Advanced MRI Reconstruction Toolbox with Accelerating on GPU

Xiao-Long Wu[*a], Yue Zhuo[c], Jiading Gai[b], Fan Lam[a], Maojing Fu[a], Justin P. Haldar[a], Wen-Mei Hwu[a], Zhi-Pei Liang[a], Bradley P. Sutton[a]

[a]Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1406 W. Green St. Urbana, IL 61801 USA, [b]Beckman Institute, University of Illinois at Urbana-Champaign, 405 North Mathews Avenue, Urbana, IL 61801 USA, [c]Bioengineering Department, University of Illinois at Urbana-Champaign, 1304 West Springfield Urbana, IL 61801 USA

ABSTRACT

In this paper, we present a fast iterative magnetic resonance imaging (MRI) reconstruction algorithm taking advantage of the prevailing GPGPU programming paradigm. In clinical environment, MRI reconstruction is usually performed via fast Fourier transform (FFT). However, imaging artifacts (i.e. signal loss) resulting from susceptibility-induced magnetic field inhomogeneities degrade the quality of reconstructed images. These artifacts must be addressed using accurate modeling of the physics of the system coupled with iterative reconstruction. We have developed a reconstruction algorithm with improved image quality at the expense of computation time and hence an implementation on GPUs achieving significant speedup. In this work, we extend our previous work on GPU implementation by adding several new features. First, we enable Sensitivity Encoding for Fast MRI (SENSE) reconstruction (from data acquired using a multi-receiver coil array) which can reduce the acquisition time. Besides, we have implemented a GPU-based total variation regularization in our SENSE reconstruction framework. In this paper, we describe the different optimizations employed from levels of algorithm, program code structures, and specific architecture performance tuning, featuring both our MRI reconstruction algorithm and GPU hardware specifics. Results show that the current GPU implementation produces accurate image estimates while significantly accelerating the reconstruction.

Keywords: MRI, GPU, SENSE, total variation regularization, field inhomogeneity, susceptibility.

1. INTRODUCTION

In clinical environment, MR image reconstruction is usually performed via fast Fourier transform (FFT) which offers an immediate reconstructed image available for the physician to review. Reconstruction with FFT, however, cannot correct imaging artifacts, such as susceptibility-induced magnetic field inhomogeneities. The field inhomogeneity is due to difference in susceptibility between air and tissue (e.g. in the orbito-frontal cortex), which creates a local magnetic field distorting the overall field and leads to signal loss and signal distortions thus hindering accurate diagnosis. To correct the imaging artifacts, we developed a reconstruction algorithm with improved image quality at some sacrifice on computation time. In order to enable clinical use of such advanced reconstruction methods, we have developed in our previous work [7-10] a fast MRI reconstruction toolbox making use of the latest advances in parallel programming via GPU. The previously proposed method aims at offering fast reconstruction of MR images while correcting for susceptibility artifacts. This is done by using a conjugate gradient algorithm with our advanced MR imaging model (include the magnetic field map and its gradients) regularized by a smoothness constraint implemented using sparse matrix representations. Previous results showed significant speedup while offering image quality enhancement.

In this work, we elaborate on our previous work by adding new features to our GPU MRI reconstruction toolbox. Recently, sensitivity encoding (SENSE) [6] has seen an increased interest due to its ability to significantly reduce acquisition time. SENSE reconstruction allows for combination of multi-coil signals during reconstruction. Using multiple receivers during the acquisition results in a parallel acquisition scheme, therefore reducing the time needed to acquire the same number of k-space samples. SENSE reconstruction further makes use of data acquired by each coil and combines them by using a SENSE weighting map. Here, we integrate SENSE reconstruction into our GPU reconstruction algorithm correcting for magnetic susceptibility artifacts. This addition to our toolbox can potentially have a great impact since it allows for combination of fast acquisition and its corresponding reconstruction with

*xiaolong@illinois.edu

correction for magnetic susceptibility artifacts (which get more severe with higher magnetic field scanners). Allowing reconstruction of 256x256 data in about half a second could be a major step for clinical applications.

In addition, this paper summarizes the optimization techniques used in the latest version of our toolbox. In this context, it can be seen as a set of general guidelines for translation of reconstruction algorithm to CUDA programming. Current optimization techniques described in this work include memory management (e.g. use of memory coalescence), branch reduction, etc.

