How GPUs Can Improve the Quality of Magnetic Resonance Imaging

Sam S. Stone, Haoran Yi, Justin P. Haldar, Wen-mei W. Hwu, Bradley P. Sutton, and Zhi-Pei Liang
{ssstone2, yi4, haldar, w-hwu, bsutton, z-liang}@uiuc.edu
University of Illinois at Urbana-Champaign, Urbana, IL 61801

***Abstract*— In magnetic resonance imaging (MRI), non-Cartesian scan trajectories are advantageous in a wide variety of emerging applications. Advanced reconstruction algorithms that operate directly on non-Cartesian scan data using optimality criteria such as least-squares (LS) can produce significantly better images than conventional algorithms that apply a fast Fourier transform (FFT) after interpolating the scan data onto a Cartesian grid. However, advanced LS reconstructions require significantly more computation than conventional reconstructions based on the FFT. For example, one LS algorithm requires nearly six hours to reconstruct a single three-dimensional image on a modern CPU. Our work demonstrates that this advanced reconstruction can be performed quickly and efficiently on a modern GPU, with the reconstruction of a 64^3 3D image requiring just three minutes, an acceptable latency for key applications.**

This paper describes how the reconstruction algorithm leverages the resources of the GeForce 8800 GTX (G80) to achieve over 150 GFLOPS in performance. We find that the combination of tiling the data and storing the data in the G80's constant memory dramatically reduces the algorithm's required bandwidth to off-chip memory. The G80's special functional units provide substantial acceleration for the trigonometric computations in the algorithm's inner loops. Finally, experiment-driven code transformations increase the reconstruction's performance by as much as 60% to 80%.

I. Introduction

Mainstream microprocessors such as the Intel Pentium and AMD Opteron families have driven rapid performance increases and cost reductions in science and engineering applications for two decades. These commodity microprocessors have delivered GFLOPS to the desktop and hundreds of GFLOPS to cluster servers. This progress, however, has slowed since 2003 due to constraints on power consumption. Since that time, advances in computational throughput have been driven by graphics processing units (GPUs), which will likely drive the next wave of computational advances for science and engineering applications. This trend is illustrated in Figure 1.

First Workshop on General Purpose Processing on Graphics Processing Units (GPGPU). October 4, 2007, Boston, MA, USA.
This version contains minor corrections of the original workshop paper.

Recent advances in architecture have also increased the GPU's attractiveness as a platform for science and engineering applications. Prior to 2006, GPUs found very limited use in this domain due to their limited support for both IEEE floating-point standards and arbitrary memory addressing. However, the recently released AMD R580 and NVIDIA G80 GPUs offer full support for IEEE single-precision floating-point values (with double-precision soon to follow) and permit reads and writes to arbitrary addresses in memory [1], [2]. Furthermore, modern GPUs use massive multi-threading, fast context switching, and high memory bandwidth to tolerate ever-increasing latencies to main memory by overlapping long-latency loads in stalled threads with useful computation in other threads [3].

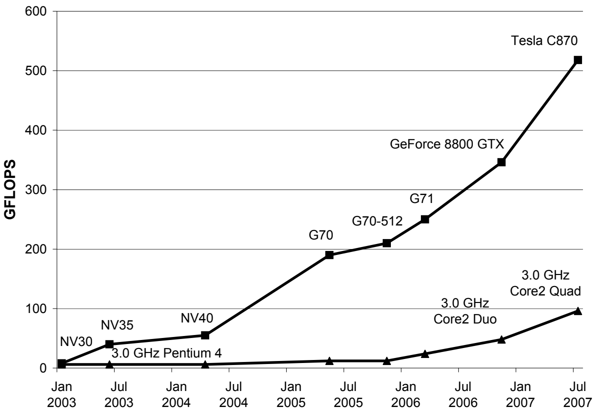

Fig. 1. Theoretical peak GFLOPS on modern GPUs and CPUs. Each core of the Core2 processors can retire four single-precision, multiply-accumulate operations per cycle, yielding peak theoretical throughput of 24 GFLOPS/core. The G80 and Tesla have peak theoretical throughput of 345.6 GFLOPS and 518 GFLOPS, respectively. Earlier data points are derived from inspection of slide 7 in [4].

Increased programmability has also enhanced the GPU's suitability for science and engineering applications. For example, the G80 supports the single-program, multiple-data (SPMD) programming model, in which each thread is created from the same program and operates on a distinct data element, but all threads need not follow the same control flow path.

As the SPMD programming model has been used on massively parallel supercomputers in the past, it natural to expect that many high-performance applications will perform well on the GPU [3], [5]. Furthermore, general-purpose applications targeting the G80 are developed using ANSI C with simple extensions, rather than the cumbersome graphics application programming interfaces (APIs) [6], [7] and high-level languages layered on top of graphics APIs [8], [9], [10] that have been used in the past.

Magnetic resonance imaging (MRI) is one application that can benefit greatly from these increases in computational resources and advancements in architecture and programmability. Emerging MRI applications such as functional imaging of the brain and dynamic imaging of the beating heart often use non-Cartesian scan trajectories to quickly obtain high-quality scan data. Advanced reconstruction algorithms that operate directly on non-Cartesian scan data using optimality criteria such as least-squares (LS) can produce significantly better images than conventional algorithms that apply a fast Fourier transform (FFT) after interpolating the scan data onto a Cartesian grid. However, advanced methods based on LS require several orders of magnitude more computation than FFT-based reconstructions because the underlying structure of the LS-based reconstruction problem is much more complex.

