

Chapter 7

Application Case Study – Quantitative MRI Reconstruction

Application case studies teach computational thinking and programming in a concrete manner. They also help demonstrate how the individual techniques fit into a top-to-bottom development process. Most importantly, they help us to visualize the practical use of these techniques in solving problems. In this chapter, we start with the background and problem formulation of a relatively simple application. We show that parallel execution does not only speedup the existing approaches, but also allows applications experts to pursue approaches that are known to provide benefit but were previously ignored due to their excessive computational requirements. We then use an example algorithm and its implementation source code from such an approach to illustrate how a developer can systematically determine the kernel parallelism structure, assign variables into CUDA memories, steer around limitations of the hardware, validate results, and assess the impact of performance improvements.

7.1. Application Background

Magnetic resonance imaging (MRI) is commonly used by the medical community to safely and non-invasively probe the structure and function of biological tissues in all regions of the body. Images that are generated using MRI have made profound impact in both clinical and research settings. MRI consists of two phases, acquisition (scan) and reconstruction. During the acquisition phase, the scanner samples data in the k-space domain (i.e. the spatial-frequency domain or Fourier transform domain) along a predefined trajectory. These samples are then transformed into the desired image during the reconstruction phase.

The application of MRI is often limited by high noise levels, significant imaging artifacts, and/or long data acquisition times. In clinical settings, short scan times not only increase scanner throughput but also reduce patient discomfort, which tends to mitigate motion-related artifacts. High image resolution and fidelity are important because they enable earlier detection of pathology, leading to improved prognoses for patients. However, the goals of short scan time, high resolution, and high signal-to-noise ratio (SNR) often

conflict; improvements in one metric tend to come at the expense of one or both of the others. One needs new, disruptive technological breakthroughs to be able to simultaneously improve on all of three dimensions. This study presents a case where massively parallel computing provides such a disruptive breakthrough.

The reader is referred to MRI textbooks [Liang] for the physics principles behind MRI. For this case study, we will focus on the computational complexity in the reconstruction phase as affected by the k-space sampling trajectory. The k-space sampling trajectory used by the MRI scanner can significantly affect the quality of the reconstructed image, the time complexity of the reconstruction algorithm, and the time required for the scanner to acquire the samples. In general, the MRI reconstruction problem is defined by equation (1),

$$\hat{m}(\mathbf{r}) = \sum_j W(\mathbf{k}_j) s(\mathbf{k}_j) e^{i2\pi \mathbf{k}_j \cdot \mathbf{r}} \quad (1)$$

Where $m(\mathbf{r})$ is the reconstructed image, $s(\mathbf{k})$ is the measured k-space data, and $W(\mathbf{k})$ is the weighting function that accounts for non-uniform sampling. That is, $W(\mathbf{k})$ decreases the influence of data from k-space regions where a higher density of samples points are taken.

If data are acquired by measuring at uniformly spaced Cartesian grid points in the k-space, then this weighting function is a constant and can thus be factored out of the summation in (1). As a result, the reconstruction of $m(\mathbf{r})$ becomes an inverse Fast Fourier Transform (FFT) on $s(\mathbf{k})$, an extremely efficient computation method. A collection of data measured at such uniform spaced Cartesian grid points is referred as a *Cartesian scan trajectory*. Fig. 7.1(a) depicts a Cartesian scan trajectory. Because Cartesian scan trajectory allows image reconstruction to be performed quickly and efficiently as an inverse FFT on the acquired data, it is the approach used in most clinical MRI scanners today.

