \times 10^8$</td>
<td>1.384123E-2</td>
<td>1.545751E-6</td>
<td>8.777445E+4</td>
<td>5.52700E+003</td>
<td>1.59E+001</td>
</tr>
</tbody>
</table>

Figure 4.17: Speedup of the offload version for the cube-tetrahedron expected value $E[V_4(C^3)]$
Figure 4.18: The sequential and parallel time results of the offload version for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points $N$.

### 4.3.2 Native

The native mode version for the cube-tetrahedron expected value $E[V_4(C^3)]$ runs on the Intel Xeon Phi card without participating CPU. Table 4.8 illustrates the accuracy and the parallel performance of the cube-tetrahedron computation of the expected value. Figure 4.19 shows the increase of the speedup with the number of points (N). For increasing N, the speedup increases until it reaches a certain point; the peak speedup is gained at $N = 2 \times 10^8$ and decreases around $N = 3 \times 10^8$. Figure 4.20 plots the sequential and parallel time.
Table 4.8: Accuracy, times and the speedup results of the native version for the cube-tetrahedron expected value $E[V_4(C^3)]$

<table>
<thead>
<tr>
<th>N</th>
<th>Result</th>
<th>Error</th>
<th>Serial Time</th>
<th>Parallel Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>1.352597E-2</td>
<td>3.168058E-4</td>
<td>2.940000E-1</td>
<td>1.18E+001</td>
<td>2.49E-002</td>
</tr>
<tr>
<td>$5 \times 10^3$</td>
<td>1.348074E-2</td>
<td>3.620358E-4</td>
<td>1.401000E+0</td>
<td>4.87E+001</td>
<td>2.87E-002</td>
</tr>
<tr>
<td>$10^4$</td>
<td>1.367648E-2</td>
<td>1.662958E-4</td>
<td>2.841799E+2</td>
<td>1.36E+002</td>
<td>2.09E+000</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>1.374489E-2</td>
<td>9.788575E-5</td>
<td>1.404400E+2</td>
<td>3.09E+002</td>
<td>4.54E-001</td>
</tr>
<tr>
<td>$10^5$</td>
<td>1.377130E-2</td>
<td>7.147575E-5</td>
<td>3.751681E+2</td>
<td>2.29E+002</td>
<td>1.64E+000</td>
</tr>
<tr>
<td>$5 \times 10^5$</td>
<td>1.385536E-2</td>
<td>-1.258425E-5</td>
<td>1.311230E+2</td>
<td>2.96E+002</td>
<td>4.43E-001</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.385536E-2</td>
<td>-1.258425E-5</td>
<td>2.559350E+2</td>
<td>2.37E+002</td>
<td>1.08E+000</td>
</tr>
<tr>
<td>$5 \times 10^6$</td>
<td>1.383551E-2</td>
<td>7.265751E-6</td>
<td>1.303061E+3</td>
<td>4.16E+002</td>
<td>3.13E+000</td>
</tr>
<tr>
<td>$10^7$</td>
<td>1.383608E-2</td>
<td>6.695751E-6</td>
<td>2.602076E+3</td>
<td>4.31E+002</td>
<td>6.04E+000</td>
</tr>
<tr>
<td>$5 \times 10^7$</td>
<td>1.383725E-2</td>
<td>5.525751E-6</td>
<td>1.302058E+4</td>
<td>7.92E+002</td>
<td>1.64E+001</td>
</tr>
<tr>
<td>$10^8$</td>
<td>1.384337E-2</td>
<td>-5.942493E-7</td>
<td>2.606409E+4</td>
<td>1.33E+003</td>
<td>1.96E+001</td>
</tr>
<tr>
<td>$2 \times 10^8$</td>
<td>1.384281E-2</td>
<td>-3.424932E-8</td>
<td>5.391147E+4</td>
<td>2.02E+003</td>
<td>2.67E+001</td>
</tr>
<tr>
<td>$3 \times 10^8$</td>
<td>1.384123E-2</td>
<td>1.545751E-6</td>
<td>8.777445E+4</td>
<td>5.11E+003</td>
<td>1.72E+001</td>
</tr>
</tbody>
</table>