For these advanced reconstruction algorithms to be viable in clinical settings, dramatic and inexpensive computational acceleration is required. We find that certain MRI reconstructions designed for non-Cartesian data are extremely well suited to acceleration on modern GPUs. In particular, we show that an advanced algorithm, which requires nearly six hours to reconstruct a 3D image on a modern CPU, completes the same reconstruction in just three minutes on a modern GPU. The acceleration of nearly 300X achieved on the GPU makes the advanced reconstruction viable in clinical settings for many emerging MRI applications.

The remainder of this paper is organized as follows. Section II describes the architecture of the NVIDIA GeForce 8800 GTX and discusses the advantages of advanced MRI reconstructions. Section III presents the GPU-based implementation of the advanced reconstruction algorithm. Section IV describes experimental methodology. Section V presents results and discusses architectural features of the GeForce 8800 that enable the advanced reconstruction algorithm to achieve roughly 150 GFLOPS in performance. Sectio VI discusses related work in GPGPU-based medical imaging. Section VII concludes.

II. Background

A. The GeForce 8800 GTX

The GeForce 8800 GTX (G80) has a large set of processor cores which are able to directly address a global memory. This architecture supports the single-program, multiple-data (SPMD) programming model, which is more general and flexible than the programming models supported by previous generations of GPUs, and which allows developers to easily implement data-parallel kernels. In this section we discuss NVIDIA's Compute Unified Device Architecture (CUDA) and the microarchitectural features of the G80 that are most relevant to accelerating MRI reconstruction. A more complete description is found in [2], [11].

From the application developer's perspective, the CUDA programming model consists of ANSI C supported by several keywords and constructs. CUDA treats the GPU as a coprocessor that executes data-parallel kernel functions. The developer supplies a single source program encompassing both host (CPU) and kernel (GPU) code. NVIDIA's compiler, nvcc, separates the host and kernel codes, which are then compiled by the host compiler and nvcc, respectively. The host code transfers data to and from the GPU's global memory via API calls, and initiates the kernel code by performing a function call.

Figure 2 depicts the microarchitecture of the G80. The GPU consists of 16 *streaming multiprocessors* (SMs), each containing eight *streaming processors* (SPs), or processor cores, running at 1.35GHz. Each SM has 8,192 registers that are shared among all threads assigned to the SM. The threads on a given SM's cores execute in SIMD (single-instruction, multiple-data) fashion, with the instruction unit broadcasting the current instruction to the eight cores. Each core has a lone arithmetic unit that performs single-precision floating point arithmetic and 32-bit integer operations. Additionally, each SM has two *special functional units* (SFUs), which perform more complex FP operations such as the inverse square root and the trigonometric functions with low latency. Both the arithmetic units and the SFUs are fully pipelined. Thus, each SM can perform 18 FLOPS per clock cycle (1 multiply-add operation per SP and one complex operation per SFU), yielding 388.8 GFLOPS of peak theoretical performance for the GPU ($16\,\mathrm{SM} * 18\,\mathrm{FLOP/SM} * 1.35\mathrm{GHz}$).

The G80 has 86.4 GB/s of bandwidth to its 768MB, off-chip, global memory. Nevertheless, with computational resources supporting nearly 400 GFLOPS of performance and each FP instruction operating on up to 8 bytes of data, applications can easily saturate that bandwidth. Therefore, as depicted in Figure 2, the G80

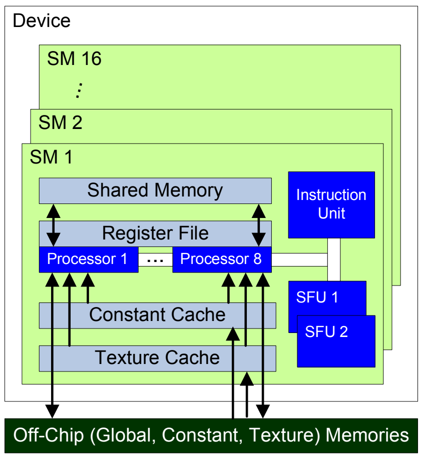

Fig. 2. Basic Organization of the G80

has several on-chip memories that can exploit data locality and data sharing to reduce an application's demands for off-chip memory bandwidth. For example, the G80 has a 64KB, off-chip *constant memory*, and each SM has an 8KB constant memory cache. Because the cache is single-ported, simultaneous accesses of different addresses yield stalls. However, when multiple threads access the same address during the same cycle, the cache broadcasts the address's value to those threads with the same latency as a register access. This feature proves quite beneficial for MRI reconstruction algorithms that operate on non-uniform scan data. In addition to the constant memory cache, each SM has a 16KB *shared memory* that is useful for data that is either written and reused or shared among threads. Finally, for read-only data that is shared by many threads but not necessarily accessed simultaneously by all threads, the off-chip texture memory and the on-chip texture caches exploit 2D data locality to provide dramatic reduction in memory latency.

Threads executing on the G80 are organized into a three-level hierarchy. At the highest level, each kernel creates a single *grid*, which consists of many *thread blocks*. The maximum number of threads per block is 512. Each thread block is assigned to a single SM for the duration of its execution. Threads in the same block can share data through the shared memory and can perform barrier synchronization by invoking the __syncthreads primitive. Threads are otherwise independent, and synchronization across thread blocks is safely accomplished only by terminating the kernel. Finally, threads within a block are organized into *warps* of 32 threads. Each warp executes in SIMD fashion, with the SM's instruction unit broadcasting the same instruction to the eight cores on four consecutive clock cycles.

SMs can perform zero-overhead scheduling to interleave warps on an instruction-by-instruction basis to hide the latency of global memory accesses and long-latency arithmetic operations. When one warp stalls, the SM can quickly switch to a ready warp in the same thread block or a ready warp in some other thread block assigned to the SM. The SM stalls only if there are no warps with ready operands available. Scheduling freedom is high in many applications because threads in different warps are independent with the exception of explicit synchronizations among threads in the same thread block.