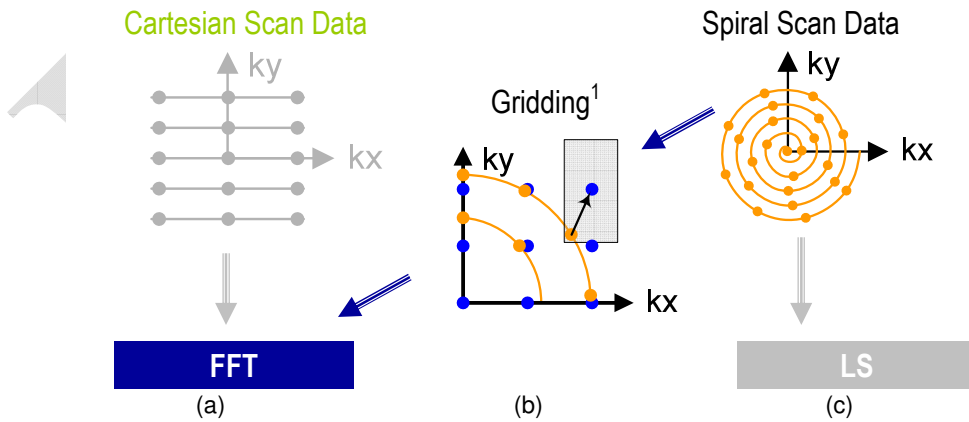


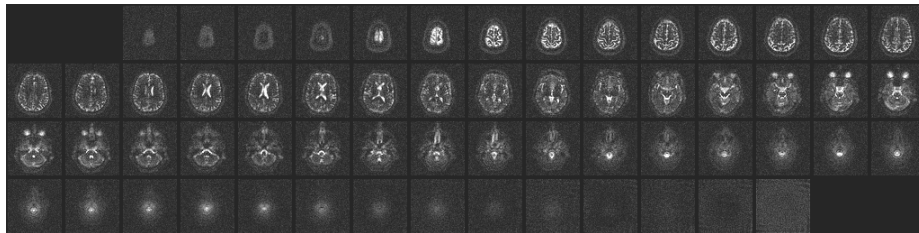
Figure 7.1. Scanner k-space trajectories and their associated reconstruction strategies: (a) Cartesian trajectory with FFT reconstruction, (b) Spiral (or non-Cartesian trajectory in general) followed by gridding to enable FFT reconstruction, (c) spiral (non-Cartesian) trajectory with linear solver based reconstruction.

Although the inverse FFT reconstruction of Cartesian scan data is computationally efficient, Cartesian scan trajectories often require longer scanner time than non-Cartesian trajectories. This stems from the fact that the number of trajectories required to satisfy Nyquist criterion in k-space sampling is different for different trajectory shapes. Non-Cartesian scan trajectories like spirals (shown in Figure 7.1(c)), radial lines (projection imaging) and rosettes cover the k-space much more efficiently. For instance, one needs fewer spiral trajectories than Cartesian trajectories to cover the same area in the k-space, thus reducing the scan time for spiral trajectories. The reduction of trajectories taken translates into less scan time, and thus improved patient comfort and reduced motion related artifacts. Furthermore, when time-dependent phenomena like bolus-spreading or movement of the heart need to be scanned, one must reduce the scan time for generating each video frame to achieve a frame rate needed to produce usable videos.

Image reconstruction from non-Cartesian trajectory data presents both challenges and opportunities. The main challenge arises from the fact that the $W(k)$ function in (1) is not a constant function for non-Cartesian trajectory data and thus can no longer be factored out of the summation. Therefore, one can no longer perform reconstruction by directly applying an inverse FFT to the k-space sample. In a commonly used approach called gridding, the samples are first interpolated onto a uniform Cartesian grid and then reconstructed using the FFT (see Fig. 7.1(b)). Earlier gridding methods used simple bilinear interpolation to map the data onto a Cartesian grid. However, simple bilinear interpolation suffered from extra artifacts and low accuracy. Thus, a popular method for the reconstruction of non-Cartesian k-space data has been to (re)grid the data onto a Cartesian grid via a convolution approach. Each of the data points, which lie along some trajectory in k-space, is convolved with a gridding kernel, and the result sampled and accumulated on a Cartesian grid. Convolution is quite computationally intensive. Studies have shown that for standard gridding parameters, the reconstruction is approximately six times longer than that of Cartesian sampled data and can make the reconstruction on a CPU too long for clinical use. Accelerating gridding computation on many-core processors enables the application of the current FFT approach to non-Cartesian trajectory data. Since we will be examining a convolution-style computation in chapter 8, we will not cover it here.

An interesting opportunity arises when we rethink the entire reconstruction process. The use of non-Cartesian trajectory allows the scanner to take more samples at a given time budget and can potentially improve SNR. However, inverse FFT satisfies no optimality or bound criterion, does not model imaging physics. In applications where the pixels in the reconstructed image are used to provide general diagnosis impression and their numerical values are not used for critical diagnosis decisions, gridding followed by inverse FFT offers a cost-effective solution. However, it does not take advantage of the potential offered by increased amount of sample data to enable more applications where the reconstructed image pixel values are used for critical diagnosis decisions.