2. METHODS

Our reconstruction toolbox is currently implemented to solve the following image reconstruction problem:

$$\hat{\mathrm{f}} = \arg\min_{\mathrm{f}} \frac{1}{2} \| \tilde{\mathrm{F}}\mathrm{f} - \tilde{\mathrm{Y}} \|_2^2 + \beta \| \mathrm{WCf} \|_2^2 \tag{1}$$

Where **f** is the target image to be reconstructed, $\tilde{\mathbf{Y}}$ is the measured data, and $\tilde{\mathbf{F}}$ is a matrix modeling the data acquisition process. The first term of the cost function in Eqn. (1) is usually referred as the data consistency constraints, while the second term is a regularization term. Regularization is important if minimizing the data consistency term alone does not guarantee sufficient well-posedness of the inverse problem. The matrix **C** is a regularization matrix, **W** is an optional weighting matrix and β is a regularization parameter obtained by tuning which different trade-offs between the regularization term and the data consistency term. The use of **W** enables advanced reconstruction formulations incorporating different kinds of regularization functions (e.g., l_1 regularization or total variation regularization [11]). The implementation of the software in solving various kinds of reconstruction problems will be presented.

2.1 SENSE reconstruction for parallel imaging with multi-receivers

Parallel imaging is performed by placing an array of receiver coils around the object to be imaged, with each receiver coil lending spatially distinct reception profiles to the acquired data sets [6]. Considering the following formulation of the complex baseband signal during an MRI experiment

$$y(t) = \int f(r)\, S(r)\, e^{-iw(r)t} e^{-i2\pi k(t) r} dr + \varepsilon(t) \tag{2}$$

where $f(r)$ is a continuous function of the object's traverse magnetization at location r; $S(r)$ is the spatial sensitivity of the receiver coil; $\omega(r)$ is the field inhomogeneity present at r; $k(t)$ is the input data trajectory at time t; $\varepsilon(t)$ is the noise term. Looking at Eqn. (2) we can see that the signal equation includes coil sensitivity information. The data from multiple coils can be combined together and each coil has its own sensitivity incorporated into a larger system of equations. With each coil receiving its own data, weighted by $S_l(r)$, where l = 1, ..., L, i.e. the complex spatial sensitivity profile of coil l, the parallel imaging model in matrix form can be represented as follows (with L = 3 as an example):

$$\underbrace{\begin{bmatrix} \mathrm{Y}_1 \\ \mathrm{Y}_2 \\ \mathrm{Y}_3 \end{bmatrix}}_{\tilde{\mathrm{Y}}} = \underbrace{\begin{bmatrix} \mathrm{F}\cdot S_1 \\ \mathrm{F}\cdot S_2 \\ \mathrm{F}\cdot S_3 \end{bmatrix}}_{\tilde{\mathrm{F}}} \mathrm{f} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \end{bmatrix},$$

where $\mathbf{Y}_l$ is the signal vector received from coil l and $\tilde{\mathbf{Y}}$ is formed by stacking $\mathbf{Y}_l$'s into a single column; S_l is the diagonal matrix holding the complex spatial sensitivity profiles on the diagonal entries from the l^{th} coils; $\tilde{\mathbf{F}}$ denotes the parallel imaging augmented system matrix.

Inclusion of parallel imaging into the GPU framework is simple. Using the image reconstruction formulation as in Eqn. (1), the two computations that we are interested in computing: $\tilde{\mathbf{F}}$ and $\tilde{\mathbf{F}}^{\mathbf{H}}$ (denoted as the forward operator and the adjoint operator). We start with the computation of $\tilde{\mathbf{F}}$,

$$\tilde{\mathrm{F}}\mathrm{f} = \begin{bmatrix} \mathrm{F} \cdot S_1 \\ \mathrm{F} \cdot S_2 \\ \mathrm{F} \cdot S_3 \end{bmatrix} \mathrm{f}\,, \tag{3}$$

In Eqn. (3), again using L = 3 as an example, $\tilde{\mathbf{F}}\mathbf{f}$ can be viewed as a column vector concatenated from L sub-vectors, each of which is obtained by first performing a matrix-vector multiplication between the input **f** the coil sensitivity profile S_l, followed by applying the single-coil forward operator on the resulting matrix-vector product. Note that multiplications with the S_l matrices are trivial as they are diagonal matrices. In order to minimize the memory usage of GPU, we represent S_l as a 1D array encoding only its diagonal elements as opposed to the full 2D matrix.

