Numerical Accuracy Differences

in CPU and GPGPU Codes

A Thesis Presented

by

Devon Yablonski

The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of
Masters of Science
in
COMPUTER ENGINEERING

Advisor:

Professor Miriam Leeser

Defense Committee:

Professor Dana Brooks
and
Professor Gunar Schirner

ELECTRICAL AND COMPUTER ENGINEERING DEPARTMENT

NORTHEASTERN UNIVERSITY

Boston, Massachusetts
2011

Abstract

This thesis presents an analysis of numerical accuracy issues that are found in many scientific GPU applications due to floating-point computation. Two widely held myths about floating-point on GPUs are that the CPU's answer is more precise than the GPU version and that computations on the GPU are unavoidably different from the same computations on a CPU. We dispel both myths by studying a specific application: Digital Breast Tomosynthesis. We describe situations where the GPGPU provides greater precision in the application. Digital Breast Tomosynthesis (1500 lines of code) was analyzed and differences between the CUDA (GPU) and C++ (CPU) implementations were completely resolved. The techniques developed can aid in debugging other scientific GPU floating-point code. This analysis requires an in-depth understanding of the arithmetic issues that arise in implementing floating-point on CPUs, multi-core architectures, and GPUs. We provide insight into the accuracy, speed and verification of floating-point codes, as GPUs are increasingly used in important scientific and medical applications.

Acknowledgements

This work would not have been possible without the guidance of my advisor, Professor Miriam Leeser. Her efforts have been multi-faceted, serving as my curriculum advisor, directing my two separate research projects and providing great learning opportunities through diverse current topic discussions at our weekly group meetings. I am grateful to have participated in research with her at Northeastern University, as I may never have experienced such enlightenment without this opportunity.

I would like to thank Professor Dana Brooks and Professor Gunar Schirner for serving as my master's thesis defense committee. I would also like to thank my lab-mates Nicholas Moore and James Brock for sharing their knowledge of GPGPU with me throughout my work and Mary Ellen Fuess for editing portions of this thesis. Also deserving gratitude are the members of the group who were responsible for writing the GPU Tomosynthesis code and providing support throughout my research, including Dana Schaa, Fernando Quivira and Professor Kaeli.

Finally, a special thank you to my family, friends and my beloved girlfriend Michelle Pelletier, as my accomplishments throughout my academic career can be directly attributed to their support and encouragement.

This work was supported in part by the National Science Foundation Engineering Research Center's Innovations Program (Award Number EEC-0946463).

Contents

List of Tables

List of Figures

1 INTRODUCTION

General Purpose Computation on Graphics Processing Units (GPGPUs) is increasingly being used for scientific computing and in safety critical applications. Many scientific codes that are too slow in their serial form can be accelerated by exploiting the extremely parallel GPGPU hardware. Some applications are in fields that are unwilling to accept errors, such as medicine, where the accuracy of the results is as important as the speed at which they are produced. This is an area that GPGPUs have struggled with in the past because they lacked adequate floating-point support.

When the results from an accelerated application differ from their slower serial predecessors, such as a CPU, programmers and their application end-users are often concerned that the fast application is not performing the computation accurately or that there are mistakes in the code. There are many sources for these differences, including the fact that floating-point is inherently not associative. However, differences can arise even in applications that follow the same order of instructions in both CPU and GPU code. To accept these differences and be comfortable using the application, the user requires assurance that the code is correct and accurate. The response to this is usually that floating-point computation can only be expected to be correct within a certain error.

However, error bounds analysis may not be sufficient, and often error bounds grow to the extent that the results are not meaningful.

This research shows that many differences can be resolved with a close examination of the code and with an understanding of underlying implementation issues. While these issues arise for all computer architectures, they are more obvious on GPUs for a number of reasons. First, GPUs support for floating-point has varied through different generations of hardware. Second, the extreme parallelism available on GPUs makes effects like non-associativity much more noticeable.

The work presented in this thesis provides insight into the computational differences between CPU and GPGPU codes that relate to floating-point. We show that modern GPUs are capable of providing accurate floating-point results, sometimes more accurate than the same application implemented on a CPU. In many cases it is possible to perform the floating-point operations on a GPGPU in an IEEE compliant way that is accurate or even matches the results of a CPU version of the same program. Our study focuses on NVIDIA devices, which are the most popular choice for scientific computing. We use NVIDIA CUDA as our GPU programming language.

One program that experiences differing results between its GPU and CPU implementations is Digital Breast Tomosynthesis (DBT) and is the case study used in this work. We analyze this program and pinpoint differences between the CPU and GPU implementations. We then describe techniques to resolve the differences and explain their effects on execution time. In the process, we create a GPU DBT code that produces identical results to the CPU application with minimal changes to the code and execution

time. We also discuss sources of differences that are not part of DBT, but arise in other applications. This level of understanding will help others produce more accurate results from their CUDA GPGPU applications and thus increase the confidence in scientific GPGPU codes.

1.1 Contribution

The contributions this thesis:

- A detailed investigation of GPU floating-point behavior that compares a CPU and GPU implementation and successfully modifies that GPU implementation to produce identical results to the CPU. To the best of our knowledge a work of this type has not been previously published.

- A discussion of insights that will allow programmers to be aware of differences in CPU and GPU floating-point code and the performance implications of each.

- A unique method of constructing GPU code along side CPU code in order to compare differences, which helps to debug both codes.

1.2 Thesis Outline

The remainder of this work is organized in five chapters. Chapter 2 includes the background information necessary to properly understand the research presented. Chapter 3 describes the exploration of a parallel implementation of calculating π as well as the DBT implemenation, which highlight several floating-point issues in GPU applications. In Chapter 4, we explain the execution time effects of the floating-point possiblities in the DBT application and how this translates to other GPU applications. Chapter 5 will

discuss four versions of DBT that are optimized for different parameters. Finally, Chapter 6 concludes this thesis and discuss future goals.

2 Background

This section consists of the information necessary to adequately understand the remainder of this thesis. Section 2.1 describes floating-point on modern machines including the basic storage layout and some differences in the IEEE floating-point standard implementation on different hardware. Section 2.2 describes the history of GPGPU and the GPU hardware that was used in this work from architectural and software perspectives. Section 2.3 describes the Digital Breast Tomosynthesis algorithm that is used in our research to compare floating-point performance. Section 2.4 discusses work related to floating-point accuracy studies regarding GPUs. Finally, Section 2.5 describes the setup used throughout the research.

2.1 Floating-Point

Modern computer architectures are binary machines. Storing positive and negative, decimal integer values in binary representation is straight forward but it is more difficult to store fractional values or very large integers (greater than what can be stored in an integer data type). In order to represent and store these values efficiently, the floating-point notation [1] was developed.

2.1.1 Notation

Floating-point notation makes it possible to express very large and very small values in a single format. The notation can most easily be described as a representation of real numbers using scientific notation [2]. There are three parts of the floating-point layout that separately represents each part of the scientific notation. These are the sign, the significand (also called the mantissa) and the exponent. The format can be described using the equation $\pm mantissa\ x\ Base^{exponent}$. The *Base* is the base of the number system in use. Though some computers use or have used base 8 (octal) or 16 (hexadecimal) number systems and floating-point notation exists for each, most computers, at the time of this work, are base 2 (binary) machines. As a result, our code uses a base 2 floating-point notation and refers to the IEEE 754-2008 [3] floating-point standard as it applies to base 2 representations.

An advantage of floating-point notation is that the representation (further explained in Section 2.1.2) acts as a sliding (floating) window of precision that expands the range of numbers that it can express. This characteristic is one that a regular integer data type does not exhibit. When the window is slid toward representing a very large number, the lesser of its significant digits can be sacrificed to allow room for it to be

represented. For instance, one could foresee using less storage for $530 * 10\text{^}9$ (530 billion) by storing 530 and 9 instead of using binary to store 530,230,423 explicitly. Oppositely, the window can slide towards representing small numbers, dropping the leading zeros and storing only the significant digits. Rather than directly mapping a bit to each 0 placeholder in 0.00001, only the significant digit 1 along with a value representing the number of 0s (stored as an exponent) are necessary. This sliding window is interpretable because it is supplemented with standardized rules explaining how to extract each portion of the value stored in scientific notation (the floating-point standard).

A disadvantage of using floating-point is that a value is stored with limited accuracy when a more precise value may be desired. For example, the decimal value 0.1 has no representation in binary floating point as it is a repeating fraction when converted to the base 2 number system. Instead, one of the representable values slightly less than or greater to 0.1 must be used, making the value less accurate than what was intended.

Another loss of precision exists because an N-bit floating-point value can store a much larger base 10 number than an N-bit integer. To do this, floating-point sacrifices some precision in order to capture more significant digits and the exponent of scientific notation. An example would be when a 32 bit integer has bits 29 to 0 set to 1, the value in base 10 is $2^{30} - 1 = 1073741823$. For reasons that will be explained in the following section, the 32 bit floating-point notation uses 23 bits to store the significant digits. This means that the value 1073741823_{10} could at most be stored as $1.0737418\ x\ 10^9$ and the final digits "23" are lost (ignoring normalization; see section 2.1.2 for more on normalized and denormalized numbers).

There are conversion applications available for converting between base 2 and base 10 [4]. A more detailed discussion of floating-point itself can be found in [5].

2.1.2 IEEE 754-2008 Floating-Point Standard

Floating-point notation only describes the three parts of floating-point that represent scientific notation. The size of the storage in bits or words, the sizes of the three parts of the value and interpretation of what is stored in each part are left to the implementation.

Operating on floating-point values requires different instructions that can be implemented in different ways. Floating-point computations are needed in many scientific and financial computations which require reproducible and understandable numerical results regardless of the architecture. It was apparent that some standardization of the format and the operations using it was necessary. In response, the IEEE (Institute of Electrical and Electronics Engineers) developed the IEEE 754 standard, with the most recent version (2008) being IEEE 754-2008 [3]. While complying with the standard partially or completely is at the hardware or software provider's discretion, most implementations that hope to achieve widespread adoption adhere as closely as possible. This helps them ensure that their product is consistent for precision dependent codes [6].

The standard defines single and double word length floating-point notation, providing different amounts of precision. Single precision is a 32 bit format to fill the standard word size for most machines. Double precision, as the name suggests, is double the size at 64 bits, or two words on the common machine.

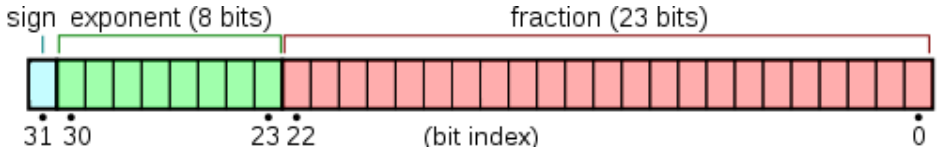

Figure 1: Single precision floating-point format [7]

For single precision, the format begins with a single sign bit, followed by 8 bits for the exponent and 23 bit to store the fractional part as shown in Figure 1. The sign is stored as the MSB (most significant bit); the bit is zero if the value is positive and the bit is 1 if it is negative. The middle portion of the data is the exponent field. It is 8 bits long and represents the exponent in scientific notation. The 8 bits hold a range of integers from 0-255, with a biasing that allows for easier comparison operations, for representing negative and positive exponents. Subtracting 127 from the integer represented in the 8 bit exponent reveals the true base 10 exponent value (likewise, 127 must be added to the exponent value you wish to store in the floating-point format). For instance, a stored exponent of $0x00_2$ represents the real exponent of -127 and the greatest exponent representable in the 8 bits is 127. This allows for very small real numbers to be stored (negative exponents) or very large (positive exponents). The significand (mantissa) field is the lower 23 bits (from 22 down to 0). This is where the significant digits of the scientific notation, to be scaled by the $base^{exponent}$, are stored.

Double precision (Figure 2) allows for much greater precision and an even greater range of values. This extended precision works nearly the same. The sign bit remains a single bit, the exponent is expanded to 11 bits, using a bias of 1023 instead of 127 as in single precision, and 52 bits are used for the significand.

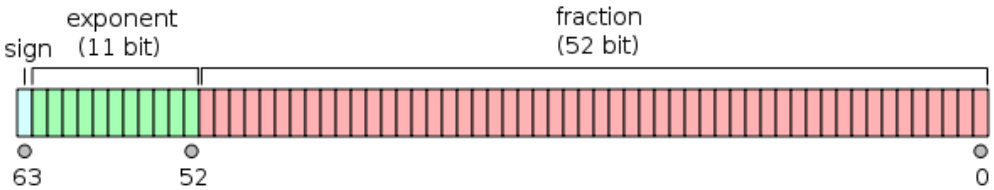

Figure 2: Double precision floating point format [8]

Depending on the exponent field, floating-point values are either normalized or denormalized (Figure 3). When the exponent bits are not all 0's or all 1's, in either precision, the values are normalized. To interpret normalized values, the significand is assumed to have a leading 1 (it is assumed you have at least one significant digit), which provides an extra bit of precision since this leading 1 is not stored explicitly. The extra assumed bit in normalized floating-point values gives 24 bits of significand and denormalized values' significands are interpreted only using the 23 bits. Otherwise, when the exponent is all 0's and the significand is non-zero, the values are denormalized and do not assume a leading one in the value. For example, $1.0 * 10^{-6}$ is normalized and the leading 1 can be assumed (binary only stores 1's or 0's so a normalized number will always start with "1.") while $0.01 * 10^{-127}$ is denormalized. Zero cannot be represented using a normalized value since implying the leading significant digit would make the value non-zero. Negative and positive zero are represented using the denormalized value $0x80000000_2$ (only the sign bit is set) and $0x0_2$ in single precision respectively.

Other special values are defined when the exponent is all 1's (Figure 3). For instance, when the significand is zero, the value is positive or negative infinity depending on the sign bit. When the significand is non-zero the value is a not a number (NaN) and can be either signaled NaNs (SNaNs) or indefinite NaNs (QNaNs). Indefinites are used in the event of square root applied to a negative number, subtracting two values of infinity, or using an empty register or indefinite as an operand. The standard also specifies how to handle how instructions execute when applied to one or more of these special values.

The floating-point specification also documents many rules that describe how each instruction must be performed to ensure predictable rounding of each result. This is

necessary because floating-point does not have infinite precision, so rounding must occur after each operation. The accuracy of instructions can be bounded using ULP (Units in Last Place) as a metric. For most instructions, the absolute error should be limited to 1 ULP. To aid hardware and software designers in reaching these error requirements, various rounding modes (described further in Section 2.1.3) are documented.

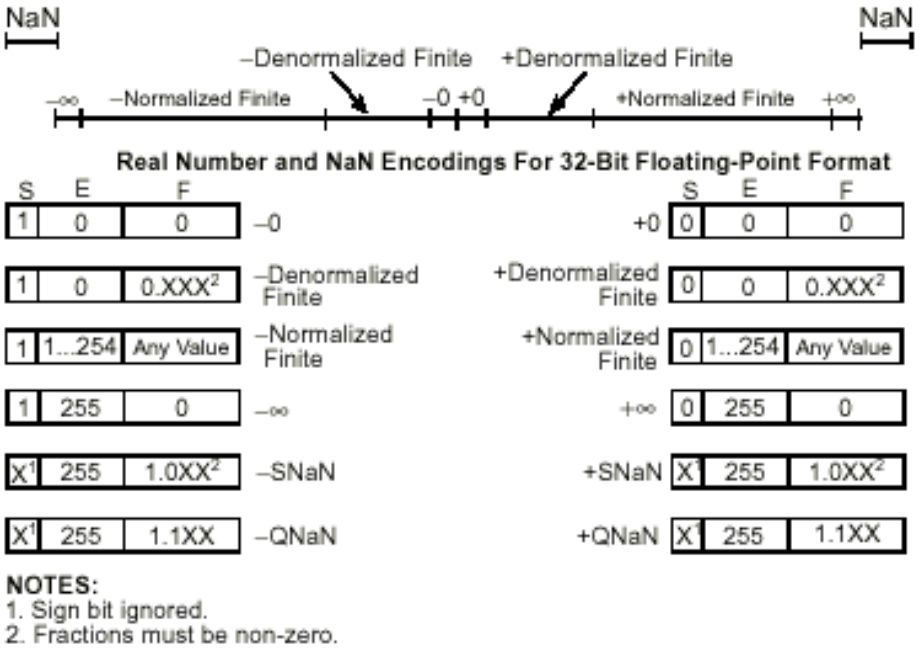

Figure 3: Floating-point values in the IEEE standard [9]

The IEEE 754-2008 standard also defines many more floating point details. Those that we have presented are relevant to our findings in the Digital Breast Tomosynthesis application we analyze in this thesis.

