Implementing Fast MRI Gridding on GPUs via CUDA

Anthony Gregerson
University of Wisconsin – Madison

agregerson@wisc.edu

Abstract

Modern graphics processing units (GPUs) have made high-performance SIMD designs available to consumers at commodity prices. This has made them an attractive platform for parallel applications, however developing efficient general-purpose code for graphics-optimized architectures has proven challenging. To explore the challenges and opportunities of exploiting general-purpose GPU processing, we implemented the non-equispaced Fast-Fourier Transform algorithm, commonly known as 'gridding', on a Geforce 8800 GPU using Nvidia's CUDA framework. Our results found that optimizations in thread scheduling, data structures, and memory access patterns could accelerate a naïve GPU implementation by over 400%. Our optimized GPU implementation showed a 114X speedup of the convolution-interpolation kernel compared to the same kernel running on an Intel Core 2 Duo CPU when running on a Geforce 8800GTX and a 17.5X speedup when running on a Geforce 8400GS.

1. Overview

Modern medical imaging techniques, such as magnetic resonance imaging (MRI) are computationally-rich image manipulation processes. Recently, functional MRI (fMRI) studies have become an extremely hot research topic in the area of neuroscience, generating hundreds of papers each year. Because fMRI studies often monitor changing conditions and complex interactions, they frequently require high-powered multiprocessor systems which can provide sufficient floating point throughput to render images in realtime. While high floating point performance can be had from supercomputers and clusters, these options may be prohibitively expensive for some researchers.

While the floating point performance of commodity CPUs has increased steadily, this has been dwarfed by the improvements in modern graphics processing units (GPUs). While CPUs and GPUs offered comparable floating point performance in early 2003, by the end of 2006, Nvidia's G80 GPU was capable of nearly 60 times the maximum theoretical floating point performance of Intel's Core 2 Duo [1]. This rapid improvement spurred researchers to begin adapting medical imaging algorithms to GPUs.

Recent attempts at porting the MRI algorithms to the DirectX and OpenGL graphics languages have shown the potential for integer-factor performance gains but have been limited by programming architectures not designed for general-purpose computations [3]. To encourage the use of GPUs as general purpose computation engines, Nvidia released the Compute Unified Device Architecture (CUDA) in 2007. CUDA provides a standardized architectural model and framework for general-purpose GPU development.

Despite targeting CUDA at general-purpose application development, early research has shown that achieving high efficiency on the CUDA platform requires careful program design and detailed knowledge of the architecture, even for relatively simple programs [9]. To explore the challenges of developing for CUDA in greater depth, we will explore the optimization of the MRI gridding algorithm. Gridding is a computationally-intensive process used in the medical field to perform MRI procedures. The gridding algorithm takes MRI measurements which are recorded in Fourier k-space

by a magnetic sensor rotating around the patient and translates them into a 2-dimensional image [2].

2. Algorithm Description

2.1 - NFFT & Convolution-Interpolation

The MRI scan commonly samples values in Fourier k-space on a spiral trajectory in either a cylindrical or spherical coordinate system. To convert these samples into a 2 or 3-dimensional image, an inverse Fourier transform must be applied to the data samples. However, the inverse Fourier transform is an $O(n^2)$ operation that does not scale well to the large datasets used in MRI scanning. It is much more efficient to use an inverse FFT, which is an O(n log n) operation, but standard FFT algorithms cannot be applied to data sampled on cylindrical or spherical coordinate systems, because they require that the dataset be sampled on an equispaced coordinate system. Transforming data which has been sampled on a non-Cartesian coordinate system requires a process variously known as the Generalized FFT, Non-equispaced FFT (NFFT), or Gridding. Gridding consists of a three-step process.

1. Discrete 2 or 3-dimensional convolution-interpolation of the data set onto a Cartesian coordinate system
2. Inverse FFT on the Cartesian set
3. Convolution kernel density compensation (also known as deapodization)

Discrete 2-dimensional convolution is commonly used in image processing. Discrete convolution involves taking a single element from the input matrix or data set, multiplying it by a specially-weighted constant matrix (referred to as the convolution kernel or window function), and adding the resulting matrix to the overlapping entries in the output matrix. The convolution window is then 'dragged' across all the elements in the input set. Depending on the window function used, discrete convolution can perform functions such as anti-aliasing, edge detection, sharpening, or softening. To perform interpolation, a symmetrical, center-weighted window function such as a Gaussian or 1st-order Kaiser-Bessel function is used.

2.2 – Software Implementation

A straightforward implementation of the algorithm takes the following form in C:

```
for(i = 0; i < n; i++)
{
   int x = ceil(in[i].x - m/2);
   int y = ceil(in[i].y - m/2);
   float val = in[i].value;
   for(j = 0; j < m; j++)
      for(k = 0; k < m; k++)
         out[x+j][y+k] +=
             val * kernel[j][k];
}
```

where the input data set contains n items and the convolution kernel is size m x m. The algorithm is characterized by a large number of memory operations and significant potential data dependences between outer loop iterations, limiting the application of loop-unrolling optimizations and instruction re-ordering.