The second bottleneck computation that is required by the CG algorithm is the calculation of the adjoint operator $\tilde{\mathbf{F}}^{\mathbf{H}}$. By going through some algebra, we have:

$$\begin{aligned} \tilde{\mathrm{F}}^{\mathrm{H}}\tilde{\mathrm{Y}} &= \left[S_1^H \cdot \mathrm{F}^H \quad S_2^H \cdot \mathrm{F}^H \quad S_3^H \cdot \mathrm{F}^H \right] \tilde{\mathrm{Y}} \\ &= \sum_{l=1}^{L=3} S_l^H \cdot \mathrm{F}^H \mathrm{Y}_l \end{aligned} \tag{4}$$

The adjoint operator is calculated as a sum of L terms. Each $S_l^H \cdot \mathbf{F}^H \mathbf{Y}_l$ term is again evaluated in two steps as $S_l^H \cdot \left(\mathbf{F}^H \mathbf{Y}_l \right)$, which can be accomplished by first applying the single-coil adjoint operator on $\mathbf{Y}_l$ and then computing the matrix multiplication between the result and the sensitivity matrix.

2.2 Preparation of GPU implementation for SENSE reconstruction

We complete Section 2 with the description of the GPU implementation of our iterative approach to a SENSE reconstruction. A GPU version of the forward/adjoint operators under the single-coil scenario has been implemented in our previous work [7-10]. Here we extend them to multicoil cases by writing a wrapper around the two operators that loops through all coils in a dataset (illustrated below in **Algorithm 1** and **Algorithm 2**). Since, in the GPU code, the coil sensitivity matrix S_l is represented as a 1D vector containing only the non-zero components on the diagonal, all the matrix-vector multiplications involving S_l are reduced to pointwise vector-vector multiplications implemented by the function 'pointMultGpu'. Inside 'pointMultGpu', each thread performs one complex multiplication on one pair of data points in the two input vectors and each pair of points are pre-loaded into registers from global memory prior to the actual computation to minimize the global memory access.

```
Input: The sensitivity matrix $S_l$ and the input vector f.
Output: 1D array B of length L×K, where K denotes the k-space size per coil.
foreach Coil l=1:L do
    tmp = pointMultGpu($S_l$, f);
    B[$l \cdot$ K : $(l + 1) \cdot$ K] = ForwardGpu(tmp);
End
```

Algorithm 1: Forward operator pseudocode for parallel imaging

```
Input: The sensitivity matrix $S_l$ and the input vector $\tilde{\mathbf{Y}}$.
Output: 1D array D of length N, where N denotes the image size.
foreach Coil l =1:L do
    tmp = AdjointGpu($\tilde{\mathbf{Y}}$ [$l \cdot$ K : $(l + 1) \cdot$ K]);
    D += pointMultGpu($S_l^H$, tmp);
end
```

Algorithm 2: Adjoint operator pseudocode for parallel imaging

Figure 1 Algorithms of GPU implementation for SENSE reconstruction

2.3 Total Variation Regularization

In certain types of applications, sufficient data for high quality reconstruction will not be available due to imaging time constrains. Hence, proper regularization is essential for obtaining high quality reconstructions from the limited data of the target image to be reconstructed. Total variation (TV) regularization is one of the most widely used techniques due to the recent development of compressed sensing based reconstruction techniques [12]. We implement an algorithm for solving TV-regularized reconstruction based on solving the problem in Eqn. (1) with different matrix W multiple times [11]. In this case, C is a dual direction finite difference operator which can be expressed as $C = [DH ; DV]^H$, where DH and DV denote the finite difference of every pixel pair along the horizontal and vertical directions of the image **f** respectively.

3. TRANSFORMATION AND OPTIMIZATIONS

The NVIDIA hardware architecture provides tremendous parallel computation power and memory bandwidth, increased by an order of magnitude compared to a modern CPU. The Intel Core2 Duo has 32 GFLOPS computation power and 8.4 GB/s memory bandwidth. Yet the current prevalent GTX 280 GPU provides 933 GFLOPS with 141.7 GB/s memory bandwidth. However, behind these exciting numbers are the many-core SIMD (Single-Instruction, Multiple-Data) hardware architecture and the Non-Uniform Memory Architecture (NUMA). There are 240 processor cores on a GTX 280 device for multi-thread processing. These 240 cores are grouped into 30 Stream Multiprocessors (SM), each of which is composed of 8 cores as an execution unit. 32 threads are executed as a whole, called *warp*, on an SM. Besides, the CUDA memory hierarchy provides a handful of memory types as compared to the dual-memory system on a CPU, i.e., registers and DRAM memory. For applications fitting this new hardware architecture well, the performance enhancement is expected to be huge. To fully take advantage of this architecture, programmers must tailor their applications to the proposed CUDA programming model by NVIDIA.