2.1.3 Performance of Floating-Point Implementations

The IEEE 754-2008 standard gives a set of rules that the notation and operations using the notation must follow. How these rules are achieved is up to the hardware or software FPU (floating-point unit) designer. They must decide on an implementation after weighing the tradeoffs of speed, complexity and the level to which they claim they are IEEE 754 compliant. Furthermore, the language designers and compiler writers are

responsible for ensuring the compiled instruction stream complies with the IEEE standards. Finally, the programmer must verify the language, compiler and architecture they use will provide the floating-point standards necessary for their code.

This hierarchy leads to many different locations for differences to appear. For instance, the IEEE 754 standard defines several rounding modes including round to nearest (which should be the default), round to zero, round upward and round downward [10]. While the language, compiler and hardware may implement the round to nearest rounding mode, they may not implement the others. This design choice makes it IEEE compliant, but can result in differences between two compliant implementations if one chooses to use a different IEEE 754-2008 compliant rounding mode. For instance, the NVIDIA GPUs we use in this project are only capable of round to nearest in double precision while the CPU code could potentially use the other IEEE rounding modes if the user specifies something other than round to nearest. This introduces one of several examples of how floating-point results can diverge between two systems.

The FPU designer may choose different options largely based on the cost and speed of the implementation. Whether you are writing a floating point unit purely in software or designing a circuit to perform floating-point in hardware, different implementations of instructions will require different numbers of computation cycles. Perhaps speed is not an issue so a slow software implementation is chosen that follows all IEEE 754 specifications and by default uses the most accurate mathematical operations. In other cases, the FPU may be designed on a small FPGA [11] where space is limited, so only limited IEEE compliance can be afforded. In the GPU's case, performance is the highest priority for time dependent graphics operations as well as in scientific uses. The

desire for speed and efficiency generally opposes great accuracy. These FPU design challenges are usually solved well before the programmer writes a program that uses that FPU, so the designer attempts to predict what accuracy and performance their customer requires. Understanding the design choices that were made can be essential to the performance and to the results of the code, especially in GPU applications.

2.2 GPGPU

GPUs have been included in consumer and commercial computer systems for many years. Their purpose has usually been for accelerated processing of graphics data and providing it to a computer's visual display with limited CPU intervention; allowing the computer to work on other tasks. NVIDIA and other GPU manufacturers have recently made it easier for software programmers to exploit the hardware to be used for non-graphical programming like scientific applications. Among many other applications, the GPU is used to accelerate Digital Breast Tomosynthesis, the medical application used in this research.

2.2.1 History

Traditional graphics processing involves a large number of independent calculations, generally one for each pixel or block of pixels. The processor industry has been struggling to continually provide the performance that has traditionally followed Moore's Law [12]. Smaller transistor sizes have allowed higher complexity processors with faster intra-chip communication, which brought faster processing speeds. As transistor shrink has slowed, the increase of processing speed has been less than in the past. In these times, GPUs and other parallel architectures are the solution to achieve greater computation efficiency.

While some motivated individuals previously programmed scientific codes using graphics languages like OpenGL or DirectX, GPU use in general computing did not take off until NVIDIA made an effort to make the hardware more accessible. Languages like NVIDIAs CUDA and Khronos Group's OpenCL [13] were created for non-graphical coding purposes and the term GPGPU (General Purpose GPU) was born. The GPU architecture introduced tremendous raw performance capabilities in comparison to modern CPUs (Figure 4).

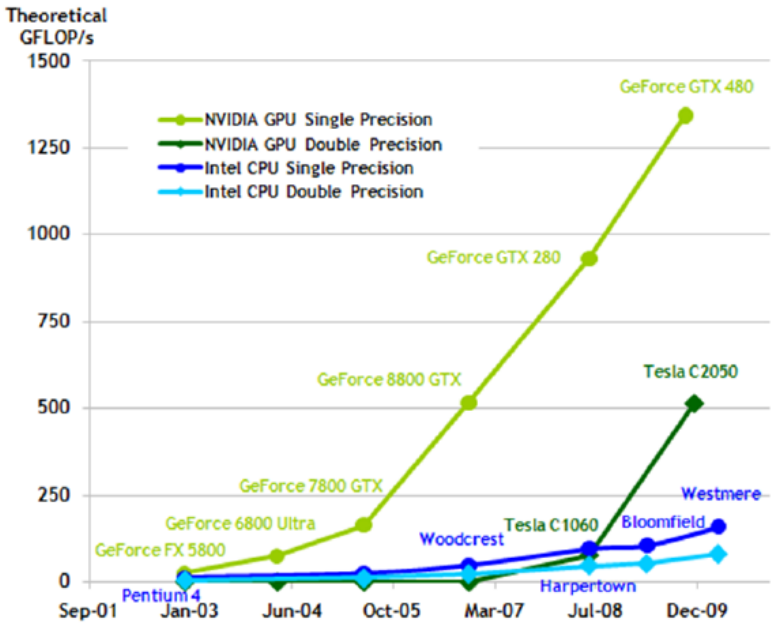

Figure 4: GFLOP/s performance of modern computing architectures [14]

NVIDIA led the way by creating the CUDA (Compute Unified Device Architecture) programming language. CUDA (the language and its compiler) is tailored directly to NVIDIA's GPU architectures and provides industry-leading performance for many applications. The language is relatively easy for novice GPU programmers to become comfortable with as it is based on the C/C++ language.

CUDA allows for programming of three generations of NVIDIA GPU hardware at the time this thesis was completed. The G80 architecture (90nm and then 65nm process) was introduced in 2006 and in 2007 CUDA was unveiled to be used on some G80 graphics cards. In 2008, the GT200 was released using a 40nm process and an increased number of cores. The GT200 series brought some of the first double precision capable GPUs, much to the delight of the scientific community. Finally in 2010, the Fermi (GF100) architecture was released using the same 40nm process but with more significant architecture changes than the switches between previous generations. The GT200 series architecture is used in our results due to Fermi not being available at the start of our research. We will discuss the documented changes in Fermi related to our research in section 3.3.

AMD has also made their GPUs more accessible to programming through the parallel architecture language OpenCL (a language that can also be used on NVIDIA devices) [13]. While some AMD hardware theoretically outperforms NVIDIA's due to more processing cores, the CUDA language coupled with NVIDIA GPUs have usually achieved better performance [15] than similar code in OpenCL on AMD devices. CUDA is the most common choice of language used in scientific GPU computation projects, and will remain so unless OpenCL becomes a widely used standard due to its heterogeneous architecture support. For these reasons, we focus on NVIDIA GPUs and applications written in CUDA.

2.2.2 Architecture

Our research does not focus on optimization strategies but it does involve discussion about how problems are parallelized, which necessitates an understanding of

the architecture. The GPU consists of hundreds of processing cores and functions as a SIMD (single instruction, multiple data) device. This implies that the cores perform a single instruction at one time across many pieces of data when the computation is devoid of any dependencies. This is a natural progression from the graphics focused nature of the hardware where each pixel calculation is largely independent. While the CPU dedicates a large portion of its chip to cache and a complicated control unit, the GPU has very limited control and cache to focus its transistors on data processing as seen in Figure 5.

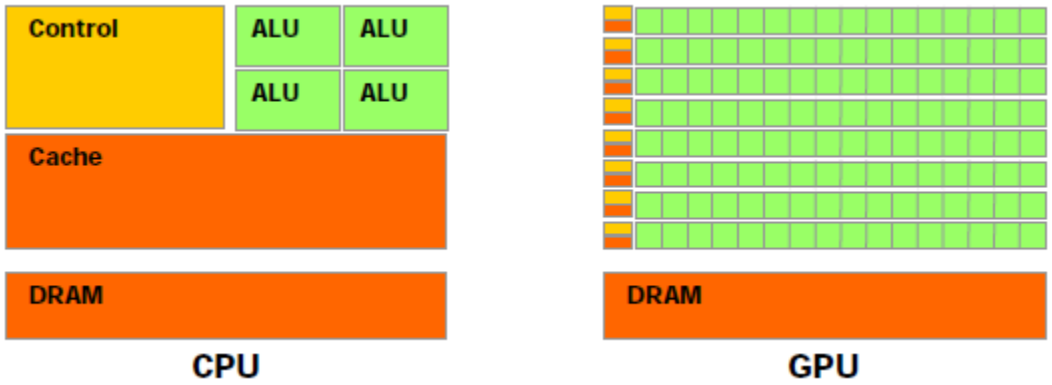

Figure 5: The GPU dedicates most of its chip real estate to data processing[14]

In contrast to many SIMD architectures, the execution of instructions is a bit more complicated on the GPU (Figure 6). The hundreds of streaming cores are grouped together in SMs (Streaming Multi-processors). There are groups of 8 streaming cores per SM on the GT200 (known as Tesla) architecture and 16 on the Fermi architecture. The threads that are distributed to the cores also have a structure. A grid consists of many blocks of threads. Blocks of threads are given to each SM with their own program (list of instructions). The SM decodes the instruction and schedules warps of 32 threads from a block (these numbers vary depending on the hardware generation). The warp of threads is sent to the SM's streaming cores that collaborate to complete them in parallel. The cores are pipelined and multiple warps are in the pipeline at any point. Warps of threads can be

rapidly switched to hide memory latencies so that other some may run while others stall on long memory accesses.

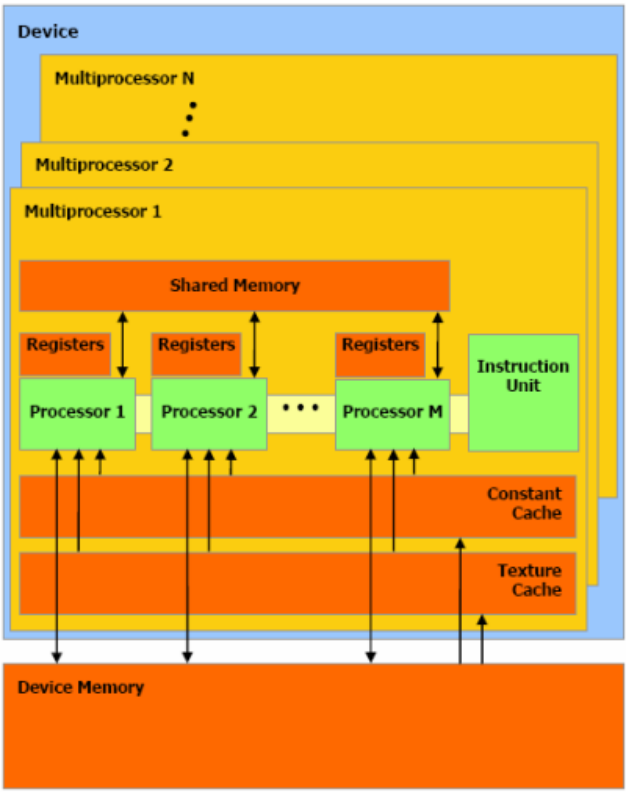

Figure 6: The NVIDIA GPU architecture, including memories and cores [14]

The memory hierarchy (also shown in Figure 6) consists of global, shared, constant and texture memories. Global memory is DDR RAM located on the graphics card and is separate from the CPU memory space. The SMs have a very wide (up to 512 bit) high speed bus from the global memory to the registers and shared memory. Shared memory local to the SM can only be accessed by threads within that SM (all the threads in one block). The shared memory has a latency of only several times that of registers while the global memory can take hundreds of cycles to retrieve data. Since the CPU and GPU have distinct memory spaces, data that will be processed on the GPU must be transferred manually by the programmer and contributes to the overall completion time of the application.

2.2.3 Language

The CUDA language was developed by NVIDIA to give the programmer command over the architecture while remaining easy to write and maintain code for. The language is an extension of C++ that allows the programmer to describe parallelism and manage the various memories and the hierarchy of compute units in the GPU. CUDA code that is intended to run on the GPU can be written in the same C++ file as CPU code. The GPU code is contained in a kernel (a function denoted with GPU device directives).

NVIDIA's NVCC compiler separates the CPU code and the GPU kernels. The CPU code is run through the host C++ compiler and the GPU code through a GPU specific compiler that NVIDIA created. The GPU code is compiled into an intermediate representation called PTX, which resembles x86 assembly code for the CPU. The PTX is then compiled into binaries that run on the targeted GPU. The PTX code can be output into a text file if the programmer wishes to analyze the compilers actions as we will do throughout this thesis.

The programmer writes the code similarly to how serial CPU code is written. The major difference is that in the kernel, the code is the program for each thread (of which there are thousands). CUDA library functions are used to transfer data from the CPU memory space to global memory, and the code in the kernel can move that data to the faster shared memory of the SM owned by the block. Many libraries are available through NVIDIA and other outlets to aid programmers in achieving near optimal performance in common functions.

The GPU is proving itself to be a very capable programming device in research and commercial applications. It is capable of performing many of the necessary

operations that CPUs provide with much greater parallelism than exists even in multi-core processors, including operations involving high precision floating-point values.

2.2.4 GPU Floating-Point

Floating-point calculations on a GPU were one of the greatest weaknesses of the hardware when CUDA was first introduced because of the nature of graphics programming. For much of GPU history, graphics computation used integers primarily and only within the last decade added support for single precision values to provide a greater range of computations. Double precision was not necessary to compute the color or action of a pixel on the screen, so such precision was not supported until GPGPUs were used for non-graphics applications. The immaturity and limitations of the floating-point implementation in GPU hardware continues to hinder the performance even in the Fermi architecture – some of which we discuss in this thesis.

IEEE floating-point compliance was not necessary prior to GPGPU since errors greater than those allowed by the standard could be tolerated because the computation was only for graphics purposes. As a result, GPUs can fit many cores on their chips because the ALUs are very basic and most of the complicated instructions are completed in software, which also allows for simpler control logic.

The NVIDIA G80 series GPUs did not support double precision at all. The GT200 series squeezed some double precision FPUs on the die, but only one per SM (an SM contains 8 streaming cores). The result is $1/8^{th}$ peak performance for double precision floating-point computation compared to single precision. Even in the Fermi consumer level architecture, there are only 2 double precision FPU per SM (the SMs in Fermi have 16 cores) meaning still 1/8 performance. The scientific computing Tesla Fermi cards

have 8 double precision FPUs per SM and provide ½ of single precision floating-point performance for double precision.

In addition to performance issues, the G80 series is very weak in its IEEE compliance for single precision floating-point. For example, IEEE compliant division was not availalbe. The GT200 series follows the IEEE standard more closely, reducing the ULP error in many math functions [14]. The GT200 also implemented 64 bit (double precision) floating-point, which provided greater precision for those algorithms that require it despite the performance hit. Fermi complies to IEEE standards even further and has corrected some of the remaining weaknesses from the GT200, such as more accurate fused multiply add instructions. Fermi also introduces ECC capabilities to ensure memory accuracy on the Fermi-Tesla cards. Previous card ignored memory faults which could become noticeable in very long running applications. We do not consider memory faults in these experiments.

The NVIDIA Programming Guide [14] and the CUDA PTX ISA manual [16] address several issues that we experience and describe in detail. These references list several deviations from the IEEE 754-2008 standard for single precision for the compute 1.x devices used in our experiments:

- There are no configurable rounding modes, though some operations support various modes, which can be specified using CUDA functions.
- There is no mechanism for detecting floating-point exceptions.
- Denormalized numbers are not supported.
- Addition and multiplication are often combined.

- Division and square root are implemented via the reciprocal in a non-IEEE compliant way.
- Several functions offer IEEE compliant (slow) software implementations.
- Safe functions are provided for type conversions.

NVIDIA states that floating-point operations may compile to different instructions when using different compiler versions (for the same or different hardware), which can also cause differences in the results. For instance, compute 2.x (Fermi) devices have the following differences from previous devices:

- Single precision division is IEEE compliant by default. The compiler option –prec-div=false can be used to match the approximate division of compute 1.x devices.
- Denormalized numbers are supported. To flush denormalized numbers to zero, as in compute 1.x, the –ftz=false option should be used.
- Single precision MAD is now a fused multiply-add (FMA) and is IEEE compliant (as in double precision on 1.x compatible devices).
- Square root functions are IEEE compliant and a compiler option –prec-sqrt=false will force the approximate version to match the 1.x device.