Tuning the performance of a CUDA kernel often involves a fundamental trade-off between the efficiency of individual threads and the thread-level parallelism (TLP) among all threads. This trade-off exists because many optimizations that improve the performance of an individual thread tend to increase the thread's use of limited resources that are shared among all threads assigned to an SM. For example, as each thread's register usage increases, the total number of threads that can simultaneously occupy the SM decreases. Because threads are assigned to an SM not individually, but in large thread blocks, a small increase in register usage can cause a correspondingly much larger decrease in SM occupancy. Section V-D examines this trade-off in the context of MRI reconstruction.

B. *MRI Reconstruction with Non-Cartesian Data*

Magnetic resonance imaging (MRI) is commonly used by the medical community to safely and non-invasively probe the structure and function of biological tissues from all regions of the body, and images generated using MRI have a profound impact in both clinical and research settings. MRI is performed using specialized hardware that takes advantage of the known quantum-mechanical interactions of atomic nuclei with magnetic fields. Due to the imaging physics, data is modeled as sampling the desired image in the k-space domain (i.e., the *spatial-frequency domain* or *Fourier transform domain*). The sampled k-space points define the scan trajectory, and the geometry of the scan trajectory has a first-order impact on the quality of the reconstructed image and on the complexity of the reconstruction algorithm.

For Cartesian trajectories, which sample k-space on a uniform grid, statistically optimal image reconstruction can be performed quickly and efficiently by applying the fast Fourier transform (FFT) directly to the acquired data. There is no such efficient algorithm for optimal reconstruction from data collected with more general sampling trajectories. However, non-Cartesian Fourier sampling is becoming increasingly common in MRI. For example, trajectories with radial [12], spiral [13], stochastic [14], and randomly-perturbed [15] sampling patterns can be superior to Cartesian trajectories in terms of imaging speed, hardware requirements, and sensitivity to artifacts caused by non-ideal experimental conditions.

A variety of techniques have been proposed to reconstruct non-Cartesian (*non-uniformly sampled*) data. In the most common approach, *gridding*, the data is first interpolated onto a uniform Cartesian grid and then reconstructed in one step via the FFT [16], [17]. However, gridding satisfies no optimality criterion, and has limited ability to deal with important imaging scenarios, such as parallel imaging [18].

In short, non-Cartesian scan trajectories are often superior to Cartesian scan trajectories in terms of the quality of the data obtained during the scan. Non-uniformly sampled data can be interpolated onto a Cartesian grid and then reconstructed via the FFT in one step, but this technique introduces inaccuracies and produces sub-optimal images. By contrast, optimal image reconstructions can be performed using advanced algorithms that perform the reconstruction iteratively [19], [20], [21], [22], [23], [24]. These iterative algorithms require substantially more computation than algorithms based on gridding. A class of iterative algorithms leverages the observations of Wajer *et al.* in [25] to remove all approximations from the reconstruction while simultaneously improving the reconstruction's speed. These advanced algorithm have been impractical for large-scale problems due to computational constraints, but become practical when accelerated on the GPU. One such algorithm is described below.

The advanced reconstruction algorithm operates directly on non-uniformly sampled data and uses the least-squares (LS) optimality criterion. The algorithm uses an iterative linear solver to solve Eq. 1, where ρ is the desired image, $\mathbf{F}^H\mathbf{F}$ is a matrix that depends only on the scan trajectory, and $\mathbf{F}^H\mathbf{d}$ is a vector that depends both on the scan trajectory and the acquired data. Element (j,k) of $\mathbf{F}^H\mathbf{F}$ is defined as $Q(x_j - x_k)$, where $Q(x)$ is given by Eq. 2. The overwhelming bulk of the algorithm's computation occurs during the precomputation of $Q(\mathbf{x})$ and $\mathbf{F}^H\mathbf{d}$ (see Eqs. 2 and 3) rather than during the iterations of the linear solver.

$$\mathbf{F}^H\mathbf{F}\rho = \mathbf{F}^H\mathbf{d} \tag{1}$$

$$Q(\mathbf{x}_n) = \sum_{m=1}^{M} |\phi(\mathbf{k}_m)|^2 \, e^{(i2\pi \mathbf{k}_m \cdot \mathbf{x}_n)} \tag{2}$$

$$\left[\mathbf{F}^H\mathbf{d}\right]_n = \sum_{m=1}^{M} \phi^*(\mathbf{k}_m)\mathbf{d}(\mathbf{k}_m) e^{(i2\pi \mathbf{k}_m \cdot \mathbf{x}_n)} \tag{3}$$

These precomputations are usually approximated using a gridding-type technique [25], [20], [26]. However, accurately approximating these precomputations becomes computationally difficult, particularly as the dimensionality of the problem increases beyond the typical 2D images to which this technique has been previously applied. In this work, we demonstrate the feasibility of computing these precomputations *exactly* using the GPU, enabling practical use of this algorithm for the large-scale 3D and 4D reconstruction problems present in MRI.

In summary, the advanced LS algorithm performs optimal reconstruction of data obtained from non-Cartesian scan trajectories, which are advantageous in many applications of MRI. Over 99.9% of the algorithm's computation is devoted to precomputing the quantities Q and $\mathbf{F}^H\mathbf{d}$. However, given a reconstruction problem of N pixels and M scan data points the precomputations of Q and $\mathbf{F}^H\mathbf{d}$ have O(MN) complexity, compared to O(NlogN) complexity for algorithms based on gridding and the FFT. Our work demonstrates that these precomputations can be performed quickly and efficiently on modern GPUs. This increased computational efficiency makes the advanced reconstruction of non-Cartesian scan data practical.