By contrast, iterative, statistically optimal image reconstruction methods can more accurately model imaging physics and bound the noise error in each image pixel value. This allows the reconstructed image pixel values to be used for measuring subtle phenomenon such as tissue chemical anomalies before they become anatomical pathology. However, such iterative reconstructions have been impractical for large-scale 3D problems due their excessive computational requirements compared inverse FFT. Recently, these reconstructions have become viable in clinical settings when accelerated on GPUs. In particular, we will show that an iterative reconstruction algorithm that explicitly models imaging physics used to take hours using a high-end sequential CPU but takes only minutes using G80 for a image of moderate resolution, a delay acceptable in clinical settings.



Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago

Figure 7.2. The use of non-cartesian k-space sample trajectory and accurate linear-solver based reconstruction enables new MRI modalities with exciting medical applications. The faster scan time allows acquisition of more samples required to collect in-vivo concentration data on less abundant substance such as sodium in human tissues. The variation or shifting of sodium concentration gives early signs of disease development or tissue death. An example of sodium map of a human brain shown in this Figure can be used to give early indication of brain tumor tissue responsiveness to chemo-therapy protocols, enabling individualized medicine.

The availability of fast, robust, and statistically optimal image reconstruction methods for non-Cartesian k-space trajectories enables new MRI modalities. The typical medical use of MRI today studies the human anatomical structures by mapping out the concentration levels of water in human tissues. The progress a disease development or treatment is measured by the anatomical changes such as enlargement or shrinkage of tumors. One can, however, measure much earlier progress that precedes the anatomical changes by measuring the concentration changes of sodium, a heavily regulated substance in normal human cells. Such a measurement is shown in Figure 7.2. Because sodium and other interesting chemical substance for medical applications are much less abundant than water molecules in human tissues, a reliable measurement of sodium concentration needs a drastic increased k-space coverage in order to achieve an acceptable level of SNR in the resulting image. The increased coverage requirement motivates the use of multiple scanners as well as non-Cartesian data trajectories in order to keep the scanning time reasonable for patients. Iterative, statistically optimal reconstruction methods are needed to deploy physics models needed to correctly stitch together the data from multiple scanners and to bound the noise error for each pixel point so that the pixel values are accurate measurements of the sodium level for early assessment of disease progress.

7.2. Iterative Reconstruction

Haldar and Liang proposed a linear solver based iterative reconstruction algorithm for non-Cartesian scan data, as shown in Fig. 7.1(c). The algorithm allows for explicit modeling and compensation for the physics of the scanner data acquisition process, and can thus reduce the artifacts in the reconstructed image. It is, however, computational intensive. The reconstruction time on high-end sequential CPUs has been hours for moderate-resolution images and thus impractical in clinical use. The objective here is to use the massive parallelism in GPUs to reduce the reconstruction time to a matter of seconds so that one can begin to deploy the new MRI modalities such as sodium imaging in clinical settings.

Figure 7.3 shows a quasi-Bayesian estimation problem formulation of the linear-solver based reconstruction approach, where ρ is a vector containing voxel values for the reconstructed image, F is a matrix that models the physics of imaging process, d is a vector of data samples from the scanner, and W is a matrix that can incorporate prior information such as anatomical constraints. In clinical settings, the anatomical constraints represented in W are derived from one or more high resolution, high-SNR water molecule scans of the patient. These water molecule scans reveal features such as the location of anatomical structures. The matrix W is derived from these reference images. The problem is to solve for ρ given all the other matrices and vectors.

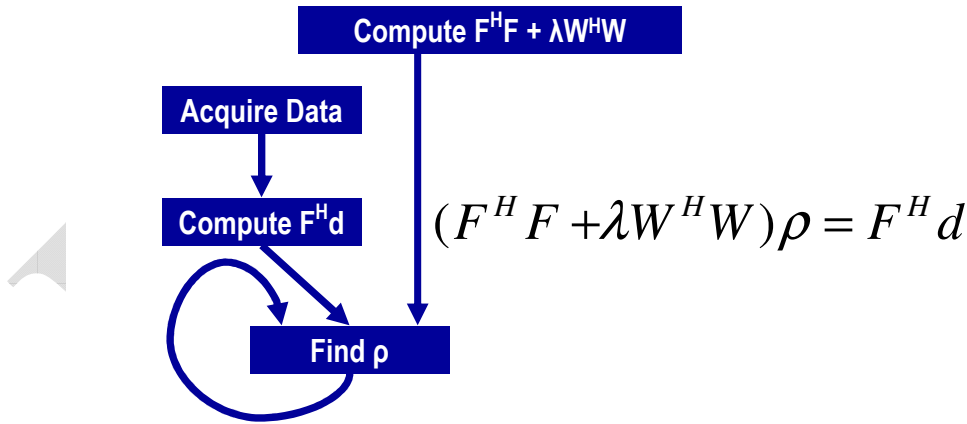


Figure 7.3. An iterative linear solver based approach to reconstruction of non-Cartesian k-space sample data