Before fitting to this parallel programming model, foremost a program must be analyzed from different levels of abstractions, such as coding styles and program semantics. And not every application or code segment suits CUDA. Since the CUDA language is an extension of C, the code segments not suitable for CUDA can still be executed by a CPU processor. Therefore, programmers need to know which code segments are best suitable for the GPU and which are for the CPU in order to fully utilize both hardware computation powers. Basically, the very first step is to identify the hot spots in an application program. Hot spots mean, the program code segments take the most of the execution time of the whole application. Then, programmers can start the analysis on these hot spots and see if they can be transformed into CUDA or, instead, optimized for the CPU architecture.

The transformation process from a high-level language into a lower-level one involves various optimizations at different phases. During our transformation, the C program serves as a performance baseline for its counterpart CUDA program which is further optimized for the target GPU architecture. This section will enumerate the considerations and optimizations we take during the transformation phases from a plain CUDA program to an optimized one.

3.1 Optimizations of the CUDA Program

Performance optimization on the GPU architecture is a multi-metric optimization problem [4] or called a non-linear optimization problem. And the transformed code achieving the optimal performance can be very different. To achieve the best performance for a given application, programmers need to be familiar with three things. Firstly, they need to know the GPU hardware features such as SPMD (Single-Program, Multiple-Data) and resource constraints like register number of the given device generation. By this they can avoid incontinent use of resources and the wrong mapping between the program and the machine. Secondly, they must understand their application at the algorithm level, software structure level, and statement level. Then they can gain parallelisms at these levels by transforming the application to match the hardware characteristics. Finally, they must be equipped with the optimization skills and strategies such that they can efficiently achieve the optimal performance instead of through exhaustive try-and-error.

Table 1 is the summarized optimization strategies from [3] we conclude from this work. Although we haven't applied all the strategies, for completeness we propose this table can be a guideline for optimizing other applications on GPU. The optimization issues can be categorized into optimizations across threads (horizontal point of view) and those within threads (vertical point of view). Horizontal optimizations can introduce warp, block, memory, and kernel level parallelisms. Vertical optimizations can add instruction and data level parallelisms. The goal of the performance tuning on GPU is to achieve the optimal parallelism for a given application. To achieve this goal, a complete understanding of the three things is a must, i.e., machine architectures, application characteristics, and performance optimization strategies. Below we enumerate the key transformations.

Table 1 Optimization strategies for performance tuning on GPU

Optimization issues	**Compute-bound**	**Memory-bound**	
	Reduce instructions	**Hide memory latency**	**Reduce memory bandwidth**
Optimizations across threads (Horizontal)	Manual warp scheduling (Avoid divergence)		Tiled compute (Using high-bandwidth memory) Memory layout transformation (Coalescing/Bank conflicts/Access patterns)
Optimizations within threads (Vertical)	Common subexpression elimination (Using registers) Reduce branches (Loop unrolling)	Automatic instruction scheduling (Loop unrolling) Manual instruction scheduling (Data prefetching)	Reuse data (Using high-bandwidth memory)

Using high-bandwidth memory for reused data: Using the registers, shared memory, or constant memory to store frequently reused data can save memory bandwidth to the global memory and thus improves the performance. But using too many registers can cause a loss on thread occupancy and potentially lose warp level parallelism. And sometimes compilers will use local memory (global memory) for overuse of registers. Using too much shared memory or constant memory can cause kernel launch failure or compilation error. As for choosing which memory, it depends on the application characteristics like the data access type, size, and usage frequency.

Figure 2 demonstrates the techniques on the use of registers, loop invariant code motion, and common subexpression elimination. Arrays v1 and v2 are stored in global memory. The code on the left-hand side makes (N*M)+(N*M) memory load requests. We can see array v1 is reused M times in the inner-most loop. Therefore we can use a register "temp", as shown on the right-hand side, to eliminate redundant memory loads of v1 elements. Also it's moved to the outer loop. The code on the right-hand side makes N+(N*M) memory loads. In addition, cos(sum) is executed twice in the inner loop of the left-hand side code segment. It can be reduced to one as listed on the right-hand side without using additional registers.

```
// Without using registers
for (k=0; k<N; i++){
    for (i=0; i<M; i++) {
       sum += v1[k]*v2[i];
       expr1 = cos(sum);
       expr2 = cos(sum)+sin(sum);
       … = expr1 …;
    }
}
```

```
// Using registers
for (k=0; k<N; i++){
    temp = v1[k];
    for (i=0; i<M; i++) {
        sum += temp*v2[i];
        expr1 = cos(sum);
        expr2 = expr1+sin(sum);
        … = expr1 …;
    }
}
```

Figure 2 Code segments with and without using registers

Manual warp scheduling: CUDA language is a multi-thread parallel programming language in the concept of SPMD. Based on this concept and the given SM-centric hardware architecture, every thread inside the same warp executes the same instruction with or without the same data. Hence, the branch instructions, like if-else and loop bound check, can diverge the thread execution path when the threads are grouped in the same warp. Such situation is usually called branch divergence.