This thesis discusses how to find the differences mentioned in the programming guides, as well as some that are not. It also considers the tradeoffs for performance and accuracy in a real algorithm. Finally, we present the necessary actions to make the GPU code match the CPU code when it is necessary to produce identical results, which makes use of some of the guide's recommendations. We explore the differences between both single and double precision codes on the CPU and GT200 series GPU.

2.2.5 NVIDIA Tesla C1060

In this thesis we do most of the experimentation on a NVIDIA Tesla C1060 GPU, which is the scientific computing version of NVIDIA's GT200 GPGPUs. This has the architecture of the GT200 series but with 4x more memory (4GB) to satisfy the needs of large scientific problems like Digital Breast Tomosynthesis, the algorithm used in our research. This thesis, is not focused on optimization for performance, but focuses on understanding the floating-point differences that arise when programming for CPU or GPU architectures. The performance specifications of the Tesla C1060 are shown in Figure 7.

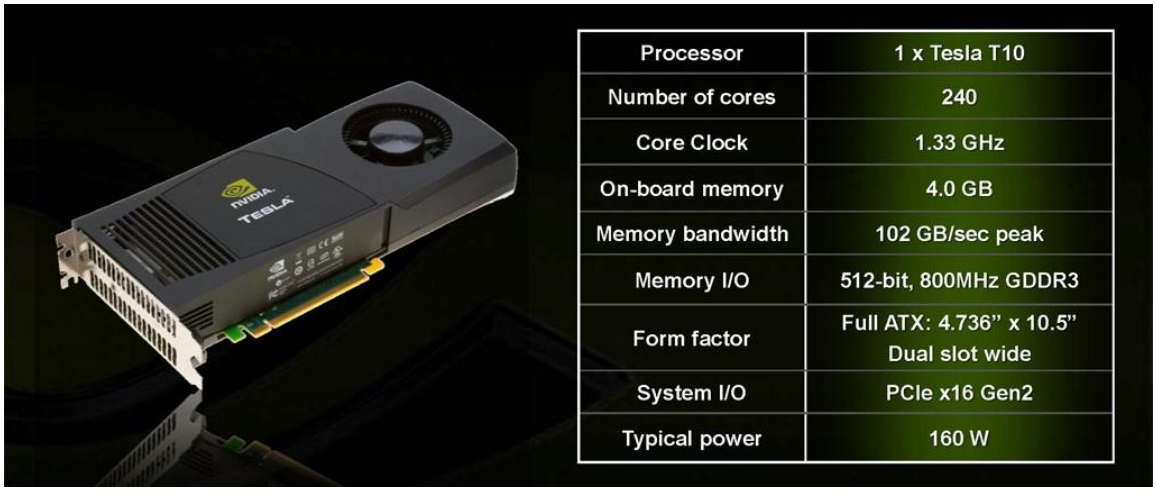

Figure 7: Tesla C1060 Specifications [17]

2.3 Tomosynthesis

Digital Breast Tomosynthesis (DBT) is a form of mammography that uses data from multiple low-dose x-ray projections (Figure 8) to create a 3-dimensional reconstruction of breast tissue using a maximum likelihood method [18]. The computation creates many high resolution two dimensional images that make up the 3D image. Due to their size and the complexity of the algorithm, generating the result is very compute intensive. Schaa et

al. [19] created a GPU version of the algorithm that is orders of magnitude faster than the original CPU code, which makes the code more amenable to being deployed in a medical setting. In this research, we use this GPU accelerated version of DBT and its CPU version that was developed at MGH and the Rochester Polytechnic Institute [20].

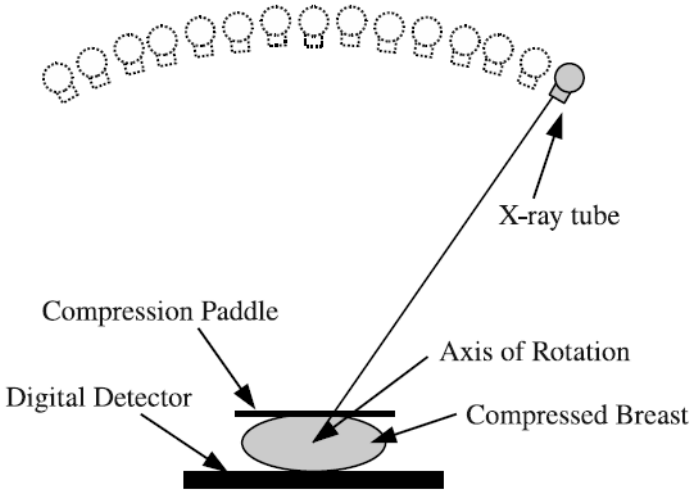

Figure 8: Tomosynthesis System Diagram

The results of the GPU and CPU versions with the test images appear correct to the naked eye but there are differences in the numerical results. Before this work, the differences in the raw data were assumed to be due to floating-point error on the GPU and that the CPU was the more accurate version due to the CPUs reputation for floating-point standard compliance. Our research challenges the belief that the GPU is performing less accurately. In addition we create a GPU version that matches the CPU output in order to verify that the algorithm performs without coding mistakes.

Tomosynthesis completes 6 iterations of the maximum likelihood algorithm to achieve a clean reconstruction of the breast tissue. Within each iteration is a forward and a back projection process that create an image based on the input data of all 15 x-ray detectors. Forward projection creates a record of the absorption of each X-ray for all

pixels in the 2D plane. Backward projection uses this data to create contrast values for each 3D image created with 45 slices deep of high resolution 2D images. Each of the 6 iterations refines the contrast levels of each pixel based on the X-ray absorption of the surrounding pixels. Figure 9 shows the layout of the algorithm, which will be helpful to illustrate the location of the floating-point differences discussed in the following chapters.

```
initialize 3D volume {make a guess}
while likelihood is too low do
   {forward projection}
   for each projection do
      for each ray do
         trace ray using volume guess
         compute new projection value
         compute attenuation value
      end for
   end for
   {backward projection}
   for each voxel in the 3D volume do
      for each X-ray source do
         compute error of ray (projection) with the
               measured value (sinogram)
         use attenuation values to refine volume guess
      end for
   end for
end while
```

Figure 9: Tomosynthesis with Maximum Likelihood Estimation Method

Our research is motivated by the concern expressed by the medical and programming personnel involved with DBT about the accuracy of the GPU code, who must determine whether this accelerated code is safe to use on human patients. Despite the fact that the CPU code has floating-point error of its own, a new version that differs from those results is difficult to accept blindly. This is true even if the error is within a tolerance and repeatable in example scans. The data acquired from X-rays of a body tissue could have many unforeseen anomalies, which could trigger computational differences not experienced in a limited set of test examples.

The GPU accelerated DBT completes in less than a minute whereas the original CPU DBT took hours on Pentium IV workstations at MGH. With our research, we can determine whether the differences are coding bugs disguised as floating-point error or if they are from operations that can be performed in a faster or more accurate way by overriding default compiler actions. By achieving this understanding of the differences, the fast GPU code will be safer to use in medical applications.

2.4 Related Work

It is a fact that GPUs do not fully support IEEE standards and as a result differences in floating-point code have been recognized in many others' work. In the early years of GPGPU, Meredith et al. [21] analyzed tradeoffs in a quantum Monte Carlo application case study comparing the CPU results with the GPU's in single and double precision against a baseline "long double" (80 bit precision) result created in CPU code. The results were compared, but only by measuring the deviation from the accepted baseline. The error was blamed on possible reordering of floating-point instructions in the parallel code and on the GPU lacking full IEEE 754 support. While these assumptions may be correct, there was no analysis of what differences were present, where they occurred and if they could be corrected. The GPU was deemed accurate for the Monte Carlo application based on comparable error to the CPU. The work fails to determine the true causes for the differences, which leaves the possibility that the differences could be avoided or even that some minor bugs exist. Our research intends to discover the reasons for the differences in CPU vs. GPU code for DBT and also discuss their effects on the result's accuracy.

To our knowledge, the most detailed GPU floating point accuracy study to date is Karl Hillesland and Anselmo Lastra's GPU Floating-Point Paranoia [22]. The work

specifically focused on poor rounding performance among common math operations using the graphics programming languages and early GPU hardware. Since it was published in 2004, the work was done well before widespread use of GPUs for scientific computing, CUDA, and many attempts by NVIDIA to improve IEEE compliance. This work was also produced well before the IEEE 754-2008 standard was released. Despite the age and limited relevance of this work to today's GPUs, several more recent published papers refer to the work by Hillesland et al. as an explanation for the GPU results differing from their CPU results [23] [24] [21]. In addition, many other CUDA programmers assume that GPUs have poor floating-point accuracy because of the results found in these pre-CUDA studies.

Mian Lu et al. [25] have created an extended precision (also known as arbitrary precision) GPU implementation to allow for greater accuracy than even double precision floating point. Its relevance to this work is that it may be necessary to use extended precision if there are cases where the GPU cannot or is perceived as not to be able to achieve results that are as accurate as the CPU code being used. Our research describes programming techniques that will protect GPU code from being less accurate than CPU code as well as illustrate instances where the GPU can exhibit greater accuracy. This may relieve the need for extended precision GPU code if it is used to match an approved CPU code. This will increase GPU performance since the non-native extended precisions are slow.

Different languages and compilers will not necessarily resolve the issues described in this work because many of the differences exist in the intermediate representation and the instruction set of the GPU. For example, OpenCL codes may

produce the same differences because they are compiling to the same instruction set as CUDA. Another project, Ocelot [26], is able to compile from PTX (CUDA intermediate representation) to various GPU and CPU architectures. This also does not solve the problem because the PTX produced by CUDA already contains the instructions in the form that cause differences between CPU and GPU code.

2.5 Experimental Setup

Our test system throughout this thesis is an Intel Xeon W3580 3.33GHz (4 core, 8KB L1 cache) CPU with 6GB system RAM. For serial code implementations, only one core of the CPU was used. The GPU is the NVIDIA Tesla C1060, described in section 2.2.5. Our C++ compiler is g++ 4.4.3 and compiles to the x86-64 CPU target for all CPU tests, using default options for mathematical precision. The operating system used is 64 bit Ubuntu 10.10 (Linux). The GPU code is compiled using CUDA version 3.2 for the Tesla C1060 target. Our results directly apply to this setup, but the lessons apply to other compilers and architectures as well.

In order to test the results of DBT, an input data set (or phantom) was provided that is used to test other DBT and tomography codes for any architecture. The data is an ACR (American College of Radiologists) phantom (Figure 10) and is the X-ray data recorded from a man-made material with similar qualities to breast tissue. Inside the material is an alphanumeric code that should be visible deep inside the 3D image produced by DBT if the algorithm is working correctly. DBT performed on this data was successful in producing a quality reconstruction of the image. Additionally, our experimental results were verified using other breast tissue phantoms.

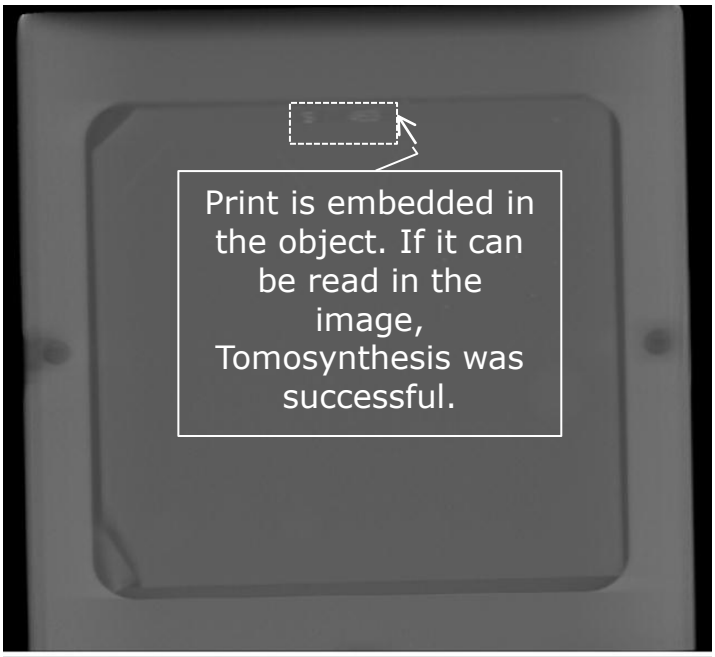

Figure 10: Output of DBT with ACR phantom input

3 FLOATING-POINT DIFFERENCE EXPLORATION

3.1 Introduction

This chapter describes several important differences that exist between CPU and GPU implementations that arise from the use of floating-point computations. Through two programming examples, DBT and π, some floating-point differences between CPU and GPU codes are highlighted as they appear in these real applications. The applications are programmed for CPU and GPU architectures in the way they would generally be created by software developers and the results are shown to be different. A technique of debugging is described that entails finding, understanding and resolving the floating-point differences in order to produce dependable, high quality GPU applications that match or even rival the accuracy of CPU applications. By modifying GPU codes to produce identical results to an original CPU implementation of the same algorithm, we can more confidently address the quality of the GPU version.

The first (Section 3.1.1) is an introductory example that computes π with different results in C++ and CUDA versions, demonstrating floating-point differences due to associativity. These differences are common in other parallel architectures in addition to the GPU. In Section 3.2, we document our exploration of the Digital Breast

Tomosynthesis (DBT) application where we find several differences not related to associativity that exist between the CPU and GPU implementations and successfully modify the GPU application to produce identical results to the CPU. Finally, in Section 3.3 we discuss the floating-point differences of the new Fermi architecture.

3.1.1 π – An Introductory Example

Inconsistencies between the results of CPU and GPU applications commonly arise from differences in floating-point precision [27]. We introduce our exploration of differences in GPU codes by discussing the most common difference, associativity, using an algorithm for computing π [28]. This example exhibits different results for each application because the floating-point computations are reordered when the algorithm is parallelized. While reordering integer additions is mathematically acceptable (the associative property), reordering floating-point instructions is not associative and can potentially cause different results.

The serial π CPU program (Figure 11) was presented in an introduction to OpenMP, presented by Mattson et al. on behalf of the Intel Corporation [29]. The algorithm refines π by executing *num_steps* iterations of a small computation, an accumulated sum and a final average. It is expected that a greater number of steps will increase the accuracy of π until the data type reaches the limits of its precision.

```
 1:   static long num_steps = 100000;
 2:   float step;
 3:
 4:   void main()
 5:   {
 6:       int i;  float x, pi, sum = 0.0;
 7:
 8:       step = 1.0/(float) num_steps;
 9:
10:      for (i=1; i<= num_steps; i++) {
11:          x=(i-0.5)*step;
12:          sum = sum + 4.0/(1.0+x*x);
13:      }
14:      pi = step * sum;
     }
```

Figure 11: Sequential π program

This algorithm is an embarrassingly parallel calculation, which means no modification of the underlying algorithm is needed to expose the parallelism necessary for implementation on a GPU or other parallel architecture. The technique used to parallelize the application is called a Map Reduce computation [30, 31]. The for loop (lines 9-12 in Figure 11) can be parallelized easily because there are no inter-loop data dependencies. The accumulating sum (line 12 in Figure 11) can be computed afterwards by using a reduction computation. The reduction method is the most common way to parallelize this type of operation on GPUs and other parallel architectures [32-35]. The GPU computation of π, using the map reduce technique, is depicted in Figure 12. The code is written in CUDA and performs each iteration of the loop in concurrent threads. An array of length num_steps maps the calculation of $\frac{4.0}{1.0+x*x}$ (from lines 11 and 12) to each entry of an array that corresponds to its loop iteration in the serial code. The cumulative sum is then calculated via a reduction tree where each GPU thread sums two neighboring values in the array and stores these intermediate results. After each thread completes one sum, the array of intermediate results is added in a similar fashion. This

process repeats until the last combination is completed, which contains the overall sum and completes the map reduce computation of π.