Assuming $m >> 1$ and ignoring loop overhead, the code above contains approximately $3nm^2$ memory accesses and $2nm^2$ arithmetic operations. For a machine with a perfect cache, such that the only requests that are serviced by main memory are compulsory cache misses, the minimum number of main memory accesses for a densely populated input set is $n + r^2$, where r is the length of the output image (assuming square images). For image processing applications, $r^2 \sim n$, suggesting that an ideal caching system can reduce the number of memory accesses by a factor of m^2.

3. Compute Unified Device Architecture

3.1 – Overview

CUDA is outlined in detail in Nvidia's CUDA Programming Guide [1]. CUDA 1.0 capable devices are based on Nvidia's G80 GPU, first introduced in the Geforce 8800 series graphics cards.

The G80 is a SIMD multiprocessor architecture with a total of 128 stream processors (SPs) grouped into 16 streaming multiprocessors (SMs). Each CUDA kernel is organized as a multi-dimensional grid of thread blocks, with each block being a multi-dimensional grouping of threads. The dimensional aspect of blocks and grids is an abstraction that can be used to facilitate the calculation of thread IDs and array strides. Threads are assigned to an SM at the block granularity. Each SM executes threads in groups of up to 32, known as warps. Each warp executes logically in SIMD lockstep, though the G80 implementation physically interleaves execution of quarter-warps in a 4-way SMT manner.

CUDA is a host-coprocessor architecture. A general-purpose CPU typically acts as the host processor and a GPU acts as the coprocessor (known as the 'device' in CUDA terminology). CUDA provides a C-like programming environment with extensions to differentiate between code and data structures meant for the host and device execution. The host is responsible for initializing the device, transferring data between system and device memory, and initiating execution of device kernels.

3.2 – Memory Hierarchy

CUDA devices have access to five different types of memory: global, shared, local, constant, and texture.

Global memory is a large memory accessible by all threads. All device kernel code and data must fit within global memory. The global memory on the Geforce 8800 is 768MB, has a 64KB block size, and a latency of 400-600 cycles [1].

Shared memory is local to each SM, and data in shared memory may be shared between threads belonging to the same thread block. If multiple blocks are scheduled to the same SM, shared memory will be evenly partitioned between them. The size of the shared memory is 16KB per SM on the G80. Each memory is 16-banked and has a 4 cycle latency.

Local memory provides the same latency benefits as shared memory, but is local to the SPs and can only be accessed on a per-thread basis. Local memory is typically used as a fast scratchpad.

The constant memory (also referred to as the symbol memory) is a special partition of global memory meant to hold constant values. It is read-only from the device scope and can only be written to by the host. The constant memory is 2-level, with distributed, hardware-managed caches in each SM. Because the constant memory is read-only, these caches do not have any coherence protocol. On the G80, each constant cache is 8KB and has a low access latency.

Texture memory is similar to constant memory, but also includes support for special optimized texture transformation operations. This extra support comes at the cost of higher latency than constant memory for normal read operations.

In addition to the five memory types, each SM has access to a pool of 8192 registers, which are divided evenly among each thread executing in a block.

3.3 – Computational Units

Each of the G80's SMs contains 8 scalar ALUs optimized for 32-bit single precision floating point arithmetic. Each ALU is capable of executing a floating point multiply or addition in a single cycle, but because of the 4-way SMT execution of each warp, an addition or multiplication instruction takes 4 cycles to complete on all threads.

CUDA 1.x devices do not support double precision floating point arithmetic. All double precision operations are automatically converted into single precision. This is an important limitation which restricts the class of scientific applications which can be ported to CUDA. Double precision support is scheduled for CUDA 2.x devices [1].

3.4 – Thread Scheduling Mechanisms

Each SP can support up to 96 active threads, for a total of 12,288 threads on the Geforce 8800. Threads are lightweight and have very low scheduling overhead, giving making CUDA devices with a large number of SMs similar to data parallel architectures. The schedulers are capable of putting threads to sleep to tolerate memory latencies, however each thread within a warp must execute in lock-step. In the case of divergent control paths

within threads of a single block, each instruction is executed in every thread, with vector masks used to disable instructions on untaken control paths.

4. CUDA Optimization Strategies

The G80 architecture provides a maximum theoretical throughput of over 300 GFLOPs, however this is only achievable with kernels with a very large ratio of computation to memory access. As described in Section 2.2, the convolution-interpolation algorithm requires a large amount of reads and writes. Because accesses to global memory require two orders of magnitude more latency than floating point operations, Amdahl's law suggests that the main optimization goal should be to increase the efficiency of memory accesses.

Our strategy for optimizing memory accesses has three main components: making use of cheap thread scheduling to tolerate memory stalls, using the shared memory as a software-managed cache and assigning work to maximize locality, and organizing data structures to maximize bandwidth.