Figure 4.19: Speedup results of the native version for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points N
Figure 4.20: Times of the native and offload versions for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points $N$

4.3.3 Comparison

Table 4.9 compares the speedup and time of the offload and native modes. The speedup of the cube-tetrahedron picking problem expected value $E[V_4(C^3)]$ in native mode is very close to the offload speedup. For example, the peak speedup for native and offload is 2.67E+01 and 2.58E+001, respectively, at $N$ around $2 \times 10^8$. The error decreases and it converges until a certain point. The offload and native parallel times are also very close, particularly for large $N$. Figure 4.22 displays the results. Furthermore, by increasing the number of points, the error decreases through $N= 2 \times 10^8$, then increases again at $N= 3 \times 10^8$ (see Table 4.3).
Table 4.9: Time and speedup results for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points $N$

<table>
<thead>
<tr>
<th>$N$</th>
<th>Offload Speedup</th>
<th>Native Speedup</th>
<th>Offload Time</th>
<th>Native Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^3$</td>
<td>1.05E-003</td>
<td>2.49E-02</td>
<td>2.79237E+002</td>
<td>1.18E+001</td>
</tr>
<tr>
<td>$5 \times 10^3$</td>
<td>4.77E-003</td>
<td>2.87E-02</td>
<td>2.93475E+002</td>
<td>4.87E+001</td>
</tr>
<tr>
<td>$10^4$</td>
<td>7.57E-001</td>
<td>2.09E+00</td>
<td>3.75168E+002</td>
<td>1.36E+002</td>
</tr>
<tr>
<td>$5 \times 10^4$</td>
<td>8.08E-001</td>
<td>4.54E-01</td>
<td>4.45355E+002</td>
<td>3.09E+002</td>
</tr>
<tr>
<td>$10^5$</td>
<td>1.00E+00</td>
<td>1.64E+00</td>
<td>3.75168E+002</td>
<td>2.29E+002</td>
</tr>
<tr>
<td>$5 \times 10^5$</td>
<td>3.61E-001</td>
<td>4.43E-01</td>
<td>3.63097E+002</td>
<td>2.96E+002</td>
</tr>
<tr>
<td>$10^6$</td>
<td>7.42E-001</td>
<td>1.08E+00</td>
<td>3.44713E+002</td>
<td>2.37E+002</td>
</tr>
<tr>
<td>$5 \times 10^6$</td>
<td>2.51E+00</td>
<td>3.13E+00</td>
<td>5.20105E+002</td>
<td>4.16E+002</td>
</tr>
<tr>
<td>$10^7$</td>
<td>4.34E+00</td>
<td>6.04E+00</td>
<td>5.99219E+002</td>
<td>4.31E+002</td>
</tr>
<tr>
<td>$5 \times 10^7$</td>
<td>1.15E+00</td>
<td>1.64E+01</td>
<td>1.13002E+003</td>
<td>7.92E+002</td>
</tr>
<tr>
<td>$10^8$</td>
<td>1.77E+00</td>
<td>1.96E+01</td>
<td>1.47002E+003</td>
<td>1.33E+003</td>
</tr>
<tr>
<td>$2 \times 10^8$</td>
<td>2.58E+00</td>
<td>2.67E+01</td>
<td>2.08969E+003</td>
<td>2.02E+003</td>
</tr>
<tr>
<td>$3 \times 10^8$</td>
<td>1.59E+00</td>
<td>1.72E+01</td>
<td>5.52700E+003</td>
<td>5.11E+003</td>
</tr>
</tbody>
</table>

Figure 4.21: Comparison between the speedup of native and offload versions for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points $N$
Figure 4.22: Comparison between the time of the native and offload versions for the cube-tetrahedron expected value $E[V_4(C^3)]$ as a function of the number of points $N$. 
Chapter 5

Conclusion and Future Work

5.1 Thesis Summary

In this thesis, we implemented parallel Monte Carlo integration on the MIC for solving problems in high energy physics and computational geometry. Two versions of the Monte Carlo simulation (native and offload) are implemented. A comparison of the parallel performance is presented for the applications.