On the surface, the computational solution to the problem formulation in Figure 7.3 should be very straightforward. It involves matrix-matrix multiplications and addition ($F^H * F + \lambda W^H W$), matrix-vector multiplication ($F^H * d$), matrix inversion $(F^H * F + \lambda W^H W)^{-1}$, and finally matrix-matrix multiplication $((F^H * F + \lambda W^H W)^{-1} * F^H d)$. However, the sizes of the matrices make the solution extremely time consuming. F^H and F are 3D matrices whose dimensions are determined by the resolution of the reconstructed image ρ . Even in a

modest resolution 128^3 -voxel reconstruction, computation of $F^H * F + \lambda W^H W$ can take days to complete on a high-end CPU. For a more desirable resolution of 512^3 -voxel reconstruction, the matrix-matrix multiply will take $4^4 = 64$ times the time. Fortunately, the computation of $F^H * F + \lambda W^H W$ does not have to be repeated for each data acquisition. It needs to be done only once for a scanner set up for a patient. However, due to the extreme long latency of the computation, it is still desirable to speed up the $F^H * F + \lambda W^H W$ computation to minimize the inconvenience of scanner setup for each patient. The acceleration of the $F^H * F + \lambda W^H W$ computation can be found in Stone, et al [Stone 2008].

The matrix-vector multiply to calculate $F^H d$ takes about one order of magnitude less time than $F^H * F + \lambda W^H W$ but can still take about three hours for a 128^3 -voxel reconstruction on a high-end sequential PCU. Since $F^H d$ needs to be computed for every image acquisition, it is critical to reduce its computation to minutes. We will show the details of this process. As it turns out, the core computational structure of $F^H * F + \lambda W^H W$ is identical to that of $F^H d$. As a result, the same methodology can be used to accelerate the computation of both.

The inversion of the $F^H * F + \lambda W^H W$ matrix can be prohibitively expensive due to the sheer size of the inverted matrix. For a 128^3 -voxel reconstruction, the inverted matrix contains well over four trillion complex-valued elements (the number of elements in the inverted matrix equals the square of the number of voxels in the reconstructed image). An iterative method for matrix inversion, such as the conjugate gradient (CG) algorithm, is therefore preferred. The conjugate gradient algorithm reconstructs the image by iteratively solving the equation in Figure 7.3 for ρ . During each iteration, the CG algorithm updates the current image estimate ρ to improve the value of the quasi-Bayesian cost function. The computational efficiency of the CG technique is largely determined by the efficiency of matrix-vector multiplication operations involving $F^H * F + \lambda W^H W$ and ρ , as these operations are required during each iteration of the CG algorithm. Fortunately, matrix W often has a sparse structure that permits efficient multiplication by $W^H W$, and matrix $F^H F$ has a convolution structure that enables efficient matrix multiplication via the FFT. As a result, the “find ρ ” step in Figure 7.3 is much less computationally intensive than $F^H d$, and accounting only less than 1% of the execution of the reconstruction of each image on a sequential CPU. As a result, we will leave it out of the parallelization scope and focus on $F^H d$ in this chapter. We will however, revisit its status at the end of the chapter.

7.3. Computing $F^H d$

<pre> for (m = 0; m < M; m++) { phiMag[m] = rPhi[m]*rPhi[m] + iPhi[m]*iPhi[m]; for (n = 0; n < N; n++) { expQ = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]); rQ[n] += phiMag[m]*cos(expQ); iQ[n] += phiMag[m]*sin(expQ); } } (a) Q computation </pre>	<pre> for (m = 0; m < M; m++) { rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m]; iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m]; for (n = 0; n < N; n++) { expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]); cArg = cos(expFhD); sArg = sin(expFhD); rFhD[n] += rMu[m]*cArg - iMu[m]*sArg; iFhD[n] += iMu[m]*cArg + rMu[m]*sArg; } } (b) $F^H d$ computation </pre>
--	---