4.1 – Dividing Work Among Threads & Exploiting Locality

The work in the convolution algorithm can be allocated to hardware either by dividing the input data set among threads or by dividing the output matrix into sub-matrices. Additionally, work can be assigned on a fine-grained or coarse-grained level. This gives rise to four distinct division strategies.

An example of a fine-grained, input-driven assignment is shown in Figure 1. This assignment strategy is the simplest to implement, as a different block is assigned to each input point. This method generates a large number of thread blocks, which is favorable for thread scheduling. However, if the input data set is densely distributed on the output matrix, there will be a large amount a data sharing between blocks, particularly as the size of the window function increases. Because data sharing between blocks is only possible via slow global memory and

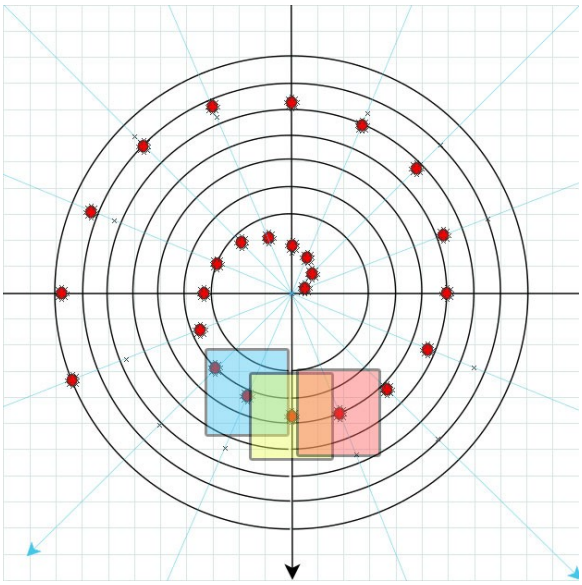

Fig. 1 – Fine-grained, input-driven work assignment. The squares represent the area of the output grid read and written by each block. Overlapping area represents data shared between blocks.

because it is difficult to coordinate threads in different blocks, it is not favorable to use this method.

Figure 2 shows a coarse-grained, input-driven assignment in which multiple consecutive sample points are assigned to the same thread block. This assignment method has low overhead and shares much more data among threads in the same block, offering greater locality. For these reasons, it is the preferred method of assigning work for architectures with hardware-managed caches. While CUDA does not provide a hardware-managed cache for global memory, it is desirable to temporarily write the output matrix values to shared memory as they must be read multiple times in each block. The spiral trajectory of MRI sampling creates an irregularly shaped output sub-matrix when assigning work based on the input data set. This irregular shape makes it difficult to assign work to different threads within the block without using a diverging control scheme or complicated arithmetic on the thread ID, adding a large amount of overhead to a CUDA implementation.

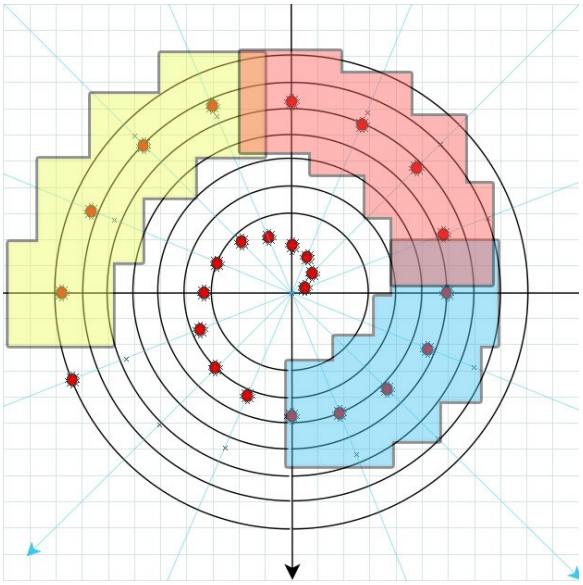

Fig. 2 – Coarse-grained, input-driven assignment

Figure 3 demonstrates fine-grained, output-driven assignment. In this method, a thread block is assigned to each point on the output grid and reads from any input samples within the range of the window function. This implementation shares no data between blocks and has a pareto optimal number of memory write operations [7], however it incurs a large amount of read overhead because each thread must search for an input data value within range of the output grid point. For sparse data sets it might be worthwhile to store the dataset in cacheable constant memory, however for sparse data sets it could be difficult to find enough data points in range to completely populate a warp, leading to performance degradation from unutilized processors.

Figure 4 shows the final configuration, coarse-grained, output driven assignment (which we refer to as sector-based assignment). This method is the one chosen in our implementation. It has the good data locality properties of coarse-grained assignment while allowing for simplified control schemes and thread assignment because of the regular shape of the output sector. The major disadvantage to this scheme is that data points must be presorted into the appropriate output sector a priori – however this can be done using efficient CPU preprocessing prior to launching the device kernel.

The calculations shown in Section 2.2 suggest that a perfect caching system could reduce the number of accesses to global memory by a factor of 256 for a convolution window size of 16. However, CUDA does not provide any cache for global memory. While the constant & texture memory partitions are cacheable, these memories are read-only, and thus cannot be used to store the output matrix which accounts for half of all memory references.