The left code snippet of Figure 3 delineates one example of this situation. A half warp will go one way and at the same time the other half warp is waiting. When the first half warp finishes, it waits for the other half warp and both move on to the following instruction together. In a sense, it doubles the execution time as compared to the situation without branch divergence. The right code snippet of Figure 3 demonstrates a different coding style, where even warps will execute the if-part path and odd warps will go the other way.

```
if (threadIdx.x % 2 == 0) {
   // if-part thread execution path
} else {
   // else-part thread execution path
}
// if-part threads must wait for the
// else-part threads
```

```
if ((threadIdx.x / WARP_SIZE) % 2 == 0) {
   // if-part warp execution path
} else {
   // else-part warp execution path
}
// No threads are waiting
```

Figure 3 Code segment with and without branch divergence

Sometimes branch divergence can be introduced due to the non-power-of-two data element number, as demonstrated in Figure 4. So all thread blocks except the last one will enter the if-control statement without doubt. As for the threads in the last thread block, they diverge. However, the performance loss comes from not only the divergence in the last block but also the branch instruction in all other blocks. To overcome this if-control statement, a technique, called padding, is widely used which introduces the regular thread processing. Besides this, it also favors further performance tunings like loop unrolling and tile-compute. For loop unrolling, no fixup code is needed. (See the details in the next paragraphs.)

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
if (i < num_elements) {
    …
}
```

Figure 4 The if-control statement for handling the leftover operations

Data prefetching: As the name suggests, it is to fetch data in advance. It's used to fetch the data for the next loop iteration(s) in order to hide the memory latency by leveraging the asynchronous aspect of memory accesses in GPU hardware. When a memory access operation is executed, it does not block other operations following it, as long as they don't use the data from the memory operation. As written on the left-hand side of Figure 5, every addition waits for its data to be loaded from memory. Inside the loop on the right-hand side, the device first launches a memory load operation for the next iteration and does an addition in parallel. The time for the addition is actually overlapping with the memory access time. But as referred in the previous optimization technique, the increased register usage may lower the number of active warps on an SM. During our optimization process, we tried data prefetching in the FT kernel but the performance doesn't increase significantly. We suspect the added latency hiding effect doesn't help to compensate the loss of occupancy due to the increased register usage.

```
// Without prefetching
for (i = 0; i < N; i++) {
   sum += array[i];
}
```

```
// With prefetching
temp = array[0];
for (i = 0; i < N-1; i++) {
   temp2 = array[i+1];
   sum += temp;
   temp = temp2;
}
sum += temp;
```

Figure 5 Kernel without prefetching vs. with prefetching

Loop unrolling: Loop unrolling is the act of executing multiple copies of the loop body within the same loop iteration. The advantage of loop unrolling is to reduce the number of branches. However, loop unrolling may require extra registers to store the data for multiple loop bodies depending on the ways of unrolling. Figure 6 and Figure 7 demonstrate two ways of loop unrolling with and without increasing register usage. Note that if the data is not a multiple of the unrolling factor, "fixup code" is needed after the main loop as in Figure 8. To avoid this, padding the data elements is needed.

The pro for the unrolling method in Figure 6 (b) is not increasing register usage. The con is not having instruction level parallelism across unrolled loop bodies. More specifically, the instructions in Figure 6 (b) cannot be reordered. On the other hand, Figure 7 provides the most parallelism on instructions across loop bodies where all instructions can be reordered by the compiler because of the use of additional registers. As a consequence, Figure 6 (b) relies on more warp-level parallelism to hide memory latency. So a bigger number of threads in each block is suggested as compared to Figure 7. In our FT and IFT kernel, we adopt the mixture of Figure 6 (b) and Figure 7 to benefit from both.