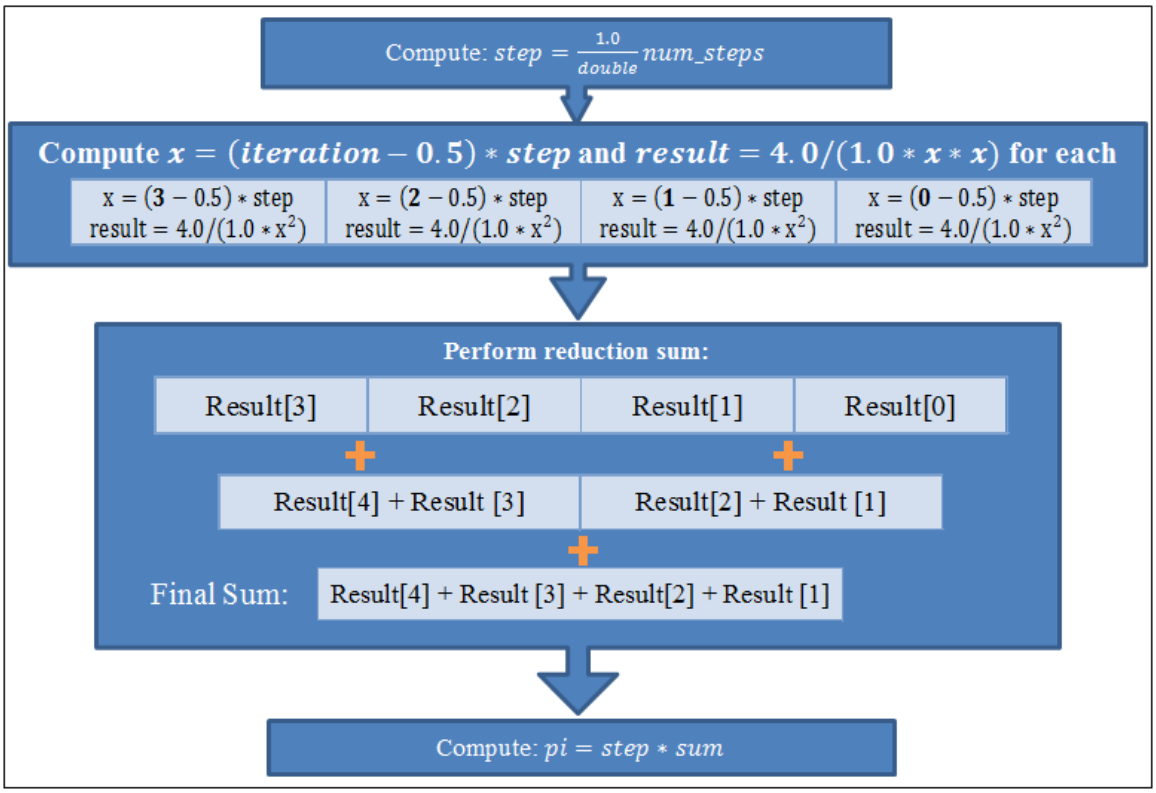

Figure 12: GPU version of π program from Figure 11

Executing the single precision π CPU implementation shows a startling loss of accuracy. Single precision can only offer slightly over 7 significant digits of accuracy, so a value matching 8 significant digits of π would be the highest achievable accuracy. Both the CPU and GPU versions approach the true value of π [28] at a couple thousand steps but the CPU version begins to lose accuracy shortly after approaching the value. As a result, π accuracy degrades over the number of steps when compared to the actual value of π. The GPU result retains great accuracy at a high number of steps (Figure 13).

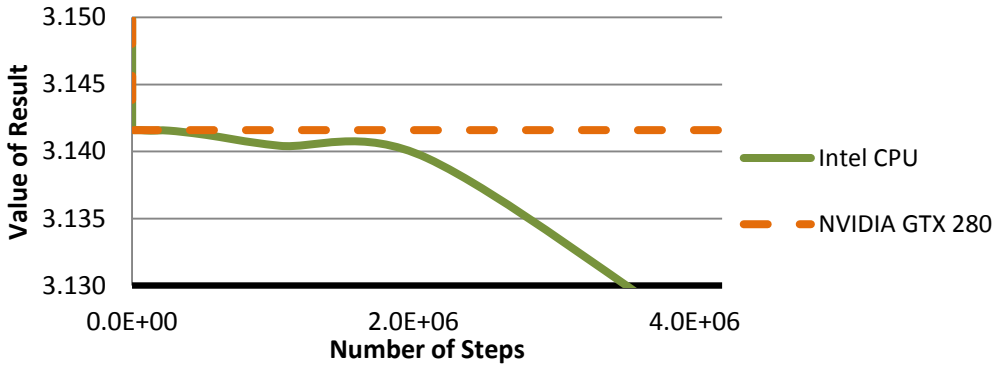

Figure 13: π single precision results comparison

Double precision has the same theoretical downfall but has 14 significant digits of representation to account for it. The differences still accumulate but take many more iterations (a larger accumulated sum) to be noticeable. In fact the CPU only starts to diverge from the GPU version near the limits of the GPU's available resources.

The differences in the C++ and CUDA versions arise specifically during the reduction portion of the program and are surprising to those not aware of floating-point's intricacies. As mentioned, floating-point computations are not associative due to rounding or loss of representation at different points of the computation when compared to a serial application, as is the case in our GPU and CPU code. The rounding will occur after the additions of different values, which may result in rounding differently at the least significant digit. Furthermore, depending on the size of each value, the addition of very large and very small values could lose more precision than summing values of similar magnitude. As discussed earlier, floating-point can represent very large or very small values because it acts as a sliding window. If the window of representation slides towards a large value, a very small value cannot be represented in the same window and can be partially or completely lost in an operation like addition. More specifically,

accumulating values serially will sometimes result in a large value that each successive small value is added to, resulting in diminished accuracy of the results.

The reduction style of computation avoids the issue in accumulating floating-point values in a way that is similar to binning. Binning describes collecting and computing small groups of values and then computing the final result by combining each result. This can result in similar magnitude values being added at each step, which can increase the accuracy of the computation. An example of this theory is illustrated in Figure 14.

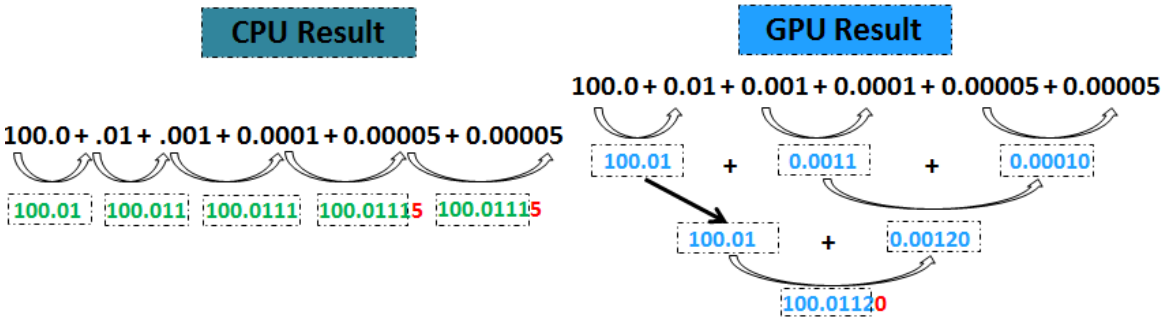

Figure 14: Serial addition can lose precision vs. reduction sum. Numbers surrounded by a box represent the actual result floating-point value.

It is often helpful to make the results of the CPU and GPU application match to make sure the codes are free of bugs. In this case, the GPU is actually more precise than the CPU and to make the GPU code match the CPU result, the computation would have to be made less precise. In performing this comparison, we essentially debugged the CPU π application because we realized why and how the CPU was losing accuracy. It is possible to make the CPU results closely or identically match the GPU, if the CPU code uses a similar binning method. For instance, if the sum is binned every 100 loop iterations and stored separately until a final sum at the end, the CPU produces nearly

identical results to the GPU. If the serial code was developed even further, using a tree structure as in reduction, the results can exactly match.

Associativity and reduction precision issues are not concepts specific to the π implementation or the GPU architecture. Other parallel architectures use reduction computations and will experience the same differences in their results. It is also true that aware programmers could program the serial code using a binning approach to avoid accuracy loss. This π example introduced a basic floating-point difference that exists between CPU and GPU codes. Understanding this source of differences can be essential to ensure accurate results from a program written for either architecture. While this is a common cause of differences in floating-point computation across different architectures, the GPU has other differences that are directly related to the CUDA language, compiler and GPU architecture. These differences will be shown in the exploration of DBT.

3.2 CPU and GPU Floating-point Differences in Digital Breast Tomosynthesis

3.2.1 Introduction

The motivation of this work is to ensure that the CUDA version of DBT performs floating-point computations with equal or better precision compared to the CPU DBT implementation. In order to do so, we must find and understand the differences in their computations in order to resolve them. Accomplishing this means that we can prove the accelerated application is as safe to use as the slower CPU version. While we cannot guarantee the CPU code is correct, having both produce the same results gives added confidence in an absence of bugs. In this DBT experiment there is no golden final result available. The results produced by the GPU implementation need to match the x86-64

CPU DBT application or at a minimum the differences must be precisely understood in order to determine that the code is free of bugs.

The output from the CPU and GPU DBT do not match and this causes concern that the GPU code has bugs or suffers a loss of accuracy that may cause a false reading by the radiologist. Unlike the π example, there are no reductions in the DBT code to explain the differences. DBT performs on the algorithm using an identical ordering of operations on each, so associativity is not the reason. There must be other less obvious floating-point differences in CPU and GPU codes. This section documents the discovery of the differences in the CPU and GPU code and discusses the implications of each. Through this experiment we were able to create a CUDA version of DBT that is fast and matches the CPU results exactly, verifying that the GPU is as accurate as the CPU for the application and that speedups reported are legitimate.

3.2.2 Initial Comparison

We compare two versions of DBT code:

- A **C++ DBT implementation**, of about 1500 source lines of code, that utilizes single precision floating-point computation and compiles and executes as an x86-64 application on our Intel Xeon based test system [20].
- A **CUDA DBT implementation** optimized for the GT200 Tesla series NVIDIA GPUs, also in single precision, that compiles and executes successfully on our test system's Tesla C1060 GPU [19].

From these, double precision x86-64 CPU and CUDA implementations were created so that both single and double precision implementations could be compared.

The first comparison test was to compare the output values. The final result that is produced by DBT (using the ACR Phantom data as input, see Section 2.5) is a 3 dimensional 1196x2304x45 resolution gray scale image. The image is represented with integer contrast levels between 0 (black) and 20000 (white). This made testing the differences in the final result a trivial comparison of each pixel's integer contrast value as produced by the CUDA and x86-64 applications.

Single Precision

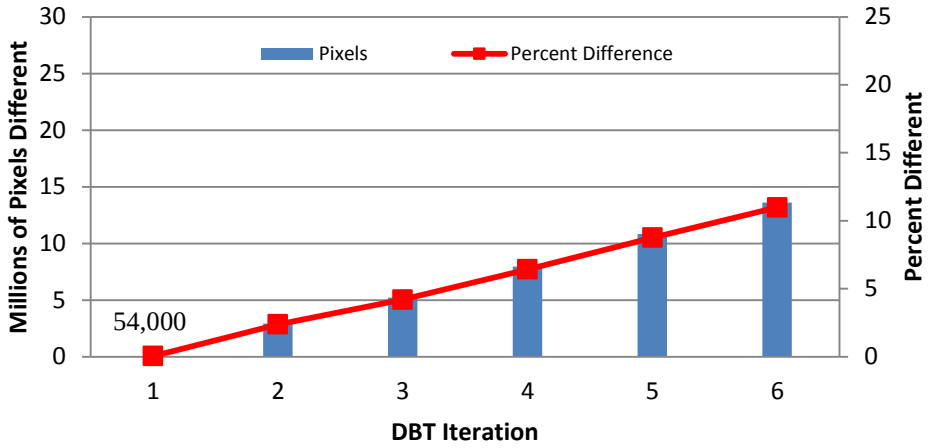

Figure 15: Differences per Tomosynthesis iteration (single precision)

The initial comparison of single precision CPU and GPU DBT results showed many differences after the final DBT iteration (Figure 15). After the first iteration, there were approximately different 54,000 pixels and the number of differences increases each iteration as the differing pixel values are used in computations of other pixels. After the final iteration, nearly 11% (over 10 million pixels) of the image's pixel contrast values differed. The average difference per pixel (Figure 16) was reassuring at slightly over 2 values of contrast (.01% of the contrast range), but there were large maximum differences of approximately 20,000 (the entire contrast range). These large differences were few and sparsely distributed throughout the image, which is why the images appeared identical.

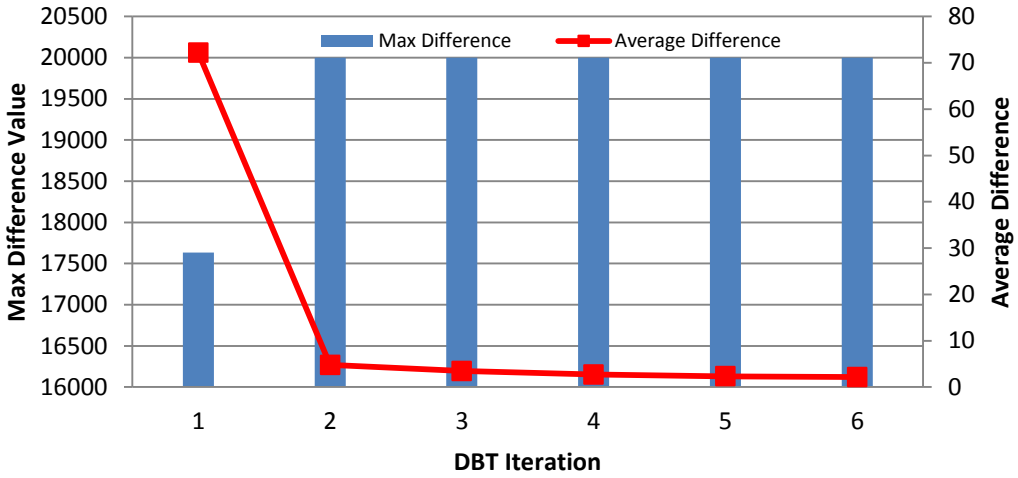

Figure 16: Differences in output image (single precision)

The results in Figure 15 and Figure 16 show the differences in the image after each complete iteration of DBT, where 54,000 differences were observed following the first iteration. We investigated the DBT implementation at a finer granularity to obtain a better understanding of where the differences originated. The DBT algorithm involves two parts: forward projection and back projection (Figure 9). The forward projection traces each X-ray and calculates a value that represents the amount of radiation from the X-ray absorbed (attenuation) for each pixel of the scanned tissue or material. There are 15 X-rays that create 15 sets of this absorption data at 1196x2304 pixels each. We observed over 40,000 differences in the absorption data after a single iteration (Figure 17) and that number increases each iteration until the last iteration of DBT where the absorption data is 16% different between the CPU and GPU implementations.

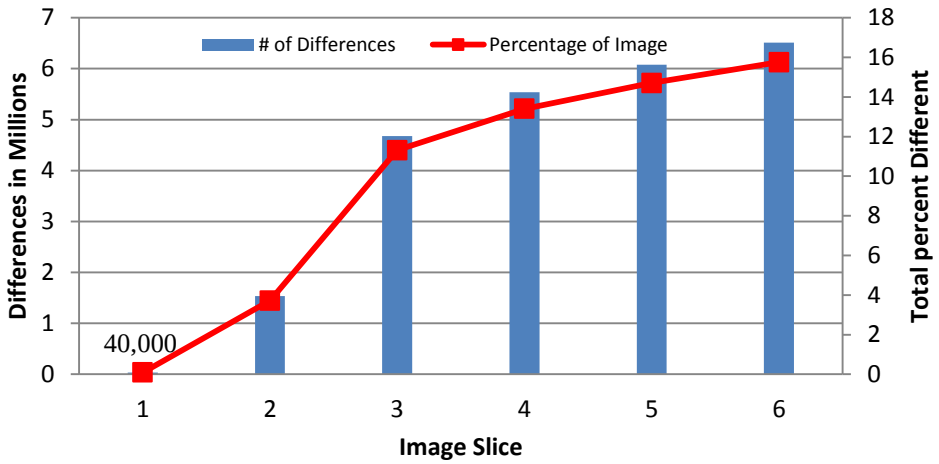

Figure 17: Absorption differences in single precision DBT

The forward projection also produces projection data, which depends solely on the absorption result. Due to this dependence, these results are less useful for our experiment. Our initial testing results showed that forward projection is where most of the differences between the CPU and GPU implementations arose.