To overcome the lack of caching on the global memory, we use the per-SM shared memory as a software-managed cache. Using shared memory

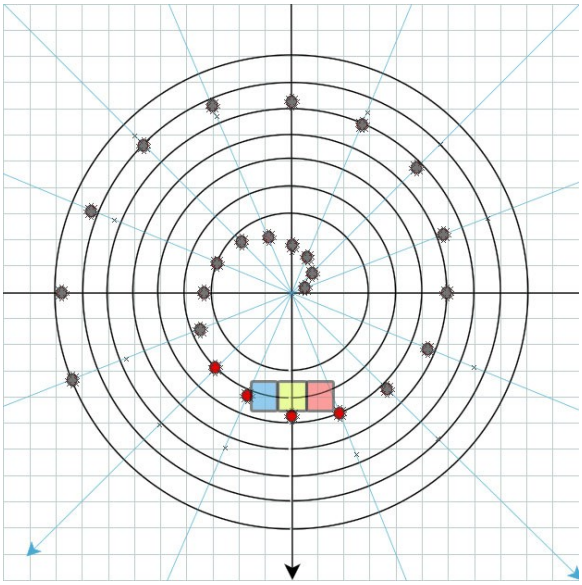

Fig. 3 – Fine-grained, output-driven assignment – Red dots indicate samples within range of the three blocks shown.

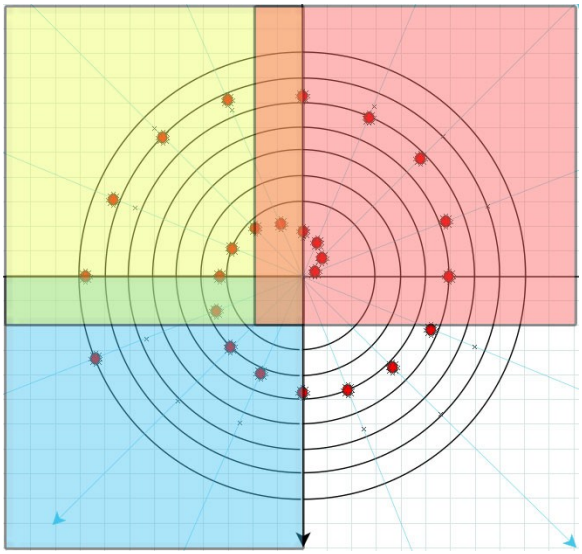

Fig. 4 – Coarse-grained, output-driven assigment

as a cache creates many challenges for kernels in which data may be shared between different thread blocks. There is no coherence mechanism to ensure that the same memory location is not 'dirty' in the shared memory on multiple SMs. Further, it is impossible to implement such a mechanism as CUDA 1.x only allows thread synchronization on a per-block basis. This means that the programmer must exercise great care in algorithm design to ensure correctness when attempting to share data between blocks. To overcome this challenge, our implementation pads the portion of the output matrix belonging to each block by an amount equal to half the size of the window function. This padded area represents an area of the output matrix that belongs to another block. Each block can write to its padded area safely. When all blocks have completed computation, each block writes portion of the output matrix it owns. When this is complete, the padded areas are added to the corresponding entries of the output matrix. Because CUDA 1.x does not support global synchronization, the adding of the padded areas is currently performed by the host.

4.2 – Thread Scheduling

Because stalling, starting, and scheduling threads is inexpensive on CUDA, launching large numbers of threads is a general method of tolerating memory latencies for any device kernel. Achieving a large number of active threads can be a complex problem. Because threads are scheduled in groups of blocks, it is necessary to create enough threads that there are multiple blocks per SM. Nvidia suggests creating at least 2 blocks per SM [1], which equates to 32 blocks on the Geforce 8800. In the case of the convolution kernel, each thread block can be considered to represent a sector of the output matrix. Assuming a square matrix, it is preferable to use a power-of-two grid size that is a perfect square. Increasing the number of blocks decreases the amount of the output matrix owned by each block, limiting the amount of locality that can be exploited by storing the output matrix block in shared memory. Conversely, the maximum sector size is limited by the amount of shared memory available to each block. Each element of the output matrix is 8-bytes in size, representing a single-precision complex number. Each shared memory is 16 KB, but this size must be divided among the number of blocks assigned to each SM. Assuming the minimum recommended value of 2 blocks per SM, each block gets 8 KB of memory, representing 1K entries. The largest square matrix that can fit in this area is 32 x 32. Increasing the sector size beyond this amount requires complex management of the shared memory resources within each block. The alternative is to schedule only one block per SM. This would allow for 2K entries per sector which could accommodate a 48 x 48 matrix which would offer more locality and less sharing between blocks at the expense of being able to schedule blocks to tolerate global memory latencies. The optimal approach depends on the specific application as the importance of large sectors is influenced by the density of the input data set and the size of the convolution window.

4.3 –Data Structures

Both the global memory and shared memory performance can benefit from optimization of the size and location of data structures in memory.