```
// (a) No unrolling
for (i = 0; i < N; i++) {
    expr = …
    sum += array[i] + expr;
}
```

```
// (b) With unrolling
for (i = 0; i < N; i=i+2) {
    expr = …
    sum += array[i+0] + expr;
    expr = …
    sum += array[i+1] + expr;
}
```

Figure 6 No unrolling vs. unrolling by a factor of 2 without increasing register usage

```
#define UNROLL_FACTOR 2 // 2 or 4
float sumt[UNROLL_FACTOR], expr[UNROLL_FACTOR];
for (i = 0; i < N; i=i+UNROLL_FACTOR) {
    expr[0] = …
    sumt[0] = array[i+0] + expr[0];
    expr[1] = …
    sumt[1] = array[i+1] + expr[1];
    #if UNROLL_FACTOR == 2
         sum += sumt[0] + sumt[1];
    #endif
    #if UNROLL_FACTOR > 2
         expr[2] = …
         sumt[2] = array[i+2] + expr[2];
         expr[3] = …
         sumt[3] = array[i+3] + expr[3];
    #endif
    #if UNROLL_FACTOR == 4
         sum += sumt[0]+sumt[1]+sumt[2]+sumt[3];
    #endif
}
```

Figure 7 Loop unrolling with increasing resisters

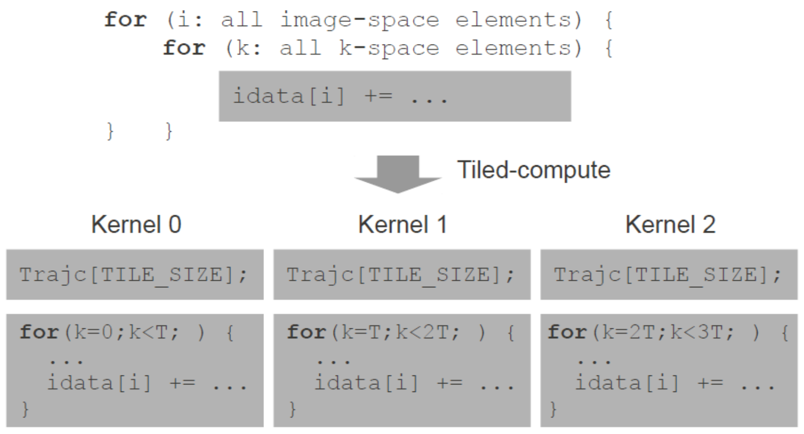
```
for (i = 0; i < N-1; i=i+2) {
    sum += array[i];
    sum += array[i+1];
}
if (N % 2 != 0) {
    sum += array[N-1];
}
```

Figure 8 Unrolling by a factor of 2 with fixup code

Memory layout transformation: Another key feature of the GPU hardware is the memory system enhanced for accessing data from the global memory. The global memory is using DRAM (Dynamic Random Access Memories) as the media. Modern DRAM is designed to favor data accesses with adjacent memory addresses, so called *coalesced* memory access patterns. In the GPU hardware, if the instruction executed by all threads of the same warp is accessing the global memory with consecutive locations, these memory requests will be grouped as one for the DRAM system. The data transfer rate will be close to the maximum global memory bandwidth. As we can see for non-coalesced memory access patterns, the memory requests can be 32. From the programmers point of view, one best way to take advantage of this feature is to have a nice memory layout such that the array indexing is usually in the form of "`array[threadIdx.x]`" instead of "`array[threadIdx.x op stride]`" where the resultant memory addresses are not continuous.

Tiled compute: Tiling of the computation helps to ease the need for a big amount of resource at one time. This is beneficial because the tiled computation can fully saturate the hardware computation and memory resources. Otherwise, the bottleneck can be limited by the global memory bandwidth. Tiled compute can be realized by the use of shared memory or constant memory. Both provide different sharing scopes, access types, and sizes. In our previous work [7-10], we found using the constant memory as the media of tiled data meets our need the best. Besides, transferring a bunch of data together from global memory to constant memory can also benefit from the maximum global memory bandwidth.

The constant memory size is 64 K bytes where only 8 K bytes are on-chip cached. If the memory access locations (or called the working set size) do not randomly exceed the 8 K bytes, the constant memory access latency can be close to that of a register access. In addition, because of the limited size, two issues need to consider for choosing the right data. Firstly, constant memory is read-only by all threads. So the chosen data are better to be used by all threads instead of parts. Secondly, the chosen data should be reused for multiple times. Based on these two criteria, we choose the trajectory data which are read by all threads in both FT and IFT kernels. Figure 9 delineates the phases of tiled compute on the IFT kernel.