Figure 7.4 Computation of Q and $F^H d$

Figure 7.4 shows a sequential C implementation of the computations for the core step of $F^H * F + \lambda W^H W$ (Part (a)) and that for $F^H d$ (Part (b)). It should be clear from a quick glance at Figure 7.4(a) and 7.4(b) that the core step of $F^H * F + \lambda W^H W$ and $F^H d$ have identical structure. Both computations start with an outer loop, which encloses an inner loop. The only differences are the particular calculation done in each loop body and the fact that the core step of $F^H * F + \lambda W^H W$ involves a much larger number of sample points (M), thus a much longer execution time. Thus it suffices to discuss one of them. We will focus on $F^H d$, since this is the one that will need to be run for each image being reconstructed.

A quick glance at Figure 7.4(b) shows that the C implementation of $F^H d$ is an excellent candidate for acceleration on the GPU because it exhibits substantial data-parallelism. The algorithm first computes the real and imaginary components of Mu at each sample point in the k-space, then computes the real and imaginary components of $F^H d$ at each voxel in the image space. The value of $F^H d$ at any voxel depends on the values of all k-space sample points. However, no voxel elements of $F^H d$ depend on any other elements of $F^H d$. Therefore, all elements of $F^H d$ can be computed in parallel. Specifically, all iterations of the outer loop can be done in parallel and all iterations of the inner loop can be done in parallel. The calculations of the inner loop, however, have a dependence on the calculation done in the same iteration of the outer loop.

Despite the algorithm's abundant inherent parallelism, potential performance bottlenecks are evident. First, in the loop that computes the elements of $F^H d$, the ratio of floating-point operations to memory accesses is at best 3:1 and at worst 1:1. The best case assumes that

the `sin` and `cos` trigonometry operations are computed using five-element Taylor series that require 13 and 12 floating-point operations, respectively. The worst case assumes that each trigonometric operation is computed as a single operation in hardware. As we have seen in Chapter 4, a floating-point to memory access ratio of 16:1 or more is needed for the kernel to be not limited by memory bandwidth. Thus, the memory accesses will clearly limit the performance of the kernel unless the ratio is drastically increased.

Second, the ratio of FP arithmetic to FP trigonometry functions is only 13:2. Thus, GPU-based implementation must tolerate or avoid stalls due to long-latency `sin` and `cos` operations. Without a good way to reduce the cost of trigonometry functions, the performance will likely be dominated by the time spent in these functions.