While the global memory has relatively large cycle latency, it also has a very high bandwidth - 86 GB/s on the Geforce 8800 GTX. This high bandwidth can be used to amortize latency costs by consolidating the reads and writes of the threads in the same warp. CUDA allows for coalescing reads and writes provided the data structures and thread access patterns meet certain criteria. Each thread in a 16-thread half-warp must be accessing its word such that all the threads access a permutation of words in a 64 or 128-byte contiguous block. Further, the thread which accesses the word aligned at the beginning of the 64-byte boundary must have a thread ID x-coordinate value of zero in the thread block [1]. This can be achieved without great difficulty because the CUDA programming extensions provide pragmas for automatically aligning data structures in memory with arbitrary boundary lengths. To ensure that the thread ID requirements were met, we structured our thread grid to have a length of 16, such that all groups of 16 threads would have one thread with a thread ID of zero.

The shared memory has 16 banks, allowing all 32 threads in a warp to access memory in a 4-

cycle read operation provided there are no bank conflicts. Shared memory addressing uses a 4-byte offset between banks – the size of one single-precision floating point number. To ensure that there are no bank conflicts when accessing a data structure in shared memory, the size of the structure in units of 4-bytes must be co-prime to 16. Initially our implementation used an 8-byte datatype that represented a complex number containing single-precision real and imaginary parts. Accessing this structure resulted in only 50% bank utilization. To solve this problem, we divided our output array into two separate arrays: one holding real values and the other holding imaginary values. This arrangement changed the element size within each structure from 8 to 4-bytes, and improved bank utilization to 100%. In cases of more complicated data structures, CUDA provides a special memory transfer operation that will automatically pad the data structures and return the optimal stride for accessing the structure; however manual optimizations may be more size efficient and allow for a larger working set within shared memory.

4.4 – Control & Computations

The SIMD nature of CUDA favors simple non-divergent control schemes. CUDA devices handle control divergence using a method similar to vector masking. Each processor must execute all instructions, including those in control paths not active for the processor. Inactive instructions simply are not committed. This means that complex and diverging control schemes may degrade performance by creating wasted execution cycles in which a processor may do no useful work.

While Amdahl's Law suggests that most optimization time should be spent on memory latency, there is room for some large-scale computational optimizations. The primary optimization is to do offline computation of the convolution window function. Because the window function is constant for a given window size, it can be computed offline and stored as a constant array. Rather than computing an expensive operation such as the Bessel function, the processor can simply store the kernel values in a special array that is indexed by the distance between an input sample point and the grid point it is being interpolated onto. The processor then only needs to calculate the distance between the sample and each grid point within the kernel window, hash the value, and use it to index into the pre-computed kernel array.

4.5 – Using Shared vs. Constant Memory

Given that constant memory provides a hardware-managed distributed cache with fast access, it would initially seem to be ideal for storing the pre-computed convolution kernel, however this is not the case. While the local memory has 16 banks for handling multiple requests in each cycle, we found that multiple requests to different addresses in the constant memory cache were serialized. We performed some preliminary tests using random and structured access patterns to constant memory, and found that the latency of constant memory was approximately equal to shared memory when all threads were accessing the same location but around three times slower than shared memory when each thread was accessing a random address.

Since accesses into the pre-computed kernel are based on the distance between the sample and each grid point within the kernel window, they rarely hit on the same memory address in the same cycle. This suggests that we can get better performance by loading the kernel into shared memory rather than using the constant memory. In practice, we found that because kernel read operations make up a relatively small fraction of computation time, there is only a slight performance benefit to storing the kernel in shared memory, and this benefit is offset by the reduced working set that can be fit in shared memory. For these reasons, we chose to use constant memory to store the kernel.

5. Methodology

We implemented the convolution-interpolation algorithm in CUDA using some of the major optimization strategies described earlier. To gauge the efficiency each strategy, we measured the improvement in computation time after applying each optimization to a naïve implementation using fine-grained, input-driven work assignment. We also

compared the performance of the fully-optimized CUDA implementation to an optimized single-threaded CPU convolution library. CUDA device code was run on a Geforce 8800 GTX and the CPU implementation was run on a 2.0 GHz Intel Core 2 Duo processor. All tests used a pool of 5000 samples uniformly distributed on a 512 x 512 grid. A Gaussian convolution kernel with a window size of 8 was used with 10X oversampling.

Additionally, we ran the same optimized device code on the Geforce 8400M GS, a GPU based on the Nvidia G86 core. The Geforce 8400M GS is a low-cost integrated graphics chip featuring only 2 SMs and a memory bandwidth of 9.6 GB/s [8]. Our purpose was to determine if code optimized for one CUDA device would scale well to another.

6. Results

The relative speed-ups of 5 CUDA-based optimizations are shown in Table 1.