III. Reconstruction of Non-Cartesian Data

The advanced MR image reconstruction algorithm described in Section II-B consists of three steps: computing Q (which depends only on the scan trajectory), computing $\mathbf{F}^H\mathbf{d}$ (which depends on the scan trajectory and on the scan data), and finding the image via a linear solver. In the CPU-based implementation over 99.9% of the algorithm's runtime occurs during the computation of Q and $\mathbf{F}^H\mathbf{d}$. We therefore choose to accelerate the computation of Q and $\mathbf{F}^H\mathbf{d}$ on the GPU, leaving the iterative solver to run on the CPU. The remainder of this section describes the algorithms used to compute Q and $\mathbf{F}^H\mathbf{d}$ and the implementations of those algorithms on the GPU. Because the two algorithms are nearly identical, we describe only Q's algorithm in detail.

A. *Computing* Q

As Figure 3(a) shows, the algorithm for Q is an excellent candidate for acceleration on the GPU because it is embarrassingly data-parallel. The algorithm first computes the magnitude-squared of ϕ at each data point in the trajectory space (k-space), then computes the real and imaginary components of Q at each point in the image space. The value of Q at any point in the image space depends on the values of every data point, but no elements of Q depend on any other elements of Q. Therefore, all elements of Q can be computed independently and in parallel.

Despite the algorithm's inherent parallelism, potential performance bottlenecks are evident. First, in the

```
for (K = 0; K < numK; K++)
  phiMag[K] = rPhi[K]*rPhi[K] +
              iPhi[K]*iPhi[K];

for (X = 0; X < numX; X++) {
  for (K = 0; K < numK; K++) {
    exp = 2*PI*(kx[K] * x[X] +
                ky[K] * y[X] +
                kz[K] * z[X]);
    rQ[X] += phiMag[K]*cos(exp);
    iQ[X] += phiMag[K]*sin(exp);
  }
}
```

(a)

```
for (K = 0; K < numK; K++) {
  rRho[K] = rPhi[K]*rD[K] +
            iPhi[K]*iD[K];
  iRho[K] = rPhi[K]*iD[K] -
            iPhi[K]*rD[K];

for (X = 0; X < numX; X++) {
  for (K = 0; K < numK; K++) {
    exp = 2*PI*(kx[K] * x[X] +
                ky[K] * y[X] +
                kz[K] * z[X]);
    cArg = cos(exp);
    sArg = sin(arg);
    rFH[X] += rRho[K]*cArg -
              iRho[K]*sArg;
    iFH[X] += iRho[K]*cArg +
              rRho[K]*sArg;
  }
}
```

(b)

```
__global__ void cmpPhiMag(float* rPhi, iPhi, phiMag, int numK) {
  int K = blockIdx.x * PHI_THREADS_PER_BLOCK + threadIdx.x;
  if (K < numK) {
    float real = rPhi[K];
    float imag = iPhi[K];
    phiMag[K] = real*real + imag*imag;
  }
}

__global__ void cmpQ(float* gx, gy, gz, grQ, giQ, int numK) {
  int globalX = blockIdx.x * Q_THREADS_PER_BLOCK + threadIdx.x;

  // register allocate image-space inputs and outputs
  x = gx[globalX];  y = gy[globalX];  z = gz[globalX];
  rQ = grQ[globalX];  iQ = giQ[globalX];

  for (int K = 0; (K < K_ELEMS_PER_GRID); K++) {
    // cKx, cKy, cKz, and cPhiMag are held in constant memory
    float exp = 2 * PI * (cKx[K] * sX[X] +
                          cKy[K] * sY[X] +
                          cKz[K] * sZ[X]);
    rQ += cPhiMag[K] * cos(exp);
    iQ += cPhiMag[K] * sin(exp);
  }

  grQ[globalX] = rQ;
  giQ[globalX] = iQ;
}
```

(c)

Fig. 3. Advanced MRI reconstruction algorithm based on least-squares (LS) optimality. Panels (a) and (b) show pseudocode for the algorithms that compute Q and $\mathbf{F}^H\mathbf{d}$, respectively. Panel (c) depicts the implementation of the Q algorithm on the GPU.

loop that computes the elements of Q, the ratio of floating-point operations to memory accesses is only 3:2. Thus, the GPU-based implementation of the algorithm must conserve memory bandwidth and tolerate memory latency. Second, the ratio of FP arithmetic to FP trigonometry is only 5:1. Thus, GPU-based implementation must tolerate or avoid stalls due to long-latency sin and cos operations.

The GPU-based implementation of the algorithm that computes Q (see Figure 3(c)) uses the G80's constant memory caches to shatter the potential bottleneck posed by memory bandwidth and latency. To overcome the memory bottleneck, the computation is divided into many CUDA grids,[1] with each CUDA grid operating on a distinct set of data values, i.e. a *data tile*. For each CUDA grid, the host CPU loads the data tile into constant memory before invoking the kernel. Each thread in the CUDA grid then computes a partial sum for a single element of Q by iterating over all the points in the data tile. This optimization increases the ratio of FP operations to global memory accesses dramatically.

Likewise, the special functional units (SFUs) enable the algorithm to avoid the potential bottleneck of long latency trigonometric operations. When the use_fast_math compiler option is invoked, the sin and cos operations are not linked to long-latency library calls, but rather are executed as individual, low-latency instructions on the SFUs. The speed of the SFU comes at the expense of some loss in accuracy when the argument to the sin or cos is very small, but, as we show in Section V, this optimization has negligible impact on the overall accuracy of the algorithm.