```
for (i: all image-space elements) {
    for (k: all k-space elements) {
        idata[i] += ...
}   }
```

```
Trajc[TILE_SIZE];
for(k=0;k<T; ) {
  ...
  idata[i] += ...
}
```

```
Trajc[TILE_SIZE];
for(k=T;k<2T; ) {
  ...
  idata[i] += ...
}
```

```
Trajc[TILE_SIZE];
for(k=2T;k<3T; ) {
  ...
  idata[i] += ...
}
```

Figure 9 Phases of tiled compute on the IFT kernel

4. EXPERIMENTAL RESULTS

Our MRI toolbox is tested on different data sets with a variety of challenges to investigate the performance, and the corresponding results are shown in this section. Our experiments are carried out through a machine with Intel Xeon E5520 CPU having 8 logical threads and one Tesla C2050 (Fermi) GPU having 448 processing cores. Through the experiments we show a possibility that GPU can be the stepping-stone to bridge the gap between a research work and the real-time MRI application in clinics.

4.1 Reconstructed images

The adopted data set is based on the abdominal scan (Double Vision) from the 2010 ISMRM Data Reconstruction Challenge [1]. It uses the spiral acquisition with 8 channel receivers and a magnetic field inhomogeneity map. To simulate different matrix sizes, the 320x320 image from a single slice of the data set is simulated using various k-space trajectories after adding noise to the image and interpolating to the correct image resolution. The spirals are designed according to [13] using a maximum gradient amplitude of 22 mT/m and a maximum slew rate of 140 mT/m/ms. The data of the image sizes, 128^2, 256^2, and 512^2 are simulated using multi-shot spiral with 4, 8, and 20 shot spirals, respectively. The 8-channel sensitivity maps and magnetic field inhomogeneity map are used from the distributed data set.

The performance comparison is shown in Table 2. Three data sets are evaluated with finite difference and 20 conjugate gradient iterations in single-precision floating-point mode. A 640x speedup for 256x256 data size is observed with normalized root mean square error less than 10^{-3} compared to the corresponding CPU reconstruction. Using double precision is moderately slower than single precision (e.g. 6x slower for 256x256), though still much faster than CPU (~128x speedup). Figure 10 shows three GPU reconstruction results of the 512x512 data set under different experimental settings. Through the use of our toolbox, iterative field-corrected non-Cartesian SENSE reconstruction produces dramatically improved image quality with impressive execution time than ever before.

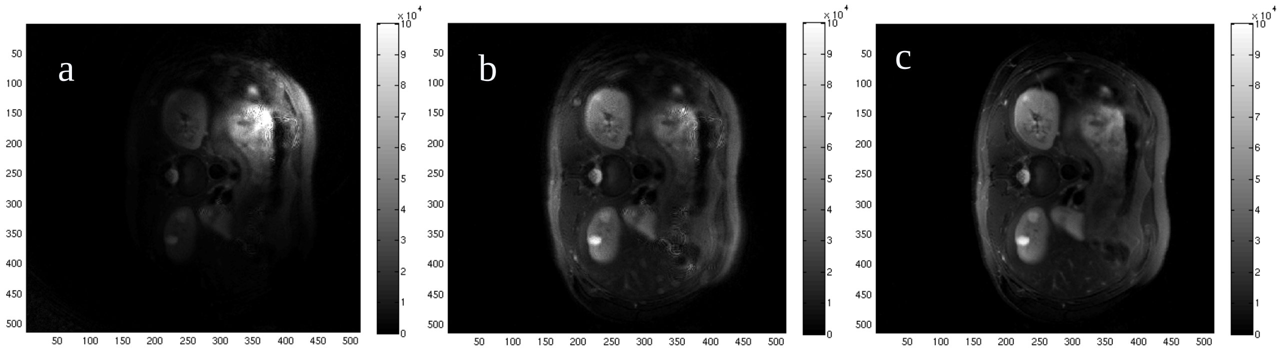


Figure 10 Reconstructed images at 512 matrix size: (a) single coil without field inhomogeneity, (b) SENSE reconstruction without field correction, and (c) SENSE reconstruction with field map correction. Notice that the field inhomogeneity correction corrects for the blurring induced by magnetic field inhomogeneities.

4.2 Performance comparisons

Table 2 lists the performance results on the data set referred in the previous section. Although we only optimize the CPU code with OpenMP and the GCC compiler option "-O3", the resultant CPU performance is far behind that of the GPU performance with/without further optimizations. For the data set with image size 512^2, the extrapolated CPU execution time can be 44 to 150 hours. On the other hand, the GPU performance seems to scale along the line.

The comparison between two GPU versions shows the optimization effect on the FT and IFT kernels which almost dominate the whole execution time. This comparison demonstrates the performance gain through the loop-unrolling technique with padding, referred previously. For the data set with 128^2 image size, the performance gain from loop unrolling is offset by the padded elements. However, the optimization effect from loop unrolling appears when padded elements become a minor portion for big data sets.

Table 2 Execution time of the CPU code, non-optimized GPU code, and the optimized GPU code, all in single-precision mode with finite difference and 20 CG iterations. All images are with SENSE and 8 coils.