We are now ready to take the steps in converting F^Hd from sequential C code to CUDA kernel.

```
__global__ void cmpFhd(float* rPhi, iPhi, phiMag,
    kx, ky, kz, x, y, z, rMu, iMu, int N) {

    int m = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];

    for (n = 0; n < N; n++) {
        expFhd = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);

        cArg = cos(expFhd);  sArg = sin(expFhd);

        rFhd[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFhd[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

Figure 7.5 First version of the F^Hd kernel. The kernel will not execute correctly due to conflicts between threads in wiring into `rFhd` and `iFhd` arrays.

Step 1: Determine the kernel parallelism structure

The conversion of a loop into a CUDA kernel is conceptually straightforward. Since all iterations of the outer loop of Figure 7.4(b) can be executed in parallel, we can simply convert the outer loop into a CUDA kernel by assigning its iterations to CUDA threads. Figure 7.5 shows the kernel from such a straightforward conversion. Each thread implements an iteration of the original outer loop. The original outer loop has M iterations, and M can be in the millions. We obviously need to have multiple thread blocks to generate enough threads to implement all these iterations.

To make performance tuning easy, we declare a constant `FHD_THREADS_PER_BLOCK` that defines the number of threads in each thread block when we invoke the `cmpFhd` kernel. Thus, we will use `M/FHD_THREADS_PER_BLOCK` for the grid size and `FHD_THREADS_PER_BLOCK` for block size for kernel invocation. Within the kernel, each thread calculates the original iteration of the outer loop that it is assigned to cover using the formula: `blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x`. For example, assume that there are 65,536 k-space samples and we decided to use 512 threads per block. The grid size at kernel invocation would be $65536/512=128$ blocks. The block size would be 512. The calculation of `m` for each thread would be equivalent to `blockIdx.x*512 + threadIdx`.

While the kernel of Figure 7.5 exploits ample parallelism, it suffers from a major problem: all threads write into all voxel elements. This means that the kernel must use the atomic operations in the global memory in the inner loop in order to keep threads from trashing each other's contributions to the voxel value. This can seriously affect the performance of kernel. Note that as is, the code will not even execute correctly. We need to explore other options.

<pre> for (m = 0; m < M; m++) { rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m]; iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m]; for (n = 0; n < N; n++) { expFhd = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]); cArg = cos(expFhd); sArg = sin(expFhd); rFhd[n] += rMu[m]*cArg - iMu[m]*sArg; iFhd[n] += iMu[m]*cArg + rMu[m]*sArg; } } </pre> <p>(a) F^Hd computation</p>	<pre> for (m = 0; m < M; m++) { rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m]; iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m]; } for (m = 0; m < M; m++) { for (n = 0; n < N; n++) { expFhd = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]); cArg = cos(expFhd); sArg = sin(expFhd); rFhd[n] += rMu[m]*cArg - iMu[m]*sArg; iFhd[n] += iMu[m]*cArg + rMu[m]*sArg; } } </pre> <p>(b) after loop fission</p>
--	---

Figure 7.6 Loop fission on the F^Hd computation.

From a quick inspection of Figure 7.4(b), we see that the F^Hd calculation can be split into two separate loops, as shown in Figure 7.6 using a technique called loop fission or loop splitting. This transformation takes the body of a loop and splits it into two loops. In the case of F^Hd, the outer loop consists of two parts: the statements before the inner loop and the inner loop. As shown in Figure 7.6(b), we can perform loop fission on the outer loop by placing the statements before the inner loop into a loop and the inner loop into a second

loop. The transformation changes the execution order of the two parts of the original outer loop. In the original outer loop, both parts of the first iteration execute before the second iteration. After fission, the first part of all iterations will execute; they are then followed by the second part of all iterations. The reader should be able to verify that this change of execution order does not affect the execution results for $F^H d$. This is because the execution of the first part of each iteration does not depend on the result of the second part of any preceding iterations of the original outer loop. Loop fission is a transformation often done by advanced compilers that are capable of analyzing the (lack of) dependence between statements across loop iterations.

```
__global__ void cmpMu(float* rPhi, iPhi, rD, iD, rMu, iMu)
{
    int m = blockIdx.x * MU_THREADS_PER_BLOCK + threadIdx.x;

    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m];
}
```

Figure 7.7 cmpMu kernel

With loop fission, the $F^H d$ computation is now done in two steps. The first step is a single-level loop that calculates the rMu and iMu elements for use in the second loop. The second step corresponds to the loop that calculates the $F^H d$ elements based on the rMu and iMu elements calculated in the first step. Each loop can now be converted into a CUDA kernel. The two CUDA kernels will execute sequentially. Since the second loop needs to use the results from the first loop, separating these two loops into two kernels that execute in sequence does not sacrifice any parallelism.

The cmpMu kernel in Figure 7.7 implements the first loop. The conversion of the first loop from sequential C code to a CUDA kernel is straightforward: each thread implements one iteration of the original C code. Since the M value can be very big, reflecting the large number of k-space samples, such a mapping can result in a large number of threads. Since each thread block can have only up to 512 threads, we will need to use multiple blocks to allow the large number of threads. This can be accomplished by having each thread block to contain a number of threads, specified by MU_THREADS_PER_BLOCK in Figure 7.4(c), and employ $M/MU_THREADS_PER_BLOCK$ blocks needed to cover all M iterations of the original loop. For example, if there are 65,536 k-space samples, the kernel could be invoked with a configuration of 512 threads per block and $65536/512=128$ blocks. This is done by assigning 512 to MU_THREADS_PER_BLOCK and using MU_THREADS_PER_BLOCK as block size and $M/MU_THREADS_PER_BLOCK$ as grid size during kernel innovation.

Within the kernel, each thread can identify the iteration assigned to it using its blockIdx and threadIdx values. Since the threading structure is one dimensional, only blockIdx.x and threadIdx.x need to be used. Because each block covers a section of the original iterations, the iteration covered by a thread is $blockIdx.x * MU_THREADS_PER_BLOCK + threadIdx$. For example, assume that $MU_THREADS_PER_BLOCK=512$. The thread

with blockIdx.x=0 and threadIdx.x=37 covers the 37th iteration of the original loop, whereas the thread with blockIdx.x=5 and threadIdx.x=2 covers the 2,562nd (5*512+2) iteration of the original loop. Using this iteration number to access the Mu, Phi, and D arrays ensures that the arrays are covered by the threads in the same way they were covered by the iterations of the original loop. Because every thread writes into its own Mu element, there is no potential conflict between any of these threads.

Determining the structure of the second kernel requires a little more work. An inspection of the second loop in Figure 7.4(b) shows that there are at least three options in designing the second kernel. In the first option, each thread corresponds to one iteration of the inner loop. This option creates the most number of threads and thus exploits the most amount of parallelism. However, the number of threads would be N*M, with both N in the range of millions and M in the range of hundred thousands. Their product would result in too many threads in the grid.