[1]To avoid confusion with MRI reconstruction via interpolation (gridding), we refer to the unit of computation performed during a kernel invocation as a *CUDA grid*.

B. *Computing* $\mathbf{F}^H\mathbf{d}$

As Figure 3(b) shows, the algorithm for $\mathbf{F}^H\mathbf{d}$ is nearly identical to the algorithm for Q. The GPU-based implementation of the $\mathbf{F}^H\mathbf{d}$ algorithm is precisely analogous to the implementation of the Q algorithm, as are the potential bottlenecks and the techniques used to overcome those bottlenecks. The primary difference between the two algorithms is that the $\mathbf{F}^H\mathbf{d}$ kernel performs an additional four FP computations, increasing the kernel's register usage by 25% to 50% over the Q kernel, depending on the loop unrolling factor.

IV. Methodology

To quantify the effects of the G80's architectural features on the performance and quality of the reconstruction, we implemented several versions of the advanced algorithm. The unoptimized version (UNOPT) simply executes in data-parallel fashion on the GPU, leveraging neither the constant memory nor the special functional units (SFUs). Another version (CMEM) uses the constant memory caches to overcome the bottleneck imposed by memory bandwidth and latency. In addition to using the constant memory caches, a third version (CMEM_SFU) uses the SFUs to compute fast, approximate versions of the sin and cos operations. Finally, a fourth version (CMEM_SFU_EXP) also uses experiment-driven code transformations to balance resource consumption, thereby improving the algorithm's utilization of the GPU.

To obtain a reasonable baseline, we implemented two versions of the advanced algorithm on the CPU. The first version (CPU_DP) uses double-precision for all floating-point values and operations, while the second version (CPU_SP) uses single-precision. Both CPU versions are coded with SSE intrinsics and linked with the AMD Core Math Library [27], which provides fast, approximate implementations of the sin and cos functions. Experiments (not shown) indicate that these optimizations increase the performance of the CPU implementations by a factor of 4X-6X.

To facilitate comparison of the advanced reconstruction based on least-squares (LS) optimality with the conventional reconstruction based on gridding and the FFT, we also ran two versions of the conventional reconstruction. The first version is coded in C, uses the FFTW library, and is highly optimized. We use this version to report the runtime of the conventional reconstruction. The second version, coded in MATLAB, is considerably less efficient, but affords more flexibility for comparing the image quality against the quality of the images produced by the advanced reconstruction. We use this version for image quality comparisons.

All reconstructions are performed on data obtained from a non-uniform, three-dimensional scan of a human brain. The scan data was obtained using a 9.4 Tesla scanner, and the imaged atoms are sodium, which exist in the brain in much lower concentrations than the hydrogen atoms that are typically scanned. There are 3.2M data points in the data set, and the image is reconstructed with 64 pixels in each dimension, for a total of 256K pixels.

The conventional reconstructions and the CPU-based implementations of the advanced reconstruction are executed on one core of a 2.66 GHz Core 2 Extreme quad-core CPU, which has a peak theoretical capacity of 10.6 GFLOPS per core. The system has a 4MB L2 cache and 4GB of main memory. The GPU-based implementations of the advanced reconstruction are executed on a 1.35 GHz GeForce 8800 GTX using CUDA version 0.8. The G80, which has a peak theoretical capacity of 345.6 GFLOPS, is housed in a system with a 2.4 GHz Core 2 dual-core CPU, 4MB of L2 cache, and 3GB of main memory. The final stage of the advanced reconstruction (the linear solver) has not been ported to the GPU, and is therefore executed on the quad-core CPU.

Throughout the following section, we make a subtle yet important distinction between *reconstruction time* and *algorithmic runtime*. Reconstruction time refers to the time required to reconstruct an image, which includes the time to compute $\mathbf{F}^H\mathbf{d}$ and the time to run the linear solver. The computation of Q is excluded from the reconstruction time because Q depends only on the scan trajectory and therefore be computed once, before the data is acquired, and then used during the reconstructions of many images. By contrast, algorithmic runtime includes the time to compute $\mathbf{F}^H\mathbf{d}$ and Q and the time to run the linear solver.

TABLE I

Performance of LS-based and FFT-based MRI reconstructions. The LS-based reconstruction times include the time to compute $\mathbf{F}^H\mathbf{d}$ and the time to run the linear solver. The computation of Q is excluded from the reconstruction time because Q depends only on the scan trajectory and can therefore be computed before the scan data is acquired.

Arch	Algorithm	Recon Time (m)
CPU (DP)	LS	519.59
CPU (SP)	LS	343.91
GPU	LS	3.19
CPU	FFT	0.39

V. Evaluation

To be useful in clinical settings, the LS-based reconstruction should exhibit runtime comparable to conventional reconstruction algorithms currently used in clinical settings. Furthermore, the GPU-accelerated implementation of the LS reconstruction (GPU-LS) should not sacrifice accuracy relative to CPU-based implementations of the same algorithm. Our experiments indicate that the GPU-accelerated, LS-based algorithm meets both of these criteria. As Table I shows, GPU-LS reconstructs the image in just over 3 minutes, while the conventional algorithm based on gridding and FFTs reconstructs the image in 23 seconds. While the gridded reconstruction is clearly faster, the speed of the GPU-LS reconstruction is sufficient for key MRI applications in clinical and research settings.

With regards to accuracy, the advanced reconstruction and the gridded reconstruction produce images that are, in this case, visually indistinguishable. Figure 4 shows the same 2D slice from several of the 3D reconstructions. The superiority of the LS-based algorithm would be better demonstrated at higher resolution or on a different data set.