Data sets	**CPU (sec)**	**GPU (non-optimized) (sec)**	**GPU (optimized) (sec)**	**Speedup GPU (non-optimized) / GPU (optimized)**	**Speedup GPU (optimized) / CPU**
128x128	872.270	4.673	4.644	1.006x	188x
256x256	34,007.280	71.425	53.142	1.344x	640x
512x512	~days	1062.608	843.776	1.260x	N/A

5. CONCLUSIONS AND FUTURE WORK

We had previously proposed a GPU implementation of our MRI toolbox for advanced MRI reconstruction. The toolbox includes modeling the physics of the magnetic field inhomogeneity to reduce image artifacts, such as geometric distortion and signal loss. In this paper, we extend the GPU implementation to sensitivity encoding for fast MRI reconstruction of parallel imaging with multi-receiver coil array to enable faster MRI data acquisition. General guidelines for porting an image reconstruction algorithm to GPU are also provided: we detail key optimization techniques that can ease the translation of other algorithms to GPU. Final results show that the proposed GPU implementation significantly speedups the SENSE reconstruction by two orders of magnitude. Future work includes further optimizations and the addition of more features towards clinical deployment.

6. ACKNOWLEDGMENTS

The authors acknowledge the high-performance computing resources provided by Institute for Advanced Computing Applications and Technologies (IACAT). This work is partially supported by NIH grant 1R21EB009768-01A1.

REFERENCES

[1] Data Reconstruction Challenge. 2010 Intl Soc Magn Reson Med. (ismrm.org/mri_unbound)

[2] Gray H. Glover, “Simple analytic spiral K-space algorithm,” Magnetic Resonance in Medicine, vol. 42, Issue 2, pp. 412–415, August 1999.

[3] Wen-mei Hwu, “ECE598 HK: Computational Thinking for Manycore Processors,” http://courses.engr.illinois.edu/ece598/hk/.

[4] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu, “Program Optimization Space Pruning for a Multithreaded GPU,” Proceedings of the 2008 International Symposium on Code Generation and Optimization, April 2008.

[5] Sam S. Stone, Justin P. Haldar, Stephanie C. Tsao, Wen-Mei W. Hwu, Zhi-Pei Liang, and Bradley P. Sutton, “Accelerating Advanced MRI Reconstructions on GPUs,” J. Parallel Distrib. Comput., vol. 68, pp. 1307-1318, 2008.

[6] Klaas P. Pruessmann, Markus Weiger, Markus B. Scheidegger, and Peter Boesiger. “SENSE: Sensitivity encoding for fast MRI,” Magn Reson Med, vol. 42, pp. 952-962, 1999.

[7] Yue Zhuo, Xiao-Long Wu, Justin P. Haldar, Zhi-Pei Liang, Wen-mei W. Hwu, Bradley P. Sutton, “The Role of GPUs in Advancing Clinical Imaging with Magnetic Resonance Imaging,” in GPU Computing Gems, W.-M. W. Hwu Ed., Elsevier Inc., 2011. In Press.

[8] Yue Zhuo, Xiao-Long Wu, Justin P. Haldar, Wen-mei W. Hwu, Zhi-Pei Liang, Bradley P. Sutton, “Accelerating Iterative Field-Compensated MR Image Reconstruction on GPUs,” Proceedings of the IEEE Intl Sym on Biomedical Imaging (ISBI), April 2010.

[9] Yue Zhuo, Xiao-Long Wu, Justin P. Haldar, Wen-mei W. Hwu, Zhi-Pei Liang, Bradley P. Sutton, “Multi-GPU Implementation for Iterative MR Image Reconstruction with Field Correction,” Proceedings of the International Society for Magnetic Resonance in Medicine (ISMRM), May 2010.

[10] Yue Zhuo, Xiao-Long Wu, Justin P. Haldar, Wen-mei W. Hwu, Zhi-Pei Liang, Bradley P. Sutton, “Sparse Regularization in MRI Iterative Reconstruction using GPUs,” Proceedings of the 3rd International Conference on BioMedical Engineering and Informatics (BMEI'10), October 2010.

[11] Mila Nikolova and Michael K. Ng, "Analysis of Half-Quadratic Minimization Methods for Signal and Image Recovery," SIAM J. Scientific Computing, vol. 27, pp. 937-966, 2005.

[12] Michael Lustig, David Donoho, and John M. Pauly, “Sparse MRI: The application of compressed sensing for rapid MR imaging,” Magnetic Resonance in Medicine, vol. 58, pp. 1182–1195, 2007.

[13] Gray H. Glover, “Simple analytic spiral K-space algorithm,” Magnetic Resonance in Medicine, vol. 42, Issue 2, pp. 412–415, August 1999.