Optimization	Speedup
Caching input samples in shared memory	40%
Aligning data structures for coalesced reads/writes & maximum bank utilization	70%
Pre-computing convolution kernel and storing in constant memory space	20%
Sector-driven assignment and use of shared memory as software-managed cache	310%
Increasing the blocks per SM from 1 to 2	35%

Table 1: Individual speedup based on optimization

The sector-driven division of work, which enabled the use of the shared memory as a software-managed cache for the output grid, produced the most significant performance increase, more than tripling the speed of the convolution interpolation algorithm when compared to the naïve implementation which only used global memory.

Aligning data structures to optimize memory bandwidth and improve shared memory bank utilization provided a large performance increase as well, improving the baseline performance by 70%. Doubling the number of blocks per SM resulted in an unexpectedly large performance increase of 35%. This indicates that block scheduling is an effective way of tolerating memory latencies.

The performance of the CUDA implementation running on the Geforce 8800 GTX and Geforce 8400M GS are compared to the CPU implementation in Table 2.

Device	Speedup vs CPU
Geforce 8800 GTX	114X
Geforce 8400M GS	17.5X

Table 2 – Speedup of CUDA implementations vs CPU implementation

The Geforce 8800 device showed an impressive performance increase compared to the Core 2 Duo, achieving a speedup of 114X, however this speedup only corresponds to a throughput of approximately 20 GFLOPs – a small fraction of the maximum theoretical throughput for the 8800. This gap is shows that our program's performance is still dominated by memory latency.

The Geforce 8400M GS produced surprisingly good results, considering it is one of the least powerful CUDA-capable devices available. Performance also scaled well – compared to the 8800, the 8400 has 8 times as fewer processors and 9 times less memory bandwidth, but only performed 6.5 times slower than the 8800. This better than expected scaling may be attributed to the fact that the 8400 used the fine-grained, input-based block assignment, which resulted in many more blocks per SM. This allowed for more efficient block scheduling.

7. Conclusions

Nvidia's CUDA coupled with the Geforce 8800 GTX produced impressive speed increases on the convolution-interpolation algorithm used in MRI gridding. In addition to medical imaging, convolution-interpolation is an algorithm with similar characteristics to many general-purpose image processing operations.

While performance improvement is impressive, it should be noted that several of the choices in the testing – such as the large kernel window size and choice of an output grid size that would fit within the distributed shared memory – were more favorable to the data parallel capabilities of the CUDA device, and that more generalized testing would likely show a lesser improvement in performance. However, the main purpose of this paper was not to determine the absolute speed difference between CPU and GPU implementations, but rather to highlight optimization strategies employed when porting an image processing algorithm to CUDA.

With Intel and AMD developing their own large-scale SIMD processor efforts [4] [5], the creation of a standardized architectural model and development environment for these massively SIMD processors will be required for wide-scale adoption. CUDA represents Nvidia's attempt to create such an environment. Whether it will be adopted as a general standard is an open question. While this work has shown that CUDA can be used to achieve large speedups over traditional CPU architectures for data parallel algorithms with a relatively small re-engineering effort, several of the optimization strategies, such as the data structure layout, thread scheduling, and use of shared vs. constant memory, are closely tied to the underlying hardware architecture.

CUDA still lacks some key features that could be used to make the adaption of gridding algorithms easier. In particular the lack of a coherence-backed cache system coupled with limited support for atomic operations and global synchronization make correct implementation of algorithms with a large amount of data sharing a challenge. To some degree, these limitations may be addressed in future revisions of CUDA. CUDA 1.1 already adds support for some atomic operations on 1.1-compatible devices. Future revisions may bring new opportunities for CUDA development.

8. Future Work

Our optimizations for 2D gridding could be extended to support 3D imaging. Since interpolation is an inherently lossy process, it would be worthwhile to explore methods of using inexact calculations to accelerate the processing algorithms, such as using a fast estimation for the distance between a sample and a grid point rather than an expensive Pythagorean computation.

While this study has focused on performance and computational efficiency, it would be useful to compare the power consumption and instruction power efficiency between CPU and GPU implementations.

Acknowledgements

The author would like to thank Doug Janes for his advice on discrete convolution optimizations and grid topologies.

References

[1] *NVIDIA CUDA Compute Unified Device Architecture Programming Guide,* http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, Accessed March 25, 2008

[2] H. Schomberg, and J. Timmer, "The gridding method for image reconstruction by Fourier transformations," *IEEE Transactions on Medical Imaging, vol. 14(3),* pp. 596-607, 1995.

[3] T. Schiwietz, T. Chang, P. Speier & R. Westermann, "MR image reconstruction using the GPU," *Proceedings of the SPIE,* vol. 6142, pp. 1279-1290, 2006.

[4] *Intel IDF Preview: Tukwilla, Dunnington, Nehalem and Larrabee*, http://www.pcper.com/article.php?aid=534, Accessed March 27, 2008.

[5] *AMD Financial Analyst Day*, http://download.amd.com/Corporate/MarioRivasDec2007AMDAnalystDay.pdf, Accessed March 27, 2008.