In terms of algorithmic runtime, the GPU-accelerated algorithm achieves a net speedup of 283x over the CPU-based implementation of the same algorithm. To put this result into perspective, the GPU-based algorithm completes in roughly 10 minutes, while the CPU-based algorithm finishes in roughly 2 days. Clearly, the CPU-based implementation of the advanced reconstruction algorithm is not a viable option. As Table II shows, the GPU accelerates the computation of Q and $\mathbf{F}^H\mathbf{d}$ by 358x and 222x, respectively.

The remainder of this section describes how the implementation of GPU-LS leverages the G80's resources to achieve such impressive performance. We find that the constant memory caches are quite effective

TABLE II

Impact of optimizations on GPU-based computation of Q and $F^H d$. The speedup column lists the speedup achieved by this version of the algorithm over the version listed on the preceding row. The first three implementations of the Q and $\mathbf{F}^H \mathbf{d}$ algorithms, which do not use experiment-driven code transformations, have 128 threads per block and a tiling factor of 256. The inner loops are not unrolled. The experimentally optimized implementations used the highest-performing combination of settings for threads per block, tiling factor, and loop unrolling factor, as determined by an exhaustive search.

	Total	Q			$F^H D$		
Optimizations	Runtime (m)	Runtime (m)	GFLOPS	Incremental Speedup (X)	Runtime (m)	GFLOPS	Incremental Speedup (X)
CPU_DP	4528.6	4009.0	0.3	N/A	518.0	0.4	N/A
CPU_SP	3022.6	2678.7	0.5	1.5	342.3	0.7	1.5
GPU_UNOPT	302.9	260.2	5.1	10.3	41.0	5.4	8.3
GPU_CMEM	83.4	72.0	18.6	3.6	9.8	22.8	4.2
GPU_CMEM_SFU	17.6	13.6	98.2	5.3	2.4	92.2	4.0
GPU_CMEM_SFU_EXP	10.7	7.5	178.9	1.8	1.5	144.5	1.6

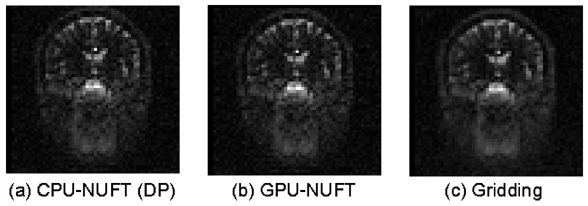

Fig. 4. Reconstructed images. The images reconstructed by the various GPU-based implementations of the advanced reconstruction algorithm are visually indistinguishable.

in reducing the number of accesses to global memory, while the special functional units provide substantial acceleration for the trigonometric computations in the algorithm's inner loops. We also find that experiment-driven code transformations have a significant impact on the algorithm's performance. Specifically, the algorithm's performance is heavily dependent on the tiling factor, the number of threads per block, and the inner loop unrolling factor. Optimal values for these parameters can be discovered using an automated experimentation.

A. No Optimizations

As Table II shows, when computing Q and $\mathbf{F}^H \mathbf{d}$ the unoptimized version of GPU-LS (GPU_UNOPT) achieves speedups of 10.3X and 8.3X, respectively, over the single-precision CPU version (CPU_SP). Because the implementations of Q and $\mathbf{F}^H \mathbf{d}$ are so similar, we restrict our discussion to the Q kernel. In this unoptimized version, the inner loop is not unrolled. There are 128 threads per block, and 256 data points are processed by each CUDA grid. Given these parameters, each thread uses 15 registers. Therefore, up to 8192/15 = 546 threads can execute on each SM simultaneously. However, given the granularity of 128 threads per block, the number of threads simultaneously executed by each SM is actually 4*128 = 512, which represents 67% utilization of the G80's cores.

Because the kernel leverages neither the constant memory nor the shared memory, memory bandwidth and latency are significant performance bottlenecks. With two 4-byte global memory accesses for every three FP operations, and with memory bandwidth of 86.4 GB/s, the upper limit on the kernel's performance is only 32.4 GFLOPS. Due to other performance bottlenecks, the kernel actually achieves only 5.2 GFLOPS. Nevertheless, the kernel's performance represents a substantial improvement over the CPU-based implementations.

B. Constant Memory

Again focusing discussion on the Q kernel, the GPU_CMEM version achieves a speedup of 3.6X over the unoptimized version. The only difference between GPU_CMEM and GPU_UNOPT is that GPU_CMEM places each CUDA grid's data tile in constant memory rather than global memory, thereby receiving the benefits of each SM's 8KB, on-chip constant memory cache.

We now analyze the off-chip memory accesses on a single SM during the execution of four thread blocks. With 7 global memory accesses per thread, 128 threads per thread block, and 4 thread blocks per SM, there are 3,584 accesses to global memory. Assuming no constant memory cache evictions due to conflicts, there are also 1,024 accesses to constant memory (256 data points per tile, with 4 floating-point values per data element), yielding a total of 4,608 off-chip memory accesses. The number of floating-point computations performed by the 4 thread blocks is 4*128*256*12 = 1,572,864. Thus, the ratio of FP operations to off-chip memory accesses has increased from 3:2 to 341:1. However, the GPU_CMEM version still achieves only 18.6 GFLOPS, which implies the existence of another bottleneck.