Double Precision

The double precision results of the CPU and GPU also have differences, which leads to the hypothesis that double precision suffers some of the same issues as single precision. In fact, the results from the CUDA and CPU DBT implementations have more differences than the single precision versions did. 17.7% of the values, or over 21 million pixels, are different between the results (Figure 18). The maximum differences in double precision were 20,000, which is identical to the single precision results.

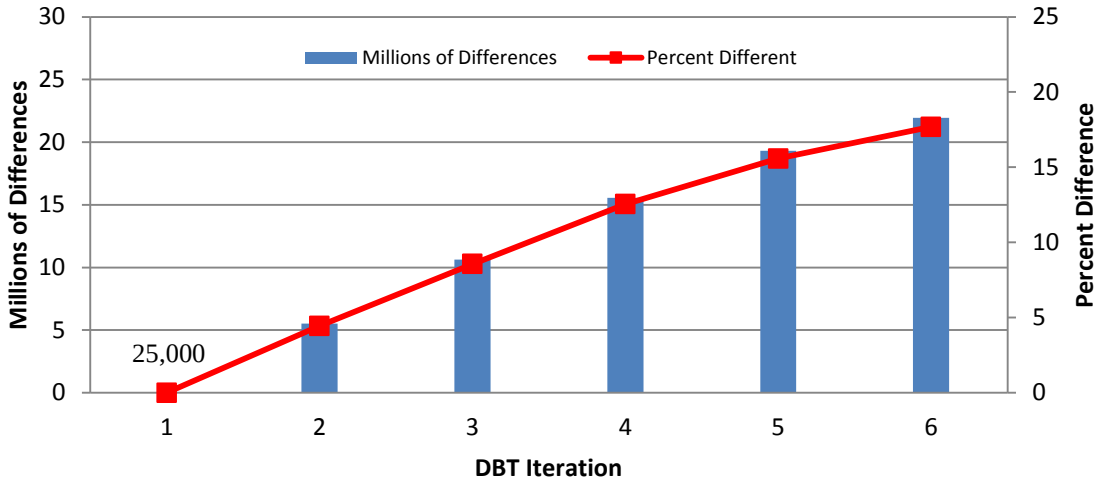

Figure 18: Differences per DBT iteration (double precision)

Forward projection is where we first locate the sources of floating-point difference in both single and double precision. The operations or instructions that cause differences appear during backward projection, which also causes, to a lesser degree, more divergence of the results.

Based on this information, it is clear that both the single and double precision algorithms must be investigated further to determine the root cause of the differences. The lack of map reduce techniques or any other reordering of computations makes it interesting and disconcerting to see this many differences. Slight differences may be acceptable, but having a concrete explanation of their cause is much more reassuring, especially in a critical application such as DBT. Also, the phantom data available for testing purposes could never anticipate every possible tissue makeup, which means that on real input data, the result could be a misdiagnosis.

3.2.3 Intermediate Testing of DBT

The next step was to test the intermediate results in order to determine the sources of differences. We joined the CUDA and C++ DBT codes by performing both the CPU and

GPU forward and backward projections consecutively in the same application. This made it easier to compare variables at various points throughout the DBT algorithm and makes debugging both CPU and GPU codes easier. The difficulty is that the computations by each thread in a kernel cannot be read or displayed until after the kernel completes and data that is never stored globally may be lost. To resolve this, test arrays were created in the GPU device memory for the data that we wanted to compare to the same point in the CPU code. We believe this combining effort could be completed on many codes that have versions that target both CPUs and GPUs. Doing so during application development and testing phases would give the programmer more access to differences that may be incurred, helping programmers to pinpoint and resolve them if feasible.

Once the codes were combined and numerical differences were found in the results, we determined what operation in the code was at fault and then researched possible reasons. Part of our research involved locating and verifying the actual instructions chosen by both the CPU and GPU compilers in the intermediate representations (IR). The IR code from both architectures more accurately represents the instructions being performed in hardware. The GPU IR is called PTX and can be output during compilation by using the "-keep" option of NVIDIA's nvcc compiler. The CPU IR is x86 assembly and can be output during compilation by use of the "-S" option of the g++ compiler. By understanding the expected behavior of CUDA and C++ code and verifying the implementation in the intermediate representation, we were able to precisely know where differences were occurring and discuss the accuracy implications of each.

Throughout this section, single precision is analyzed first and the behavior is described in detail. A description of the double precision behavior will follow in less

detail unless it behaves differently than the single precision version. To find the differences, each variable of the code was tested in the order it was executed and thus the differences are described in the order they appeared.

3.2.4 Tomosynthesis Difference Source #1 – Unsafe Casts

The drastic differences in the naïve GPU and CPU implementations were found to be due to a coding error that exists in the original CPU code. The error was that a floating-point value was converted to an unsigned short (16 bit) data type, which occurred twice in each of the CPU and GPU implementations. The conversion was done using a cast, as shown in Figure 19, which is undefined when converting from floating-point to integer in C++ and CUDA. This is a common coding error, especially in cases where a scientist writes the code rather than a trained programmer, and goes unnoticed because it will often provide the correct answer.

```
float          float_val   =  4.0423
unsigned int   integer_val =  (unsigned integer)  float_val;
                     integer_val result is 4
```

Figure 19: Generic unsafe casting example in C++

The reason that this type of conversion is unsafe is because if the floating-point value is negative or greater than the maximum representable integer the result is undefined and thus unpredictable. Suggested safe alternatives are float-to-integer functions, a set of known safe series of casts, or using conditional statements (if, else) to ensure the value is not negative and/or will not cause an overflow. These are typically defined in the programming language manual. Both C++ and CUDA have recommended procedures or functions to safely cast a floating-point value to an integer. Despite the warnings, many programmers use this unsafe cast when they believe their floating-point value will never be negative or cause overflow, or when they assume a certain result for

these special cases. For example, it might be expected that casting a negative value will be flushed to 0.

Since the casting mistake is explained in the programming manuals, it may be considered unnecessary to document. To understand the importance, consider this example. In 1996, Flight 501 of the Ariane 5 rocket was unsuccessful as the rocket exploded 37 seconds after take-off after 10 years and $370 million of development [36]. The disaster was caused by the automated self-destruct system, which contained a bug. The issue was that a 64-bit floating-point value was converted to a 16-bit integer, overflowed and caused an exception that triggered the explosion. This error is similar to the casting error result in DBT, which could similarly affect people as it may cause a misdiagnosis of a disease.

The error in DBT was specifically a negative floating-point value being cast to an integer. The offending line of code is shown in Figure 20. Surprisingly, it is the CPU code that produces an erroneous result; the CUDA code actually behaves as the programmer intended. The result from the C++ code showed that casting a negative number produced a large value instead of the more reasonable result of 0 that is returned by the CUDA implementation.

```
device_absorb[klm] = (unsigned short)(absorb_temp *
                (float)device_length[klm]*fLength);
```

Figure 20: Illegal cast in both CUDA and C++ DBT

The CPU version truncates -69.235 to a -69 signed short value (0xFFBB) and then interprets it as an unsigned value. The unsigned short value of 0xFFBB (65467) is the result the CPU reported. This issue presents itself in both single and double precision floating-point and both perform better on the GPU (Table 1).

Table 1: CPU and GPU results of casting a negative floating-point value

Cast	CPU	GPU	Preferred Result
(unsigned short) (float)-69.235	65467	0	0
(unsigned short) (double)-69.235	65467	0	0

The resolution of this problem is to rewrite the line of code in Figure 20 for each version using both C++ and CUDA recommended solutions. The CPU code should follow the rule for converting floating-point values in C described by Microsoft [37]. This rule is to first convert the floating-point value to a *long* and then to convert it to an *unsigned short*. In the GPU code, a CUDA function is available to safely convert the value to an unsigned integer. These changes are shown in Figure 21 and highlighted in blue. This series of operations will flush to 0 in the case that the floating-point value is negative.

```
CPU: device_absorb[klm] = (unsigned short)(long)
                (absorb_temp * (float)device_length[klm]*fLength);

GPU: device_absorb[klm] = (unsigned short)__float2uint_rn(
                (absorb_temp * (float)device_length[klm]*fLength)
```

Figure 21: CPU and GPU safe data conversion

The coding error also existed in another line of the DBT code in the backward projection and contributed to the total number of differences. The behavior was identical to the former error. The results from the code reported in the remainder of this thesis reflect both of these coding errors being resolved.

The floating-point implementation of the GPU CUDA application avoided the bug. Often, differences in floating-point computations are assumed to be the fault of the GPU when it may be due to a bug in the original code. In DBT, it was the CPU code that produced the wrong result and could have raised doubts about the GPU code's quality. If a certain input data had triggered many of these errors, the GPU would have produced

more accurate results (according to what the programmer desired) while also performing much faster and possibly have led to a better diagnosis. By joining the CPU and GPU DBT codes, we found an error in the original DBT application that went unnoticed otherwise.

In single precision tests, fixing the casting error removed 2973 differences, which is 0.002% of the final image. Since the 3D result is 45x1196x2304 pixels, this percentage emphasizes why these errors went unnoticed in the original CPU version. It would take many differences in a single area to actually notice a difference by eye. More importantly, the average and maximum differences show significant improvement after the correction (Figure 22 and Figure 23). The average difference is improved by 75% and 50% in the second and final DBT iterations respectively (note the logarithmic scale). The maximum difference is reduced from 20000 to about 6000. The first iteration has a maximum difference of 509, which shows that the maximum initial differences are compounded in later iterations. These results now appear to be more representative of expected floating-point differences which will be further discussed in the following sections.

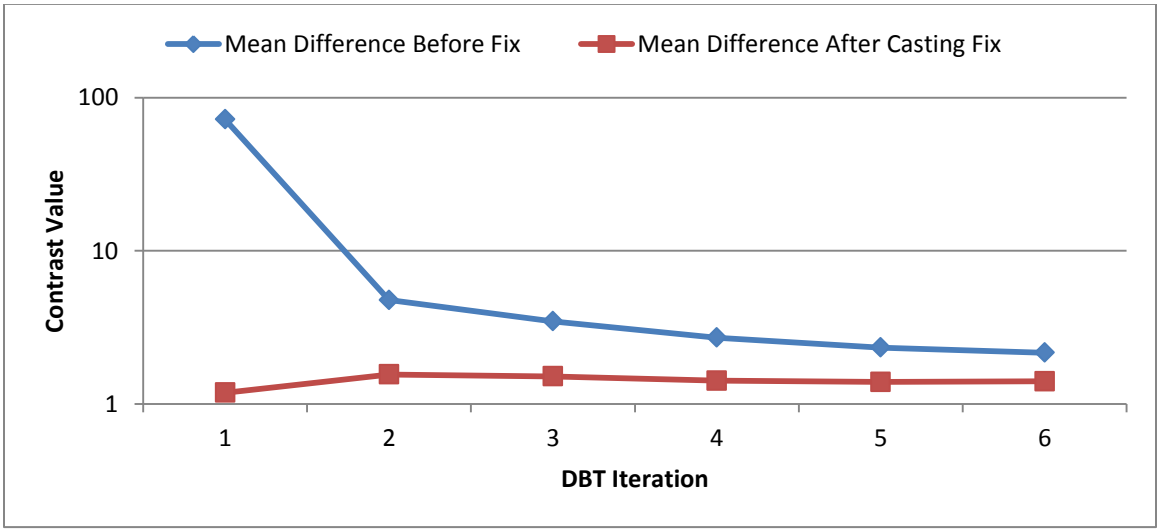

Figure 22: The effect on mean differences after the casting fix

In DBT and other GPU applications, the double precision implementation is equally as susceptible to this floating-point difference in the code. The casting mistake caused 8889 differences in the double precision code and they were repaired in a similar way, using the function *__double2uint()* as the conversion function. There were still more than 21.9 million differences remaining after this modification.

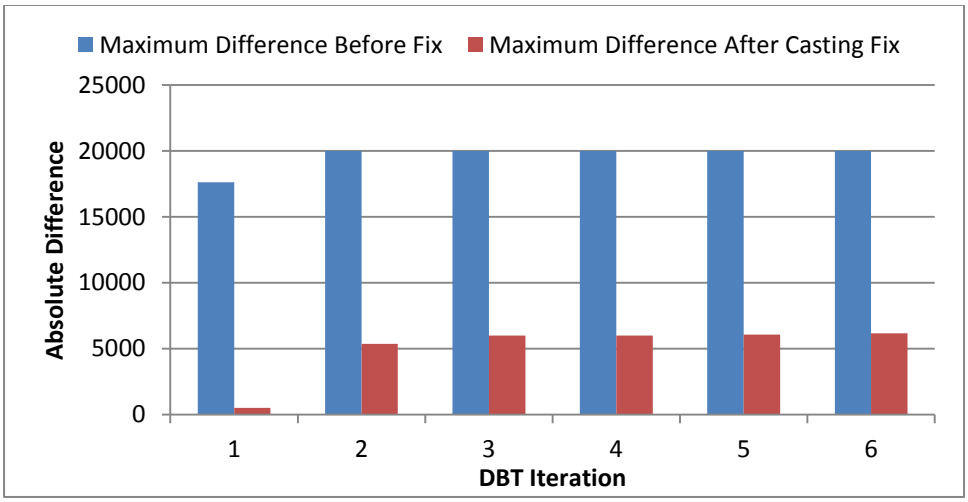

Figure 23: Casting fix affect on maximum difference (single precision)

Single and Double Precision Casting Difference – Conclusion

By fixing the casting issues in both the single and double precision codes, we have increased the accuracy of the CPU application and made the GPU application safer. Several thousand differences were removed from single and double precision codes by fixing two lines of code in each.

3.2.5 Tomosynthesis Difference Source #2 - Division

The next source of difference between the CPU and GPU code is found in a floating point division early in the DBT forward projection. The CPU version contains the identical statement written in C++ and furthermore, the code is completely valid in C++ and CUDA, unlike the the unsafe situation in the casting example. The obvious conclusion is that the GPU must be performing a different division operation than the CPU.

Division is an operation that deviates from the IEEE 754-2008 standard in CUDA by default for the compute 1.x devices. In fact, division was one of the most unsafe floating-point operations in the early GPUs tested in Hillesland's work [22]. The issue is described in the CUDA Programming Guide [14] in the following way:

> "Division is implemented via the reciprocal in a non-standard-compliant way."

This division is not IEEE compliant and is reported to produce a maximum error of 2 ULP (Units in Last Place. This error is small, as most floating-point differences are, but can have a large impact on an iterative algorithm like DBT where results influence the next iteration.

Single Precision

In order to decide the best action to resolve the differences, the actual instructions being used were verified in both codes: the assembly code of the gcc compiler and the Parallel Thread eXecution (PTX) from nvcc. The lines involving single precision division in PTX and in x86-64 assembly code are shown in Table 2.. The instruction used for division in the CPU IR (intermediate representation) is *divss* – the Divide Scalar Single Precision Floating-point Values instruction of the SSE instruction set [38]. This is an IEEE division that is used by default for single precision x86-64 code. The GPU uses the instruction *div.full.f32* and is defined in the CUDA PTX manual:

> "div.full.f32 implements a relatively fast, full range approximation that scales operands to achieve better accuracy, but is not fully IEEE 754 compliant and does not support rounding modifiers."

Table 2: Single Precision division instruction as it appears in CPU and GPU IR

Device	Instruction	Type of Division	IEEE
CPU	divss %xmm1, %xmm0;	Divide Scalar Single Precision Floating-point Values	✓
GPU	div.full.f32 %f3,%f2,%f1;	Full range approximate division that scales values to achieve better accuracy	✗

It should be mentioned that this definition is not the division that the CUDA PTX manual describes as a reciprocal division, which is a common faster alternative to the default IEEE division on x86 architectures. The reciprocal division in PTX would appear as *div.approx.f32* and also incurs at most 2 ULP error. NVIDIA does not provide any further information about the implementation of *div.full.f32* but we assume this division uses the same implementation as the reciprocal division but with the added accuracy of scaling the operands.