[6] T.S. Sorensen, T. Schaeffter, K.O. Noe and M. Schacht-Hansen, "Fast Gridding on Commodity Graphics Hardware," *Proceedings of Workshop on Non-Cartesian MRI*, ISMRM: Arizona, 2007.

[7] T.S. Sorensen, T. Schaeffter, K.O. Noe and M. S. Hansen, "Accelerating the Nonequispaced Fast Fourier Transform on Commodity Graphics Hardware," *IEEE Transactions on Medical Imaging*, vol. 27, 538-547, 2007.

[8] *Geforce 8400M*, http://www.nvidia.com/object/geforce_8400M.html, Accessed May 07, 2008.

[9] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, and W. Hwu. "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA," *PPoPP 2008*, ACM: Salt Lake City, 2008.

Appendix

Partial source code listing for CUDA implementation

convInt.h

```
/* Header for convInt */
#ifndef _CONVINT_
#define _CONVINT_

#define KERNEL_RADIUS 8
#define GRANULARITY 0.5f
#define IVG 2 //Must be (GRANULARITY)^-1 (should be int)
#define GRID_SIZE 512
#define NUM_SAMPLES 8192
#define THREADS_PER_TBLOCK 32
#define BLOCK_SIZE 32
#define NUM_TBLOCKS 256 //Must be (GRID_SIZE/BLOCK_SIZE)^2

/* Represents complex number in rectangular coordinates */
typedef struct __align__(8) {
	float real;
	float img;
} complex;

/* Datapoint from MRI scan. Contains an X-Y coordinate corresponding to
   Fourier k-space and a complex value */
typedef struct __align__(16) {
	float x;
	float y;
	complex data;
} kdata;

__constant__ float const_kernel[4*KERNEL_RADIUS*KERNEL_RADIUS];
#endif //_CONVINT_
```

convInt2.cu

```
/* 2-Dimensional Convolution-Interpolation on CUDA */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

/* Program constants, macros, and structs declared here */
#include <convInt.h>

/* CUDA Utilities */
#include <cutil.h>

/* GPU kernel code */
#include <convInt_kernel2.cu>

/* Calculates the k-space Gaussian weighting for a given distance */
float gaussian(float dist)
{
	return expf(-(dist*dist)/2.0f);
```

```
}

/* GPU kernel wrapper
        Initializes device kernel and executes data transactions between
        device and host */
void conv(kdata* dataset, complex* out_grid, float* conv_kernel)
{
        /* Initialize CUDA Device */
        CUT_DEVICE_INIT();

        /* Copy convolution kernel to symbol memory on device */
        CUDA_SAFE_CALL(cudaMemcpyToSymbol(const_kernel, conv_kernel,
4*KERNEL_RADIUS*KERNEL_RADIUS*sizeof(float)) );

        /* Allocate global device memory for data list & copy data */
        kdata* data_pt;
        int memsize = NUM_SAMPLES * sizeof(kdata);
        CUDA_SAFE_CALL(cudaMalloc((void**)&data_pt, memsize) );
        CUDA_SAFE_CALL(cudaMemcpy(data_pt, dataset, memsize, cudaMemcpyHostToDevice) );

        /* Allocate global device memory for grid */
        complex* grid_pt;
        memsize = GRID_SIZE * GRID_SIZE * sizeof(complex);
        CUDA_SAFE_CALL(cudaMalloc((void**)&grid_pt, memsize) );

       /* Initialize & start GPU timer */
       unsigned int timer = 0;
       CUT_SAFE_CALL(cutCreateTimer(&timer) );
       CUT_SAFE_CALL(cutStartTimer(timer) );

        /* Compute execution configuration */
        dim3 dimBlock(THREADS_PER_TBLOCK, 1);
        dim3 dimGrid(NUM_TBLOCKS, 1);

        /* Launch computation on GPU */
        devcon<<<dimGrid, dimBlock>>>(data_pt, grid_pt);
       CUT_CHECK_ERROR("Kernel execution failed");

        /* Read data from device into host memory */
        CUDA_SAFE_CALL(cudaMemcpy(out_grid, grid_pt, memsize, cudaMemcpyDeviceToHost) );
        printf("Output check: %f\n", out_grid[10].real);

       /* Stop timer, display result, and clean-up */
       CUT_SAFE_CALL(cutStopTimer(timer));
       printf("Processing time: %f (ms) \n", cutGetTimerValue(timer));
       CUT_SAFE_CALL(cutDeleteTimer(timer));

        /* Clean-up device memory */
        cudaFree(data_pt);
        cudaFree(grid_pt);

}

/* Host-Main */
int main()
{
        /* Pre-compute convolution-interpolation lookup table */
        /* Allocate lookup table */
        const int OS_KERNEL_RADIUS = (int) (KERNEL_RADIUS/GRANULARITY) + 1;
        const int KERNEL_ENTRIES = 4*OS_KERNEL_RADIUS*OS_KERNEL_RADIUS;
        float *conv_kernel = (float *)malloc((KERNEL_ENTRIES)*sizeof(float));
        if(conv_kernel == NULL) {printf("Memory allocation error\n"); exit(1);}
```

```
	/* Populate table */
	float distance = 0.0f;
	for(int i = 0; i < KERNEL_ENTRIES; i++)
	{
		conv_kernel[i] = gaussian((distance/((float) KERNEL_RADIUS)));
		distance = distance + GRANULARITY;
	}

	/* Generate data set */
	srand(1338);
	kdata *dataset = (kdata *) malloc(NUM_SAMPLES * sizeof(kdata));
	if(dataset == NULL) {printf("Memory allocation error\n"); exit(1);}

	/* Generates a random point within the GRID with given granularity
		points are sorted by their target block */
	int blocks_per_row = GRID_SIZE / BLOCK_SIZE;
	int num_blocks = blocks_per_row * blocks_per_row;
	int samples_per_block = NUM_SAMPLES / num_blocks;
	for(int i = 0; i < num_blocks; i++)
	{
		int x_start = (i % blocks_per_row) * BLOCK_SIZE;
		int y_start = (i / blocks_per_row) * BLOCK_SIZE;
		for(int j = 0; j < samples_per_block; j++)
		{
			kdata gtmp;
			gtmp.x = x_start + 0.001f + GRANULARITY * (float) (rand() % ((BLOCK_SIZE
- 1) * IVG ));
			gtmp.y = y_start + 0.001f + GRANULARITY * (float) (rand() % ((BLOCK_SIZE
- 1) * IVG ));
			gtmp.data.real = rand() / (float) (RAND_MAX);
			gtmp.data.img = rand() / (float) (RAND_MAX);
			dataset[i*samples_per_block + j] = gtmp;
		}
	}

	/* Allocate padded output grid */
	complex *out_array = (complex *) calloc((GRID_SIZE*GRID_SIZE), sizeof(complex));
	if(out_array == NULL) {printf("Calloc Memory allocation error\n"); exit(1);}

	/* Perform convolution-interpolation */
	conv(dataset, out_array, conv_kernel);

	/* Cleanup Host Memory */
	free(conv_kernel);
	free(dataset);
	free(out_array);

	/* Finished */
	printf("Program completed successfully\n");
	exit(0);
}
```