C. Special Functional Units

In computing Q, the GPU_CMEM_SFU version achieves speedup of 5.3X over the version that uses only the constant memory. In this case, the special functional units (SFUs) are responsible for increasing the algorithm's performance from 18.6 GFLOPS to 98.2 GFLOPS. When compiled without the use_fast_math compiler option, the kernel uses implementations of sin and cos provided by an NVIDIA math library. However, when compiled with the use_fast_math option, the sin and cos computations each execute as a single instruction on an SFU. As the images reconstructed by the versions that use the SFUs are visually indistinguishable from the images reconstructed by the versions that do not use the SFUs, we conclude that the SFU's approximate implementations of sin and cos have negligible impact on the reconstruction's accuracy for this data set.

D. Experimental Optimization

While the GPU_CMEM_SFU version overcomes the potential bottlenecks related to off-chip memory accesses and trigonometric computations, the algorithm still performs at only 98.2 GFLOPS, which is less than one-third of the G80's peak theoretical performance. There are two culprits: instruction mix and resource utilization. Without unrolling the inner loop, the Q kernel performs only 12 FP computations for every 26 overhead instructions (such as memory accesses and integer instructions). When the loop is unrolled 16 times, the percentage of floating-point instructions rises from 32% to 60%. However, the per-thread register usage also rises from 11 to 30. Because the number of threads that can execute simultaneously is inversely proportional to the number of registers per thread, the loop unrolling optimization must carefully balance the competing goals of increasing the percentage of FP instructions and maintaining high utilization of the G80's cores.

To determine the potential performance impact of experiment-driven code transformations, we conducted an exhaustive search that varied the number of threads per block from 64 to 512, the tiling factor from 32 to 2,048, and the loop unrolling factor from 1 to 16. Each parameter was varied by powers of 2. For reference, the GPU_CMEM_SFU kernels performed no loop unrolling and set the number of threads per block and the tiling factor to values near the center of search space. The data set used during experimental optimization was roughly 20X smaller than the data set used for the reconstructions reported in this paper. For the Q algorithm, the experiment-driven optimizer selects 64 threads per block, a tiling factor of 2,048, and a loop unrolling factor of 16. For the $\mathbf{F}^H\mathbf{d}$ algorithm, the optimizer selects 64 threads per block, a tiling factor of 64, and a loop unrolling factor of 8. The optimizer increases the performance of the Q and $\mathbf{F}^H\mathbf{d}$ kernels to 179 GFLOPS and 145 GFLOPS, respectively, representing increases of 82% and 57% over the GPU_CMEM_SFU kernels.

VI. Related Work

Medical imaging was one of the first GPGPU applications, with computed tomography (CT) reconstruction achieving a speedup of two orders of magnitude on the SGI RealityEngine in 1994 [28]. A wide variety of CT reconstruction algorithms have since been accelerated on the GPU [3], [29], [30], [31]. In [31] the GPU is used to accelerate Simultaneous Algebraic Reconstruction Technique (SART), an algorithm that increases the quality of image reconstruction relative to the conventional filtered backprojection algorithm. SART, which is not typically used in clinical settings because it requires significantly more computation than backprojection, becomes a viable clinical option when executed on the GPU.

By contrast, MRI reconstruction on the GPU has not been studied extensively. Research in this area has focused on accelerating the fast Fourier transform (FFT), which is a key component of many MRI reconstruction algorithms. Speedups on the order of 2x-9x have been achieved [32], [33], [34].

VII. Conclusions and Future Work

The computational resources, architectural features, and programmability of the GeForce 8800 GTX reduce the time for an optimal reconstruction of non-uniform MRI scan data from six hours to three minutes, making the algorithm practical for many clinical and research applications. This capability will also allow for the practical application of more advanced algorithms that incorporate other physical effects into the image reconstruction, resulting in more accurate and informative imaging techniques. However, there is still much work to be done. The linear solver, which now accounts for 50% of the reconstruction time, becomes an important candidate for acceleration. Also, a more rigorous analysis of the accuracy of the advanced reconstruction and the gridded reconstruction is warranted. Phantom images, for which the exact reconstruction is known a priori, are often used for this purpose in MRI research. Finally, the optimizations that enabled the reconstruction to achieve over 150 GFLOPS on the

GPU, including data tiling and experiment-driven code transformation, should now be applied to CPU-based implementations. Optimizing the CPU-based reconstruction should provide a faster baseline and a better understanding of the architectural features that make the advanced reconstruction feasible.

VIII. Acknowledgments

The authors wish to thank David Kirk and John Nickolls at NVIDIA, as well as Matthew Frank and Shane Ryoo at the University of Illinois, for their feedback and invaluable insights. We also thank Keith Thulborn and Ian Atkinson of the Center for MR Research at the University of Illinois at Chicago for providing MRI scan data and the gridded MRI reconstruction. We acknowledge the support of the Gigascale Systems Research Center, one of five research centers funded under the Focus Center Research Program, a Seminconductor Research Corporation program. This material is based on work supported under two National Science Foundation Graduate Research Fellowships (Sam Stone, Justin Haldar). Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. This work was performed with generous hardware loans from NVIDIA and donations from Intel, and was supported in part by the following research grants: NIH-P41-EB03631-16 and NIH-R01-CA098717.

References

[1] "AMD Stream Processor," http://ati.amd.com/products/ streamprocessor/index.html.

[2] "NVIDIA CUDA," http://developer.nvidia.com/object/ cuda.html.

[3] K. Mueller, F. Xu, and N. Neophytou, "Why do commodity graphics hardware boards (GPUs) work so well for acceleration of computed tomography?" in *SPIE Elec. Imag. 2007, Computational Imaging V Keynote*, 2007.

[4] S. Green, "GPU Physics," SIGGRAPH 2007 GPGPU Course. http://www.gpgpu.org/s2007/slides/15-GPGPU-physics.pdf.