```
__global__ void cmpFhD(float* rPhi, iPhi, phiMag,
    kx, ky, kz, x, y, z, rMu, iMu, int N) {

    int m = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    for (n = 0; n < N; n++) {
        float expFhD = 2*PI*(kx[m]*x[n]+ky[m]*y[n]+kz[m]*z[n]);

        float cArg = cos(expFhD);
        float sArg = sin(expFhD);

        rFhD[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFhD[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}
```

Figure 7.8 Second option of the F^{Hd} kernel

A second option is to use each thread to implement an iteration of the outer loop. This option employs fewer threads than the first option. Instead of generating N*M threads, this option generates M threads. Since M corresponds to the number of k-space samples and a large number of samples, on the order of a hundred thousand, are typically used to calculate F^{Hd}, this option still exploits a large amount of parallelism. However, this kernel suffers the same problem as the kernel in Figure 7.5. The problem is that each thread will write into all voxels, thus creating an extremely large number of conflicts between threads. As is the case of Figure 7.5, the code in Figure 7.8 will not correctly without adding atomic operations that will significantly slow down the execution. Thus, this option does not work well.

A third option is to use each thread to compute one voxel elemt. This requires interchange the inner and outer loops and then use each thread to implement an iteration of the new outer loop. This is shown in Figure 7.9. Loop interchange is necessary because the loop being implemented by the threads must be the outer loop. Loop inter change makes each of

the new outer loop iteration to process a voxel element. Loop interchange is permissible here because all iterations of both levels of loops are independent of each other. They can be executed in any order relative to one another. Loop interchange, which changes the order of the iterations, is allowed when these iterations can be executed in any order. This option generates N threads. Since N corresponds to the number voxels in the reconstructed image, the N value can be very large for higher-resolution images. For a 128^3 images, there are $128^3 = 2,097,152$ threads, resulting in a large amount of parallelism. For higher resolutions, such as 512^3 , we may need to invoke multiple kernels, each kernel generates the value of a subset of the voxels. Note these threads now all accumulate into their own rFhD and iFhD elements. There is no conflict between threads. These threads can run totally in parallel. This makes the third option the best choice among the three options.

<pre> for (m = 0; m < M; m++) { for (n = 0; n < N; n++) { expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]); cArg = cos(expFhD); sArg = sin(expFhD); rFhD[n] += rMu[m]*cArg - iMu[m]*sArg; iFhD[n] += iMu[m]*cArg + rMu[m]*sArg; } } (a) before loop interchange </pre>	<pre> for (n = 0; n < N; n++) { for (m = 0; m < M; m++) { expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]); cArg = cos(expFhD); sArg = sin(expFhD); rFhD[n] += rMu[m]*cArg - iMu[m]*sArg; iFhD[n] += iMu[m]*cArg + rMu[m]*sArg; } } (b) after loop interchange </pre>
--	---

Figure 7.9 Loop interchange of the F^HD computation

The kernel derived from the interchanged loops is shown in Figure 7.10. The threads are organized into a two-level structure. Each thread covers an iteration of the new outer (n) loop: $\text{blockIdx.x} * \text{FHD_THREADS_PER_BLOCK} + \text{threadIdx.x}$. Once this iteration (n) value is identified, the thread executes the inner loop based on that n value. This kernel can be invoked by having each thread block to contain a number of threads, specified by a global constant FHD_THREADS_PER_BLOCK. Assuming N is the variable that gives the number of voxels in the reconstructed image, $N/\text{FHD_THREADS_PER_BLOCK}$ blocks cover all N iterations of the original loop. For example, if there are 65,536 k-space samples, the kernel could be invoked with a configuration of 512 threads per block and $65536/512=128$ blocks. This is done by assigning 512 to FHD_THREADS_PER_BLOCK and using FHD_THREADS_PER_BLOCK as block size and $N/\text{FHD_THREADS_PER_BLOCK}$ as grid size during kernel innovation.