Chapter 1 gives a general description of the Intel Xeon Phi coprocessor and explains 1.1 of the offload, native and symmetric executions. It far then includes specifications of the Intel compiler suite for Fortran, C and C++. The Intel compiler is used for compiling in both modes, offload and native, for all applications. OpenMP is used to parallelize the applications and for dividing the work among the threads. A brief history of OpenMP is also provided.

In addition, features of the MKL (Math Kernel Library), of interest to our work are highlighted. chapter 2 introduces the Monte Carlo method. The supporting parallel programming VSL (the Vector Statistical Library) is demonstrated, such as splitting blocks, leapfrog, and using method parameter sets (leading to BRNGs sets). Furthermore, the Kahan summation method helps to obtain more accurate
results by reducing roundoff error.

In chapter 3, the applications to the Feynman loop integrals and cube-tetrahedron picking problem are explained and calculations are presented. Chapter 4 provides tables and graphs with results for each mode of application, while focusing on the differences between sequential and parallel time, the speedup, and single/double precision time. Comparisons are also drawn between the offload and native modes for each application in terms of time, speedup and accuracy, and shown in the tables. Charts are also included to depict how they differ with respect to time. Chapter 5 gives conclusions and points to future work.

5.2 Conclusions

The Intel Xeon Phi is a coprocessor with 61 cores, 8 GB of memory, where each core has the power of the first Intel Pentium (P54C) generation from back in 1995. For our implementations and simulations we used the Thor cluster of the High Performance Computing and Big Data Centre in the College of Engineering and Applied Sciences of WMU. In terms of programming, C and OpenMP are used in native mode, whereas in the offload version C, OpenMP and some special offload directives are used to transfer data and functions between the host and coprocessor.

In this thesis, we demonstrated Monte Carlo integration, for solving physics and mathematical problems. The time and speedup are shown for applications to Feynman loop integrals in high energy physics for up to 6-dimensions.

The time and speedup are shown for the Tetrahedron picking problems in particular the computation of $E[V_n(K)]$ in stochastic computational geometry, which
leads to 12 dimensional integrals. The times and speedup for the native and offload execution modes are discussed.

It is clear that the different modes of programming for the Xeon Phi are important for the applications. The division of work between the host and the Xeon Phi card is critical for the applications. Native mode uses the Intel Xeon Phi card only without the host (CPU). We obtain slightly better results for native mode than the offload. The symmetric mode uses the Message Passing Interface which can deal with the card as a separate node.

For speedup calculations, the sequential version of Monte Carlo is run on the CPU with the function erand48() for random number generation. Furthermore, the applications are designed in both single and double precision where the single precision programs make use of Kahan summation.

5.3 Future Work

5.3.1 Mode of Programming

In terms of the modes of programming, the offload and native modes were implemented in this thesis, whereas the symmetric mode was not because of technical problem as mentioned in chapter 2. The symmetric mode uses MPI (Message Passing Interface), for communications between the CPU and the coprocessor. In our future work, the symmetric mode can be implemented using MPI in three different ways, as illustrated in Figures 5.3-5.1. Furthermore, it is possible to use NVIDIA GPU cards together with Intel Xeon Phi coprocessors in symmetric mode.
Figure 5.1: Symmetric mode using MPI between nodes

Figure 5.2: Symmetric hybrid OpenMP+MPI with offload
Figure 5.3: Symmetric hybrid OpenMP+MPI (native)
5.3.2 Trends in Technology

The next generation of Intel Xeon Phi is Knights Landing, which is more powerful than previous versions as it has cores and a different type of memory (MCDRAM) is used. Furthermore, Knights Landing has an on board chip for motherboard, which can run an operating system like CPU and transforms data between main memory and device memory will be easier.
References


63