[5] P. Trancoso and M. Charalambous, "Exploring graphics processor performance for general purpose applications," in *Euromicro Symposium on Digital System Design, Architectures, Methods, and Tools (DSD 2005)*, 2005.

[6] M. Segal and K. Akeley, *The OpenGL Graphics System: A Specification (Version 2.0)*, Silicon Graphics, Inc., October 2004.

[7] "DirectX Developer Center," http://www.msdn.com/directx/.

[8] "Cg," http://developer.nvidia.com/page/cg_main.html.

[9] I. Buck, *Brook Specification v0.2*, October 2003.

[10] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data parallelism to program GPUs for general-purpose uses," in *Int'l Conference on Architectural Support for Programming Languagues and Operating Systems (ASPLOS-XII)*, 2006.

[11] J. Nickolls and I. Buck, "NVIDIA CUDA software and GPU parallel computing architecture," Microprocessor Forum, May 2007.

[12] P. C. Lauterbur, "Image formation by induced local interactions: Examples employing nuclear magnetic resonance," *Nature*, vol. 242, pp. 190–191, 1973.

[13] C. B. Ahn, J. H. Kim, and Z. H. Cho, "High-speed spiral-scan echo planar NMR imaging," *IEEE Trans. Med. Imag.*, vol. 5, no. 1, pp. 2–7, 1986.

[14] K. Scheffler and J. Hennig, "Frequency resolved single-shot MR imaging using stochastic k-space trjaectories," *Magn. Res. Med.*, vol. 35, no. 4, pp. 569–576, 1996.

[15] M. Lustig, J. M. Santos, J. H. Lee, D. L. Donoho, and J. M. Pauly, "Compressed sensing for rapid MR imaging," in *Proc. SPARS*, 2005.

[16] J. I. Jackson, C. H. Meyer, D. G. Nishimura, and A. Macovski, "Selection of a convolution function for Fourier inversion using gridding," *IEEE Trans. Med. Imag.*, vol. 10, no. 3, pp. 473–478, 1991.

[17] H. Schomberg and J. Timmer, "The gridding method for image reconstruction by Fourier transformation," *IEEE Trans. Med. Imag.*, vol. 14, no. 3, pp. 596–607, 1995.

[18] K. P. Pruessmann, M. Weiger, M. B. Scheidegger, and P. Boesiger, "SENSE: Sensitivity encoding for fast MRI," *Magn. Res. Med.*, vol. 42, no. 5, pp. 952–962, 1999.

[19] K. P. Pruessmann, M. Weiger, P. Börnert, and P. Boesiger, "Advances in sensitivity encoding with arbitrary k-space trajectories," *Magn. Res. Med.*, vol. 46, no. 4, pp. 638–651, 2001.

[20] J. A. Fessler, S. Lee, V. T. Olafsson, H. R. Shi, and D. C. Noll, "Toeplitz-based iterative image reconstruction for MRI with correction for magnetic field inhomogeneity," *IEEE Trans. Signal Process.*, vol. 53, no. 9, pp. 3393–3402, 2005.

[21] B. P. Sutton, D. C. Noll, and J. A. Fessler, "Fast, iterative image reconstruction for MRI in the presence of field inhomogeneities," *IEEE Trans. Med. Imag.*, vol. 22, no. 2, pp. 178–188, 2003.

[22] J. A. Fessler and B. P. Sutton, "Nonuniform fast Fourier transforms using min-max interpolation," *IEEE Trans. Signal Process.*, vol. 51, no. 2, pp. 560–574, 2003.

[23] J. P. Haldar, D. Hernando, M. D. Budde, Q. Wang, S.-K. Song, and Z.-P. Liang, "High-resolution MR metabolic imaging," in *Proc. IEEE EMBS*, 2007, pp. 4324–4326.

[24] J. P. Haldar and Z.-P. Liang, "High-resolution diffusion MRI," in *Proc. IEEE EMBS*, 2007, pp. 311–314.

[25] F. T. A. W. Wajer, "Non-Cartesian MRI scan time reduction through sparse sampling," Ph.D. dissertation, Technische Universiteit Delft, Delft, Netherlands, 2001.

[26] J. A. Fessler and D. C. Noll, "Iterative reconstruction methods for non-cartesian MRI," in *Proc. ISMRM Workshop on Non-Cartesian MRI*, 2007.

[27] Numerical Algorithms Group Ltd., Advanced Micro Devices, Inc., "AMD core math library (ACML) version 3.6.0 user guide," 2006.

[28] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *1994 Symposium on Volume Visualization*, 1994.

[29] X. Xue, A. Cheryauka, and D. Tubbs, "Acceleration of fluoro-CT reconstruction for a mobile C-Arm on GPU and FPGA hardware: A simulation study," in *SPIE Med. Imag. 2006*, 2006.

[30] K. Chidlow and T. Möller, "Rapid emission tomography reconstruction," in *Int'l Workshop on Volume Graphics*, 2003.

[31] K. Mueller and R. Yagel, "Rapid 3-D cone-beam reconstruction with the simultaneous algebraic reconstruction technique (SART) using 2-D texture mapping hardware," *IEEE Trans. Med. Imag.*, vol. 19, no. 12, pp. 1227–1237, 2000.

[32] T. Sumanaweera and D. Liu, "Medical image reconstruction with the FFT," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, Ed. Addison-Wesley, March 2005, pp. 765–784.

[33] T. Schiwietz, T. Chang, P. Speier, and R. Westermann, "MR image reconstruction using the GPU," in *SPIE Med. Imag. 2006*, 2006.

[34] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve, "Fourier volume rendering on the gpu using a split-stream FFT," 9th International Fall Workshop on Vision, Modeling, and Visualization, 2004.