convInt_kernel2.cu

```
/* Device kernel */

#include <convInt.h>

/* Calculates the index into the kernel lookup table for set of points */
__device__ int get_kernel_index(int x, int y, float xf, float yf)
{
	float del_x = xf - x;
	float del_y = yf - y;
	float dist = sqrt(del_x * del_x + del_y * del_y);
	return (int) (dist / GRANULARITY);
}

/* Device convolultion-interpolation */
__global__ void devcon(kdata* dataset, complex* out_array)
{

	int start = (NUM_SAMPLES / NUM_TBLOCKS) * blockIdx.x;
	int length = NUM_SAMPLES / NUM_TBLOCKS;
	start = start + threadIdx.x;

	int blocks_per_row = GRID_SIZE / BLOCK_SIZE;
	int x_start = (blockIdx.x % blocks_per_row) * BLOCK_SIZE;
	int y_start = (blockIdx.x / blocks_per_row) * BLOCK_SIZE;

	/* Grid sub-section in shared memory */
	__shared__ complex local_block[BLOCK_SIZE*BLOCK_SIZE];

	/* Perform convolution-interpolation */
	for(int i = threadIdx.x; i < length; i = i + THREADS_PER_TBLOCK)
	{
		__shared__ kdata datasample;
		datasample = dataset[i];

		/* Find convolution points. */
		int window_x1 = max(0, (int)ceil((datasample.x - x_start) - KERNEL_RADIUS));
		int window_x2 = min((BLOCK_SIZE - 1), (int)floor((datasample.x - x_start) +
KERNEL_RADIUS));
		int window_y1 = max(0, (int)ceil((datasample.y - y_start) - KERNEL_RADIUS));
		int window_y2 = min((BLOCK_SIZE - 1), (int)floor((datasample.y - y_start) +
KERNEL_RADIUS));

		/* Apply interpolation kernel to points within window */
		for(int j = window_x1; j < window_x2 + 1; j++)
		{
			for(int k = window_y1; k < window_y2 + 1; k++)
			{
				int out_idx = j*BLOCK_SIZE + k;
				int lidx = get_kernel_index(j, k, (datasample.x - x_start),
(datasample.y - y_start));
				float conv_mul = const_kernel[lidx];
				local_block[out_idx].real += datasample.data.real * conv_mul;
				local_block[out_idx].img += datasample.data.img * conv_mul;

				/* Ensure all threads finished updating their grid point before
					attempting to read from the next point */
				syncthreads();
			}
		}
	}
```

```
        /* Copy local block back to global memory */
        for(int i = 0; i < BLOCK_SIZE; i = i + 1)
                for(int j = threadIdx.x; j < BLOCK_SIZE; j = j + THREADS_PER_TBLOCK)
                        out_array[(y_start + i)*BLOCK_SIZE + x_start + j] =
local_block[i*BLOCK_SIZE + j];
}
```