The CUDA Programming Guide offers a function that forces the division to produce an IEEE compliant instruction, *div.f32*. The function, *__fdiv_rn(x, y)*, with *x* and *y* being the two operands, must be used instead of the standard division operator ("/"). This will force the compiler to use the *div.f32* instruction - a fully IEEE compliant single precision division. There is no compiler option to force the GPU code to be IEEE compliant for compute 1.x devices so this function must be used in place of each division in the code. Fermi devices are compute 2.x and default to IEEE compliant division, which is discussed in Section 3.3.

To match the GPU and CPU division operations, each occurrence of division in the DBT GPU kernel is replaced with the *__fdiv_rn(x,y)* function. The *rn* suffix is used

because round-to-nearest is the default rounding mode of C++ IEEE division that we are trying to match. All *div.full.f32* instructions were now implemented as *div.f32* in the PTX after recompilation (Table 3).

Table 3: Single Precision division in IR after fix

Device	Instruction	Type of Division	IEEE
CPU	divss %xmm1, %xmm0;	Divide Scalar Single Precision Floating-point Values	✓
GPU	div.f32 %f3,%f2,%f1;	IEEE 754 Compliant Rounding Division	✓

With this correction, the overall GPU results are only slightly more similar to the CPU's, although the impact was greater than the casting fix. Nearly 50,000 differences (.04% of the entire image) have been removed from the final results after all 6 DBT iterations. The magnitude of the individual differences were less affected than they were by the casting fix, as the mean difference lowered to 1.403 contrast values – a nearly undetectable and visually imperceptible change of -0.002. The maximum difference remained unchanged (6184) but these large differences are rare.

Double Precision

The double precision code experiences a different behavior for division instructions. We confirmed the behavior by analyzing the CPU and GPU intermediate representations in the same way as we did for single precision. The GPU uses the *div.rn.*f64 (Table 4), which is the double precision version of the IEEE compliant division div.f32 we used in the single precision version.

Table 4: Double precision division instruction as it appears in CPU and GPU IR

Device	Instruction	Type of Division	IEEE
CPU	divsd %xmm1, %xmm0	Divide Scalar Double Precision Floating-point Values	✓
GPU	div.rn.f64 %fd3, %fd2, %fd1;	IEEE 754 Compliant Rounding Division	✓

The CUDA Programming Guide and PTX ISA (instruction set architecture) manual confirm that double precision division only has one mode, *div.rn.f64*, which is IEEE compliant. Unlike single precision IEEE division which is capable of all rounding modes, round-to-nearest-even (*rn)* is the only supported rounding mode in double precision on compute 1.x devices (like our Tesla C1060). This may trouble GPU codes if they are compared to CPU codes that use one of the other three IEEE compliant rounding modes. The reason that there are limited capabilities in double precision division is likely because double precision was a recent addition ISA, originating shortly after the GT200 series was released. Double precision is used when high precision is a requirement and displays relatively poor performance in comparison to single precision. The lack of focus on fast approximate instructions might be attributed to these requirements. The double precision GPU code does not require any changes to produce matching results to the CPU version. If a CPU application uses another rounding mode (not the default, round to nearest), it may not be possible to match its result in double precision. In that case, CPU code would have to be changed to match the GPU rounding-mode.

Single and Double Precision Division - Conclusion

In the single precision implementations, 50,000 corrections in the final image, resulting from using IEEE compliant divisions, significantly improves the accuracy of our application and produces results that are trending towards the CPU results. The double precision code was not improved at all by studying the division performance. Not only did we find a critical difference between single precision CPU and GPU codes in a real application, but we also explained how to increase the single precision accuracy of all GPU codes that use division and compute 1.x architectures. In the next chapter, we discuss the performance implication of this change.

3.2.6 Tomosynthesis Difference Source #3 – Multiply Add Combination

This third and final difference accounts for the remaining 13.5 million differences in the single precision code and the 22 million differences in double precision. The NVIDIA CUDA compiler aggressively combines lines that produce consecutive multiplication and addition instructions into single instructions using a multiply-add instruction (MAD in compute 1.x). This instruction is designed in hardware to complete the two instructions in nearly the time it would take for one. NVIDIA reports that 8 addition, multiplication or MAD instructions can be completed per clock cycle per multiprocessor. MAD instructions are the largest contributing factor to the differences in the CPU and GPU DBT codes.

Single Precision

The first instruction that was found in the DBT GPU code that experienced the effects of a MAD instruction involved a calculation of a pixel coordinate during ray tracing in the forward projection and is shown in Figure 24.

```
1   float Px = imgShiftX -
2                (startX + (float) i*device_stepX[klm]) *
3                reci_imgpixD;
```

Figure 24: DBT computation that results in FMAD

The CUDA Programming Guide states that programmers should expect single precision add and multiply instructions to be combined into an FMAD (floating-point multiply-add) instruction. To verify that the differences in DBT are due to this compiler feature, we analyze the PTX and x86-64 IR as we did in the divide example. Table 5 shows the instruction used for the multiply-add combinations in the IR.

Table 5: Single precision mad instruction as it appears in CPU and GPU IR

Device	Instruction	Description	IEEE
CPU	addss -168(%rbp), %xmm1 mulss -180(%rbp), %xmm1	Single Precision Floating-Point Addition and Multiplication	✓
GPU	mad.f32 %f24,%f23,%f1,%f19;	Single instruction, Multiply two values and add a third value	✗

The CUDA programming guide also warns that these instructions may cause a loss of precision:

> *"*Addition and multiplication are often combined into a single multiply-add instruction (FMAD), which truncates (i.e. without rounding) the intermediate mantissa of the multiplication."

The *mad.f32* instruction is not IEEE 754-2008 compliant because of the truncation of the mantissa between the multiplication and addition. Meanwhile, the CPU computes the multiplication and addition in succession using IEEE compliant instructions (Table 5). In order to protect against this combination and force IEEE compliance on the GPU we use the *__fadd_rn(x,y)* function provided by NVIDIA (Figure 25).

```
1   float Px = imgShiftX -
2                (__fadd_rn(startX, (float) i*device_stepX[klm])) *
3                reci_imgpixD;
```

Figure 25: Using __fadd_rn protects against FMAD

The PTX after the change is shown in Table 6. As described in the programming guide the instructions correctly split into separate, single precision, IEEE compliant multiplication and additions functions.

Table 6: Single precision mad is avoided using the __fadd_rn function

Device	Instruction	Description	IEEE
CPU	addss -168(%rbp), %xmm1 mulss -180(%rbp), %xmm1	Single Precision Floating-Point Addition and Multiplication	✓
GPU	mul.rn.f32 %f32,%f31,%f3; add.rn.f32 %f33,%f27,%f32;	Separate IEEE single precision add and multiply	✓

In all, twelve *mad.f32* instructions existed in the PTX assembly, which were located in the CUDA code during both forward and backward projection. They were all fixed by using the *__fadd_rn(x,y)* function. After executing the CPU and GPU code with the fix, all 13.5 million single precision differences in the resulting 3D image are removed.

Double Precision

Similar to division instructions, double precision behaves differently regarding MAD instructions. Double precision shows differences at the same lines of code but the MAD instructions do not have the same effects on the accuracy of the computation. The difference in the way double precision GPU code behaves is that the MAD created by combining a multplication and addition, *mad.f64*, is IEEE compliant (Table 7), which was not true of single precision MAD. The reason is explained in the PTX manual:

"mad.f64 computes the product of **a** and **b** to infinite precision and then adds **c** to this product, again in infinite precision. The resulting value is then rounded to double precision using the rounding mode specified by .rnd. Unlike **mad.f32**, the treatment of subnormal inputs and output follows IEEE 754 standard."

Table 7: Double precision FMAD

Device	Instruction	Description	IEEE
CPU	addsd -168(%rbp), %xmm1 mulsd -180(%rbp), %xmm1	Single Precision Floating-Point Addition and Multiplication	✓
GPU	mad.rn.f64 %fd24,%fd23,%fd1, %fd19;	IEEE compliant FMAD	✓

In previous examples, making the GPU code use IEEE compliant instructions would make the GPU results match the CPU results. In this case, difference exists despite using an IEEE compliant instruction. Because of the infinite precision in between the multiplication and addition, the double precision MAD is more accurate than separate multiplication and addition operations, which endure rounding at both instructions.

NVIDIA offers a similar function that forces the compiler to leave the multiplication and addition as separate instructions. This instruction is *__dadd_rn(x,y)* and uses the round-to-nearest rounding mode (Figure 26).

```
1   double Px = imgShiftX -
2               (__dadd_rn(startX, (float) i*device_stepX[klm])) *
3               reci_imgpixD;
```

Figure 26: __dadd_rn protects against a double precision FMAD optimization

This modification makes the double precision code less accurate in order to match the CPU code since the naïve implementation retains more accuracy than separate instructions. Because most of DBT's differences were caused by the MAD instructions, it is very likely that most of the 21 million differences were because of more accurate results in the GPU code. Verifying this assumption is part of future work.

Single and Double Precision MAD Opimizations - Conclusion

After modification, the single and double precision GPU and CPU codes produce completely identical final results. The identical result signals the completion of our effort to prove that the GPU DBT results can match the CPU's. We also prove that naïve GPU applications, that do not provide identical results, can be modified, sometimes relatively easily, to have a result of equal or even better accuracy than the CPU code. This effectively proves that the GPU DBT application is now as safe as the CPU version, at a faster execution speed. The actual effects of these modifications on the execution speed are described in the next chapter.

3.3 Fermi

NVIDIA has released a new generation of GPGPU architecture, which includes changes to the core layout and has an enhanced compute capability (compute 2.0). The floating-point performance in Fermi is greatly improved. The Fermi Tesla series GPUs have 16 double precision floating-point units per streaming multiprocessor consisting of 32 cores making it possible to achieve double to single precision performance of 1:2, instead of 1:8 as it was in the GT200 Tesla series.

There are several changes in floating-point operation accuracies from previous devices. First, MAD instructions are replaced with fused multiply-add (FMA, *fma.rnd.[f32,f64]*) in single and double precision. FMA instructions compute the MAD identically in both precision modes, retaining full accuracy between the multiplication and addition. This means that fast, hardware implemented, IEEE compliant multiply-add instructions can be completed in either precision, unlike the compute 1.3 device used in

our experiments. The FMA instructions would still need to be avoided in order to compare to CPU code until FMA is available in CPUs in the future. Furthermore, Fermi devices are more accurate than the CPU code in both precisions under naïve circumstances, which should further improve the GPU's reputation for accurate floating-point computation.

Division defaults to IEEE compliant modes in single and double precision in Fermi. Another improvement is that all rounding modes are available in both, which was not the case for compute 1.x devices. This may help create matching GPU and CPU applications that may not use default rounding modes.

3.4 Floating-point Difference Exploration Conclusions

In this section we:

- Described how and why floating-point's lack of associativity of reduction techniques on the GPU can increase the accuracy of the floating-point computation in comparison to serial CPU code.

- Described a data type conversion where the GPU code can have different results because of undefined behaviors, which was, in this case, due to a bug in the CPU code.

- Showed that GPU code is not limited in its capability of providing an IEEE compliant solution for division (without speed considerations).

- Displayed how multiply-add compiler optimizations can cause a large number of differences and explained how the compiler behaves differently for double and single precision codes.
    - We further described a process of observing the PTX and x86 IR codes to verify which instructions are used since they must be individually corrected in the original code.
- Proved that GPU code is not necessarily destined to have different results than a CPU based solution because of its hardware floating-point capabilities.
    - The GPU has several functions that are not, by default, IEEE compliant (compute 1.x) but CUDA provides functions that can force IEEE compliant functionality. This is a less known capability that can play a large role in gaining acceptance for use in the medical field.
- Described other differences that would be expected when compiling an application to be executed on the new, compute 2.x Fermi devices.

4 Execution Performance Results

In this chapter, the effects that the previous changes on the execution time have on DBT are discussed. The execution time of the algorithm is generally the most important goal of GPGPU. Relating the changes made in creating a CPU matching result to execution performance is important not only to DBT but to all GPU applications that may need similar modifications. DBT was originally written in CUDA so that it could be performed faster than previous versions for the CPU. This benefit would be diminished if the code is significantly slower after being modified to match the precision of the CPU. A slow GPU code that matches the CPU results may still be useful for verification purposes but if the code performs well in the more accurate versions, then GPU codes can be written accurately in more cases. Through our experiments, we show that the changes do not severely impact the performance of DBT and also explain whether this experience can be expected in other GPU applications.

The modifications from Chapter 3 are discussed in order from most significant to least and are compared to the naïve GPU application. All of our tests are executed on our test system (Section 2.5) at least 10 times and average times are reported.

4.1 CPU DBT Application Performance

The CPU application provided at the start of the project, in single precision, averaged 1789.16 seconds to execute the DBT application. The double precision application that we created from this completed in 1880.10 seconds – roughly 100 seconds longer. Double precision performance on the CPU is generally about 1:2 in comparison to single precision but DBT shows much less of a slowdown because it is a memory bound application, which hides most of the extra time to compute with the extra precision.

4.2 GPU DBT Application Performance

4.2.1 Naïve

Single Precision

The original naïve single precision GPU application does not implement any of the changes mentioned in Chapter 3 and executes DBT in 18.36 seconds, which is a speedup of 97x over the single precision CPU version. The CUDA application certainly satisfies its original purpose, to accelerate the CPU based DBT application. The speed of the GPU makes the use of DBT in a medical situation feasible if only execution time is considered.

Double Precision

The double precision CUDA application matches the naïve code except that it uses double precision data types where possible. The execution time in double precision is 46.15 seconds, about 40x faster than the double precision CPU application but 2.5x slower than the single precision GPU application. This 2.5x slowdown from a single precision to double precision GPU code is expected. Double precision codes naturally perform slower than single precision codes on compute 1.x devices because of the 1:8

ratio of double and single precision floating-point units on the GPU. This ratio does not usually apply directly to execution results because most applications are memory bound. Since the memory is the bottleneck in most GPU applications, extra or lower throughput operations can be used with less effect on the execution time than the raw performance statistics of the GPU hardware would suggest.

4.2.2 Multiply-Add Modification

MAD operations are the largest contributor to differences and execution time in the analysis of DBT. As described in Chapter 3, MAD instructions are created due to a compiler optimization that combines individual multiplications and additions into single, fast instructions. This is in contrast to the CPU that uses IEEE compliant multiplication and addition separately. In both single and double precision, it is expected that some performance degradation will occur be restricting the use of MADs.

Single Precision

After modifying the compiler behavior by using a function provided by CUDA, the GPU application was forced to use IEEE compliant addition and multiplication operations and matched the precision of the CPU output. After resolving over 13 million differences, the code executes in 18.62 seconds. This is only 0.26 seconds (0.014%) slower than the naïve version – a very miniscule slowdown.

The MAD instructions are executed many times per pixel, which makes the relatively insignificant slowdown surprising. Effectively, we increased the accuracy of our single precision code with almost no affect on the execution time. Like in the naïve single and double precision GPU comparison, the memory bottleneck hides most of the extra computation.

Double Precision

The double precision MAD instruction, *mad.rn.f64,* has the significant difference from the single precision instruction *mad.f32* that it is IEEE compliant. This makes the instruction more precise than the CPU default that uses two IEEE instructions and thus experiences rounding twice. Despite MAD's improved accuracy, it is still necessary to restrict the use of MAD instructions in order to match the CPU application that does not use them. After modifying the application to avoid MAD optimization the GPU implementation executes in 53.17 seconds, 7 seconds slower than the naïve double precision GPU implementation.

The 7 second slowdown is the only significant change in performance we experience in the modifications of DBT. The double precision GPU slowdown of 15% (compared to single precision) is not overly worrisome since it is still much faster than the double precision CPU code (33x) and matches the CPU allows verification of the application's function. The more noticeable slowdown does imply that caution should be used when restricting the compiler from using MAD instructions when execution time is a concern. The need for this modification in production code is diminished because the double precision MAD is IEEE compliant. Perhaps this modification should only be used in double precision to verify or debug code by comparing it to the CPU result. Then the modification could be removed for the final release of the GPU code with the understanding that any differences are due to more precise, faster computation.

Single and Double Precision MAD Performance Conclusions

These results show that while performance optimization is very important in GPU codes, it may be worthwhile for the developer to test various improvements to their codes

to see if more precise implementations are feasible while still achieving great execution time. This is especially the case for the GT200 series GPUs since they are not IEEE compliant when using single precision MAD. We have just shown that in DBT, forcing the application to match the CPU results may be possible with only a small affect on execution time. This has a high likelihood to also be the case in other GPU applications as they are usually memory bound.

4.2.3 Division Modification

The second difference in DBT was the division operation. Division is one of the most feared mathematical operations in terms of execution time, whether it is performed in software or hardware. It can sometimes require 10x more cycles to complete when compared to an addition or multiplication on the same architecture, and an IEEE compliant division is generally even slower. The difference in execution time of different division modes on the GPU is large at the individual instruction level. The slowdowns may only have a minor impact when part of a memory bound application. Division operations turn out to have little effect on execution time of DBT for these reasons, as was the case with MAD instructions in the previous section.

Single Precision

Earlier in this thesis we established that CUDA compiles the division operation into a full range, approximate division that is not IEEE compliant. After modifying the operation so that an IEEE compliant division is used, we expected an equal or slightly worse execution time in a memory bound application like DBT. After modifying the naïve version to use IEEE compliant division, over 50,000 differences were removed and the execution time was 18.28 seconds; a surprising 80 millisecond improvement. To alleviate our concerns that this was a testing error, we tested 100 executions and achieved the same

result with a standard deviation of only 5 milliseconds. We accepted this to be an accurate reading and assume that because there are loop counters that depend on the division result in DBT, there are variations in the number of iterations the loops run and the runtime is affected. Further investigation of this issue is future work.

To test all relevant situations that may relate to other GPU applications, the faster, approximate division was also analyzed. Using this instruction, *div.approx.f32*, the GPU DBT implementation executed in 18.60, which is 240 milliseconds slower than the naïve version. This is again surprising but can also be explained by the loop counter dependence on the division result.

Double Precision

In double precision there is no approximate division operation and thus there is no comparison to discuss. The naïve GPU application performance of 46.15 seconds is the execution time using the double precision IEEE compliant division in DBT.

Division Performance Conclusions

The DBT application demonstrate that the expectations that more accurate division instructions cause slowdowns in execution time are not always valid. Programmers should always attempt to use the most accurate division instructions in their code and then test faster approximate instructions where they can accept a loss of precision. In section 4.4.1 we will describe more generally the performance of all divison operations available on the GPU and how they may effect other GPU applications.

4.2.4 Casting Modification

The casting error was diagnosed as a coding mistake that should not exist in either CPU or GPU codes. Since it is a change of the original, accelerated DBT implementation and

because the problem may exist in other GPU codes, it is still sensible to be sure that using the correct casting procedure in both C++ and CUDA does not negatively effect execution times.

Single Precision

The execution time of DBT after the casting modification is 18.36 seconds, indicating no change from the naive GPU performance. The cast was performed at least once in both forward and backward projections for each of the millions of pixel computations. We assume that if there was a significant difference in execution speeds of the casting functions it would be reflected in the test. In future work, we will investigate whether the C/C++ accepted way of casting floating-point values is safe for the GPU as well, which would allow codes for CPU and GPU to be written the same way.

Double Precision

Unlike in single precision where no speedup or slowdown was experienced, the double precision fix showed a speedup of 120 milliseconds, executing in 46.03 seconds. There is no detailed description of the hardware implementation of the casting function available so the reasons for this result cannot be determined. We did not focus on this issue because the correct choice is always to perform the casting in the recommended way.

4.3 DBT Performance Overview

The modifications only produced minor changes in execution time (Figure 27). The naïve single precision performance was 18.36 seconds and improved slightly with the IEEE division implementation and finally lost nearly a quarter of a second when restricting MAD optimizations. In double precision, there was no division fix but the MAD caused a more significant 7 second increase.

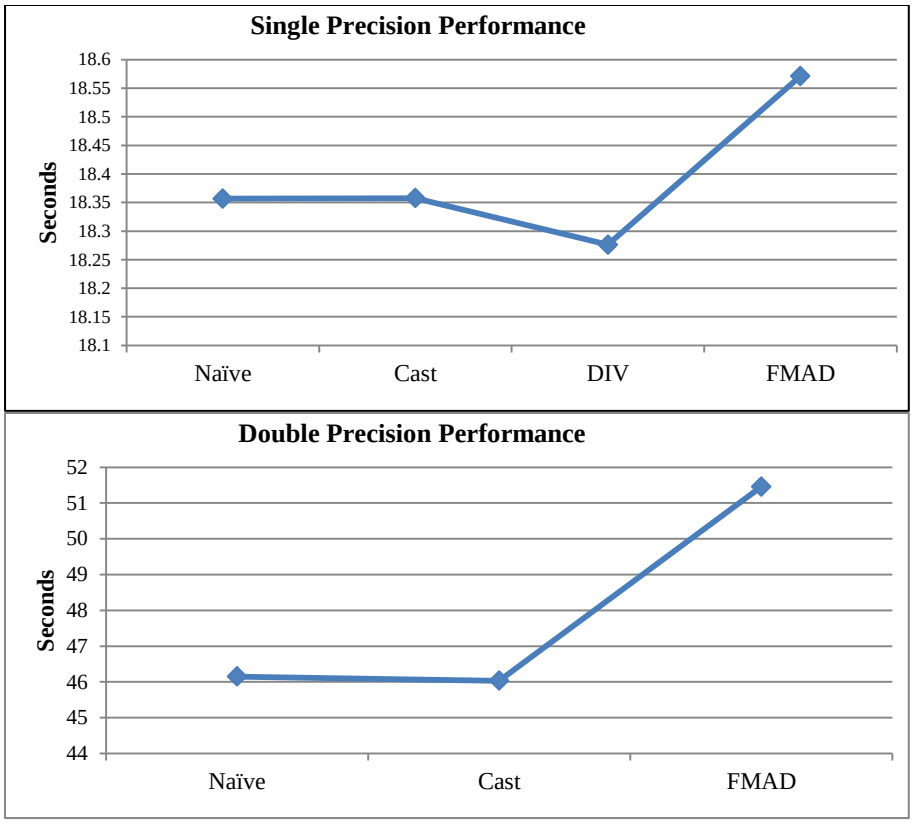

Figure 27: Single and Double Precision execution time performance

4.4 Detailed CPU and GPU Instruction Performance

Our tests thus far show the performance of MAD and division modifications only on the GPU DBT application. These tests may not accurately represent the performance of these instruction changes as it compares to other GPU applications. For this reason, we describe in this section several details of the execution time of division and MAD optimizations. The results give a better understanding of the possible performance effects of each.

4.4.1 Division

It was peculiar that the IEEE compliant division improved the time in the DBT application noticeably in single precision. There are very few divisions used in DBT and thus a small difference should be expected, but it would be expected to be slower rather

than faster. These tests compare the performances of the commonly used division instructions on the CPU and GPU. These include:

- **GPU**
  - *div.approx.f32*: Single precision division is performed via the reciprocal and a multiplication (not IEEE compliant)
  - *div.full.f32*: Approximate single precision division with operands scaled to retain precision (default and not IEEE compliant)
  - *div.f32*: Single precision IEEE compliant division
  - *div.rn.f64*: Double precision IEEE compliant division
- **CPU**
  - *divs[s,d]*: Single or double precision IEEE compliant division (default)

To test these instructions, a program was created to capture only the CUDA and x86-64 floating-point division performance. It consists of 1000 floating-point divisions on operands in registers (to avoid latency and bandwidth issues) on over 4 million threads to ensure high device utilization and to force the computation to dominate the execution time. We were not able to find any documentation or previous work that compared the division instructions explicitly in this manner. We report end to end application execution times (Figure 28) for each instruction. These execution times cannot be extrapolated to single instruction times because there are other instructions and data transfers involved, such as conditionals to control the loop. Loop unrolling was utilized to minimize these extra instructions.

This experiment confirms that the higher precision (IEEE compliant) divisions are much slower (note the logarithmic scale) than the approximate versions. The fastest, approximate division test takes 54ms. This is followed by 84ms for the default, slightly more precise, approximate division (nearly 50% slowdown). The IEEE version is over 10x slower at 962ms, because it is performed in software (which we verified in the PTX).

Double precision division in CUDA is always IEEE compliant and is only about 2x slower (1854ms) than single precision IEEE division. Since there is only 1 double precision FPU for every 8 single precision FPUs, the slower performance is expected. It is not a factor of 8 because of the many instructions used to execute IEEE single precision division in software.

The x86-64 IEEE compliant division instructions take 16 and 25 seconds respectively to complete the test in single and double precision. This equates to over 13x speedup for the GPU even in the compliant versions.

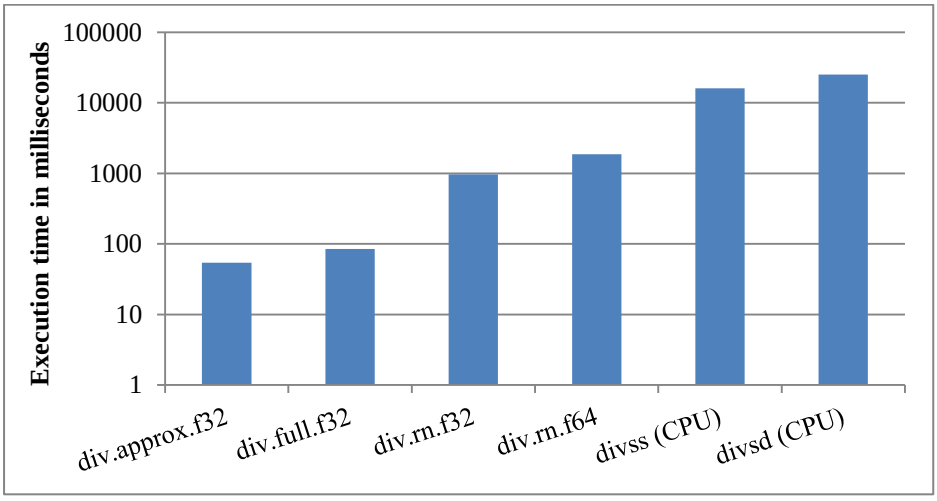

Figure 28: Execution times of CPU and GPU division implementations

These results do not explain the original issue; the single precision IEEE division version of DBT is faster than the naïve application that uses the approximate division.

Discovering the underlying reasons for this is part of future work. Beyond the possibility that loop counters are being affected by the results, it is also possible that the CUDA compiler optimizes the code better in this particular case when the different divide operation is used.

The DBT example and this more focused division test illustrate that understanding the floating-point division implementation behind the CUDA code is important in order to decide on the best implementation for a particular application. Even when the programmer understands all the floating-point implications, it may still be necessary to test various solutions to make an intelligent choice between the fastest and the most accurate result.

4.4.2 MAD

Timing analysis was also performed on CUDA MAD instructions in single and double precision. This section shows the results of testing the following versions of the CPU and GPU code:

- **GPU**
  - *mad.f32*: Single precision combined multiplication and addition (default, not IEEE compliant)
  - *mul.rn.f32 & add.rn.f32*: IEEE compliant single precision multiplication and addition
  - *mad.f64*: Double precision combined multiplication and addition (IEEE compliant)
  - *mul.rn.f64 & add.rn.f64*: Double precision IEEE compliant multiplication and addition

- **CPU**
  - *mulss & divss*: IEEE compliant multiplication and addition in single precision
  - *mulsd & divsd*: IEEE compliant multiplication and addition in double precision

The test code used is similar to the focused division test except that it executes 10,000 of the instructions (instead of 1,000) on each value in order to ensure that kernel overhead and data transfer do not significantly contribute to the execution time. This is necessary because multiplication and addition operations are much faster on the GPU than division. The applications operate on the data in registers to rule out as many bandwidth and latency issues as possible and the execution time of the full test is reported (Figure 29).

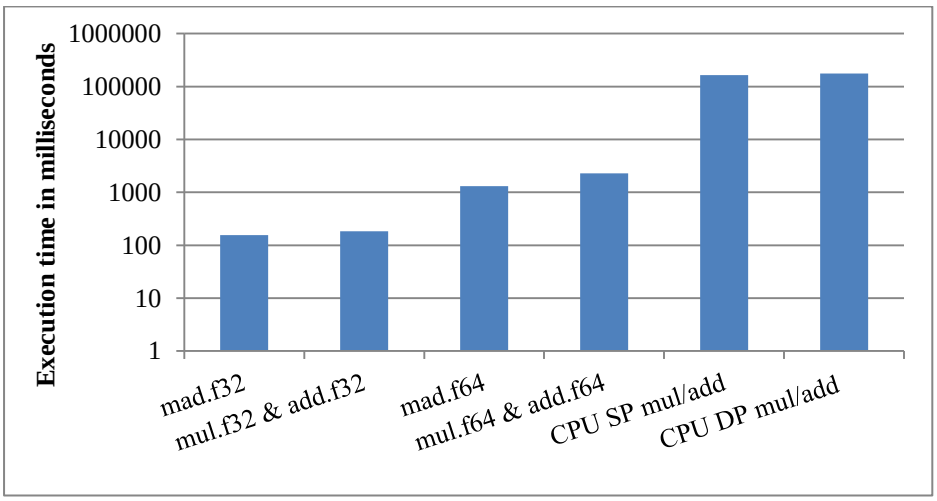

Figure 29: Execution times of CPU and GPU MAD implementations

The results demonstrate slowdown when MAD is not used and when going from single to double precision, as expected. Single precision CUDA instructions (160ms for MAD and 182ms for separated) were about 8x faster than double precision (1299ms and

2380ms). The x86-64 instructions in both precisions were hundreds of times slower (taking 70-180 seconds).

This execution time study shows first that the difference between splitting the MAD in single precision (not IEEE compliant) into separate addition and multiplication (IEEE compliant) may only add approximately 10% to execution time in single precision but allows the programmer to match results to an x86-64 CPU solution and retain better precision in the program by complying with IEEE standards. Similar to the division problem, the MAD performance cost in DBT was small. This further emphasizes the fact that if the programmer is aware of the consequences of the floating-point instructions in their CUDA code and properly tests their performance, better accuracy may be feasible on the GPU while retaining execution speed.

4.5 Conclusions

In this chapter, that discusses the effects of these floating-point differences and modifications on execution time, we learned:

- The casting, division and MAD modifications that allow the GPU results to match the CPU's can all be applied to DBT without large negative effects on execution time. This demonstrates that you do not always have to sacrifice execution performance to achieve IEEE compliance or more accurate code.

- The IEEE compliant single precision division instruction is approximately ten times slower than the approximate divisions, yet some applications can utilize it with little or no negative influence on execution time.

- Although it is useful to avoid MAD instructions when comparing codes, it may be the case that the performance of compute (rather than memory) bound applications will be worse if the MAD is not allowed.

5 SELECTING THE BEST GPU APPLICATION

To this point, the discovery, accuracy and execution performance have been described for each difference between the DBT CPU and GPU applications. In this chapter, we use all of this information to build the best DBT applications. The term “best” is a very general term in computing, especially in acceleration projects. Is the best code the most accurate, the fastest or the one that matches the CPU version that the code was based on? The answer to this question is highly dependent on the application, the use and the execution time requirements. In any application, there must be an understanding of what causes the differences in the chosen performance metrics in order to make it possible to create a solution that achieves them. This chapter describes several versions of DBT that could be considered “best” under different requirements. The four versions of DBT that are discussed are:

1. Naïve
2. CPU matching
3. Fastest executing
4. Most accurate floating-point computation

5.1 Naïve

The naïve implementation of the code is not necessarily the best solution, especially for safety critical code. Only codes that are written with an awareness of the codes' computational accuracies can be considered the best solution for their particular problem. Especially in codes where accuracy is paramount, the programmer should not depend on NVIDIA's implementation decisions which may not have specifically focused on accuracy.

5.1.1 Single Precision

We determined that the naïve single precision implementation of the DBT application had three issues in its floating-point computations. These include the division operator that is not IEEE compliant, the FMAD optimization that is also not IEEE compliant and a coding mistake involving an improper casting procedure. With these not accounted for, this solution is not the best in any situation.

5.1.2 Double Precision

The naïve double precision DBT application, despite having the same incorrect procedure for casting, does not have the issues of using less accurate instructions. The MAD and division operator used by default are IEEE compliant. This is a preferable situation in most codes. For debugging, we desired a CPU matching solution, which was not achieved in the naïve solution because of the different implementation of FMAD instructions on GPU and CPU.

5.2 Matching the CPU

Chapter 3 succeeded in creating a CUDA DBT implementation that produces matching results to the CPU. This GPU DBT implementation was the best solution for debugging the DBT code.

5.2.1 Single Precision

In the single precision GPU DBT code, it was necessary to use the CUDA function *__fadd_rn()* surrounding our floating-point addition operators where they were being combined with multiplications into a single FMAD. The FMAD is not IEEE compliant and the compiler optimization is the default case. The code also required the *__fdiv_rn()* CUDA function to force the division to match the CPU's default IEEE compliant division. The best solution for this and other single precision codes that perform these floating-point operations is to force the IEEE division and to avoid FMAD instructions. Along with the correct casting fix, this knowledge allowed us to create a CPU matching single precision version on the GPU.

5.2.2 Double Precision

The double precision GPU code only required one fix other than the casting error. The fix was to avoid the FMAD instruction despite it being IEEE compliant and possibly more precise then the CPU code. The CPU compiler creates a separate IEEE compliant multiplication and addition, as in single precision, which needs to be matched by the GPU. After debugging, the programmer may wish to make use of the FMAD instruction with the knowledge that greater accuracy than the CPU version is being achieved.

5.3 Fastest

The most commonly desired solution in GPU programming is the fastest executing version. Many codes are not nearly as dependent on accuracy as our DBT application. Despite NVIDIA's decision to use fast operations while sacrificing precision as the default CUDA behaviors, the naïve implementation may not give the best solution in terms of execution time. Changes are necessary to provide the fastest DBT solution, which may differ in other algorithms. Any application should be tested in all possible implementations of the floating-point computations that provide acceptable accuracy if speed is the focus.

5.3.1 Single Precision

In the single precision form, the GPU DBT code requires the division operator to be the IEEE compliant instruction to increase the performance. This was an interesting behavior that would likely not be expected by the programmer, further stressing the need for knowledge and testing of such floating-point implementations in all GPU applications. Unlike in the most accurate version, the fastest version must use the FMAD instruction as this is optimized specifically for speed.

The best solution for single precision GPU code in terms of execution time, at 18.28 seconds (Figure 27), was the code using the non-IEEE FMAD optimization (default) and the IEEE compliant division (via CUDA functions) along with the required casting fix.

5.3.2 Double Precision

The double precision DBT code has fewer options since there is only one version of divide available (which is IEEE compliant). The FMAD instruction is the only choice,

since the casting operation is required for correctness. Double precision codes generally have less of an emphasis on speed than they do on precision since the GPU performs double precision much slower than single precision. Allowing the FMAD optimization to be performed is the best choice in order to achieve the fastest computation in double precision. Codes that must be matched to the CPU for verification should consider reverting back to the default FMAD optimization since in double precision it is IEEE compliant, if speed is important. The fastest solution for the double precision DBT code executes in 46.03 seconds (Figure 27).

5.4 Most Accurate

The goal of the most accurate GPU code possible is an extension to the concept of our work on DBT, where we attempted to match the CPU code. The CPU code uses IEEE compliant floating-point computation. This is not necessarily the same thing as the most accurate version. In the DBT example the MAD optimization in double precision is the most accurate.

5.4.1 Single Precision

The most accurate DBT implementation incorporates the concepts from the previous chapters. In Section 3.2.4 the unsafe casting situation was described, where the GPU provided accurate results and the CPU did not. In Section 3.2.5, it was shown that the single precision division operation can be forced to be IEEE compliant, removing inaccuracies that would be found in the naïve implementation. Finally in Section 3.2.6, the possibility that removing inaccurate single precision FMAD instructions and replace them with IEEE addition and multiplication instructions was described.

Applications other than DBT will exhibit similar results if they are forced to use the IEEE instructions available. Additionally, in Section 3.1.1 we described that reduction routines will generally result in a more accurate solution in its naïve GPU implementation without the programmer's awareness.

5.4.2 Double Precision

Double precision is used in GPU codes that require accurate computation. In the work with DBT, the most accurate version is achieved in the naïve version. The MAD instruction in double precision, unlike single precision, is IEEE compliant and actually does not suffer rounding between the multiplication and addition that are fused together. This makes it more accurate than the CPU version while also being faster than not using FMAD. Programmers who understand these behaviors can rightly argue that the differences in results appear due to using the most accurate floating-point operations in double precision GPU code.

5.5 Conclusions

In this chapter, we explored several important concepts that are important to the CPU and GPU DBT, as well as their applicability to many GPU applications.

- A naïve DBT GPU implementation in single precision requires both division and FMAD considerations in order to ensure that only IEEE compliant operations are performed.
- A naïve DBT GPU implementation in double precision only needs to be written in correct CUDA (no casting errors or other undefined behaviors) in order to provide a very accurate solution. In fact, the naïve solution may be more accurate because of its high accuracy IEEE compliant functionality.

- The CPU and GPU code can be forced to have similar results by forcing the GPU to use IEEE compliant division in single precision and splitting the FMAD instruction in both single and double precision.
- The fastest code uses FMADs and IEEE compliant division in single precision and double precision. The difference is that in double precision, the IEEE compliant division is by default, where in single precision it is not.
- The premise that the GPU code is not accurate, cannot match the CPU, or is not IEEE compliant is sometimes an incorrect assumption, as displayed by our ability to create matching CPU and GPU DBT applications.

6 CONCLUSIONS AND FUTURE WORK

Our analysis of CUDA applications' performances regarding floating-point accuracy has brought to light several differences that can be predicted and avoided in many cases. By understanding the differences in this detail, the CUDA programmer can more effectively debug and ensure the accuracy of their codes while retaining execution performance. This will instill greater confidence in the application end-user about the quality of the CUDA application.

In an introductory π example we showed how map reduce operations may cause a different result because of the order of operations and also from the way the GPU bins the results during the computation. In order to match an application to the GPU in this situation, it is necessary to bin the results on the CPU in a similar or identical fashion or to serialize the GPU code (not recommended).

In Chapter 3, we demonstrated through our DBT case study a process of difference discovery and proved that with rather simple changes to the GPU code, identical results to an x86-64 CPU DBT application can be produced. The modified GPU code now uses only IEEE compliant instructions, matches the x86-64 Intel CPU implementation results and suffers a modest 1.2% slowdown from the extremely fast naïve GPU. Additionally, a double precision GPU DBT application with 30x speedup

over the CPU version was created for the first time and can now be used in the situation where higher precision is desired.

Chapter 4 provided information about how DBT execution time was affected based on the modifications to make it match the CPU implementation output. We were able to show that it is possible that some implementations may suffer little or not at all from the accuracy modifications like using IEEE compliant division or removing MAD optimizations. This leads to the conclusion that most applications should have some form of floating-point analysis completed before being used for their intended purpose. We also provided more information regarding division and MAD execution time performance that should help a programmer make the correct decisions throughout code development.

Lastly, Chapter 5 described how the modifications to floating-point computation would affect DBT if it were being optimized for various metrics, such as accuracy, execution time and matching a CPU code. This information can help a programmer understand how modifications can affect different codes in different ways. This understanding will allow them to make better floating-point computation decisions based on the goals of their application and ultimately produce a higher quality result.

The information and insight provided throughout this thesis will help correct the negative reputation that GPUs have endured throughout their participation in scientific computing. Programmers can provide higher precision code, explain differences that arise and even argue at times that the GPU accuracy is superior to the serial code from which it was derived. Adoption of this style of thinking in programming will benefit the GPGPU

community; one that often requires rapid, accurate results from applications and a high level of trust from their users concerning the quality of the results.

6.1 Future Work

In this work, the issues of floating-point associativity, division, MAD optimizations and type conversion in CPU and GPU codes were discussed. These are certainly not the only differences that may exist in floating-point computation in CUDA code. There may be other differences that may or may not be avoidable such as square-root and trigonometry functions that are also reported as having greater ULP error than an IEEE compliant solution. We hope to examine other example programs and also explore all instructions that are available to the CUDA compiler to obtain a complete collection of the differences, resolutions and implications on accuracy and performance.

In regards to the DBT application, we plan to further investigate the performance findings in the case of division, where our application experienced slight speedup when using a slower, more accurate operation. We also wish to perform the entire experiment on a Fermi device to determine the performance and verify accuracy in comparison to what is stated in NVIDIA documentation

We also wish to further investigate the conversion of floating-point values to integers. There are several ways to achieve number conversions correctly. For instance, instead of using the CUDA supplied functions it may be possible to perform a faster check using a conditional to avoid improper values reaching the cast.

Eventually, we would like to create guide for CUDA (and potentially OpenCL) programmers that describes the floating-point implications of the GPU's programming

languages, compilers and architectures in order to increase the quality of other GPU applications.

REFERENCES

[1] Daintith, J. *Floating-point operation*. Oxford University Press. A Dictionary of Computing. 2004.

[2] Hall, B. E. 2002. *Scientific Notation*. Mathematics, Encyclopedia.com. Retrieved March 20, 2011, from http://www.encyclopedia.com/topic/scientific_notation.aspx.

[3] IEEE Standard for Floating-Point Arithmetic. 2008. *IEEE Std 754-2008.*1-58. DOI=10.1109/IEEESTD.2008.4610935.

[4] Vickery, D. C. 2008. *IEEE-754 Calculators.* Queens College, NY. Retrieved March 29, 2011, from http://babbage.cs.qc.edu/IEEE-754/.

[5] Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5-48. DOI=http://0-doi.acm.org.ilsprod.lib.neu.edu/10.1145/103162.103163.

[6] Lopez, G., Taufer, M. and Teller, P. Evaluation of IEEE 754 floating-point arithmetic compliance across a wide range of heterogeneous computers. In *Proceedings of the Richard Tapia Celebration of Diversity in Computing Conference,* (Orlando, Florida, October 2007). DOI=http://dx.doi.org/10.1145/1347787.1347793.

[7] 2008. *File:Float example.svg.* Wikipedia. Retrieved March 25, 2011, from http://en.wikipedia.org/wiki/File:Float_example.svg.

[8] 2008. *File: IEEE 754 Double Floating Point Format.* Wikipedia. Retrieved March 20, 2011, from http://en.wikipedia.org/wiki/File:IEEE_754_Double_Floating_Point_Format.svg,

[9] Randelshofer. W. 2011. Real Numbers and Floating Point Formats. *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. 4.8, 4-15. Retrieved April 15, 2011, from http://www.intel.com/assets/pdf/manual/253665.pdf.

[10] Lea D. 1992. *User's Guide to the GNU C++ Library.* Version 2.0. Retrieved March 20, 2011, from http://www.gnu.org/s/libc/manual/.

[11] Kathiara, J. 2011. *The Unified Floating Point Vector Co-processor for Reconfigurable Hardware*. Master of Science Thesis, Electrical and Computer Engineering, Northeastern University. Boston, MA.

[12] Moore, G. E. 1965. Cramming more components onto integrated circuits. *Electronics Magazine*. 4. Retrieved March 20, 2011, from ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.

[13] A. Munshi. 2010. The OpenCL Specification v1.1. *Technical report, Khronos OpenCL Working Group,* Retrieved March 28, 2011, from http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[14] NVIDIA. 2011. *NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 3.2*. NVIDIA, 2011.
[15] K. Karimi, N. G. Dickson, and F. Hamze, "A performance comparison of CUDA and OpenCL," in *CoRR*, 2010, abs/1005.2581.
[16] NVIDIA. 2010. *PTX: Parallel Thread Execution ISA Version 2.2*. NVIDIA, 2011.
[17] Angelini. 2008. *NVIDIA Editor's Day: The Re-Introduction of Tesla*. Hot Hardware. Retrieved March 20, 2011, from http://hothardware.com/Articles/NVIDIA_Editors_Day_The_ReIntroduction_of_Tesla/.
[18] Le Cam, L. 1990. Maximum likelihood - an introduction. *ISI Review* 58, 2, 153-171.
[19] Schaa, D., Brown, B., Jang, B., Mistry, P., Dominguez, R., Kaeli, D., Moore, R., Kopans, D. B. 2011. GPU Acceleration of Iterative Digital Breast Tomosynthesis. *GPU Computing Gems*, Morgan Kaufmann, Boston, 647-657. DOI: 10.1016/B978-0-12-384988-5.00040-1.
[20] Wu T., et al. 2003. Tomographic mammography using a limited number of low-dose cone-beam projection images, *Medical Physics*. 30, 356– 380. DOI=http://dx.doi.org/10.1118/1.1543934.
[21] Meredith, J., Gonzalo, A., Maier, T., Schulthess, T. and Vetter, J. 2009. Accuracy and performance of graphics processors: A Quantum monte carlo application case study. *Parallel Computing*, 35, 3 (March, 2009). DOI=http://dx.doi.org/10.1016/j.parco.2008.12.004.
[22] Hillesland, K. and Lastra, A. 2004.GPU floating-point paranoia. In *ACM Workshop on General Purpose Computing on Graphics Processors*, C8, (August, 2004).
[23] Ruijters, D., Haar Romeny, B. M. and Suetens, P. 2008. Accuracy of GPU-based B-spline evaluation. In *Proceedings Tenth IASTED International Conference on Computer Graphics and Imaging (CGIM '08)*, ACTA Press, Anaheim, CA, 117-122.
[24] Bui, P. and Brockman, J. 2009. Performance analysis of accelerated image registration using GPGPU. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (GPGPU-2). ACM, New York, NY, 38-45. DOI=http://doi.acm.org/10.1145/1513895.1513900.
[25] Lu, M., Bingsheng, H., Luo, Q. 2010. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware* (DaMoN '10), Anastasia Ailamaki and Peter A. Boncz (Eds.). ACM, New York, NY, USA, 19-26. DOI=http://doi.acm.org/10.1145/1869389.1869392
[26] Diamos, G. F., Kerr, A.R., Yalamanchili, S. and Clark, N. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (PACT '10). ACM, New York, NY, USA, 353-364. DOI=http://0-doi.acm.org.ilsprod.lib.neu.edu/10.1145/1854273.1854318.
[27] Gough B., and Stallman, R. *An Introduction to GCC*. Network Theory, Ltd. Retrieved March 24, 2011, from

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.264&rep=rep1&type=pdf.

[28] Weisstein, E. Pi. In *MathWorld- A Wolfram Web Resource.* Retrieved March 14, 2011, from http://mathworld.wolfram.com/Pi.html.

[29] Mattson, T. and Eigenmann, T. 1999. OpenMP: An API for writing portable SMP application software. In *Proceedings of the SuperComputing 99 Conference*, (November, 1999).

[30] M. harris, "NVIDIA: Optimizing Parallel Reduction in CUDA," ed, 2008.

[31] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Common. ACM* 51, 1 (January 2008), 107-113. DOI=http://doi.acm.org/10.1145/1327452.1327492.

[32] Chandra, R., Menon, R., Daqum, L., Kohr, D., Maydan, D. and Mcdonald, J. 2000. *Parallel Programming in OpenMP*. Morgan Kaufmann. San Francisco, CA, USA. ISBN-13= 978-1558606715

[33] Gutierrez, E., Plata, O. and Zapata, E. 2001. Improving parallel irregular reductions using partial array expansion. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)* (Supercomputing '01). ACM, New York, NY, USA, 38-38. DOI=http://0-doi.acm.org.ilsprod.lib.neu.edu/10.1145/582034.582072.

[34] Ravi, V. T., Ma, W., Chiu, D. and Agrawal, G. 2010. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing* (ICS '10). ACM, New York, NY, USA, 137-146. DOI=http://0-doi.acm.org.ilsprod.lib.neu.edu/10.1145/1810085.1810106.

[35] Garzaran, M. J., Prvulovic, M., Zhangy, Y., Torrellas, J., Julia, A., Yu, H. and Rauchwerger, L. 2001. Architectural Support for Parallel Reductions in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques* (PACT '01). IEEE Computer Society, Washington, DC, USA, 243.

[36] Ben-Ari, M. 2001. The bug that destroyed a rocket. *SIGCSE Bull.* 33,2 (June 2001), 58-59. DOI=http://doi.acm.org/10.1145/571922.571958.

[37] Microsoft. 2005. Conversions from floating-point types. *MSDN.* Retrieved March 20, 2011, from http://msdn.microsoft.com/en-us/library/d3d6fhea(v=vs.80).aspx.

[38] Intel. 2011. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Retrieved March 10, 2011, from http://www.intel.com/Assets/PDF/manual/252046.pdf.