```

__global__ void cmpFhD(float* rPhi, iPhi, phiMag,
    kx, ky, kz, x, y, z, rMu, iMu, int N) {

    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    for (m = 0; m < M; m++) {
        float expFhD = 2*PI*(kx[m]*x[n]+ky[m]*y[n]+kz[m]*z[n]);

        float cArg = cos(expFhD);
        float sArg = sin(expFhD);

        rFhD[n] += rMu[m]*cArg - iMu[m]*sArg;
        iFhD[n] += iMu[m]*cArg + rMu[m]*sArg;
    }
}

```

Figure 7.10 Third option of the F^Hd kernel

Step 2: Getting Around the Memory Bandwidth Limitation

The simple cmpFhD kernel in Figure 7.10 will provide limited speedup due to memory bandwidth limitations. A simple analysis shows that the execution is limited by the low compute to memory access ratio of each thread. In the original loop, each iteration performs at least 14 memory accesses: kx[m], ky[m], kz[m], x[n], y[n], z[n], rMu[m] twice, iMu[m] twice, rFhD[n] read and write, and iFhD[n] read and write. Meanwhile, about 13 floating point multiple, add, or trigonometry operations are performed in each iteration. Therefore, the compute to memory access ratio is approximately 1, which is too low according to the analysis in Chapter 4.

```

__global__ void cmpFhD(float* rPhi, iPhi, phiMag,
    kx, ky, kz, x, y, z, rMu, iMu, int N) {

    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

    for (m = 0; m < M; m++) {
        float expFhD = 2*PI*(kx[m]*xn_r+ky[m]*yn_r+kz[m]*zn_r);

        float cArg = cos(expFhD);
        float sArg = sin(expFhD);

        rFhDn_r += rMu[m]*cArg - iMu[m]*sArg;
        iFhDn_r += iMu[m]*cArg + rMu[m]*sArg;
    }
    rFhD[n] = rFhDn_r; iFhD[n] = iFhDn_r;
}

```

Figure 7.11 Using registers to reduce memory accesses in the F^Hd kernel

We can immediately improve the compute-to-memory-access ratio by assigning some of the array elements to registers. As we discussed in Chapter 4, this can be done via

automatic variables. A quick review of the kernel in Figure 7.10 shows that for each thread, the same $x[n]$, $y[n]$, and $z[n]$ elements are used across all iterations of the `for` loop. This means that we can load these elements into automatic variables before the execution enters the loop. The kernel can then use the automatic variables inside the loop, thus converting global memory accesses to register accesses. Furthermore, the loop repeatedly reads from and writes into $rFhD[n]$ and $iFhD[n]$. We can have the iterations to read from and write into two automatic variables and only write the contents of these automatic variables into $rFhD[n]$ and $iFhD[n]$ after the execution exits the loop. The resulting code is shown in Figure 7.11. By increasing the number of register usage by 5 for each thread, we have reduced the memory access done by each iteration from 14 to 7. Thus, we have improved the compute to memory access ratio from 13:14 to 6:14. This is a very good improvement and a good use of the precious register resource.

Recall that the register usage can limit the number of blocks that can run in an SM. By increasing the register usage by 5, we really increase the register usage of thread blocks by $5 * FHD_THREADS_PER_BLOCK$. Assuming that we have 128 threads per block, we just increased the block register usage by 640. Since each SM can accommodate a combined register usage of 8912 registers among all blocks assigned to it, we need be careful that any further increase of register usage can begin to limit the number of blocks that can be assigned to an SM.

We need to further improve the compute to memory access ratio to something closer 10:1. We need to further eliminate global memory accesses in the `cmpFHD` kernel. The next candidates to consider are the k-space samples $kx[m]$, $ky[m]$ and $kz[m]$. These array elements are accessed differently than the $x[n]$, $y[n]$ and $z[n]$ elements: different elements of kx , ky and kz is accessed in each iteration of the loop in Figure 7.11. This means that we cannot load each k-space element into an automatic variable register and access that automatic variable off a register through all the iterations. So, the registers will not help here. However, we should notice that the k-space elements are not modified by the kernel. This means that we might be able to place the k-space elements into the constant memory. Perhaps the cache for the constant memory can eliminate most of the memory accesses.

A simple analysis of the loop in Figure 7.11 reveals that the k-space elements are indeed excellent candidates for constant memory. The index used for accessing kx , ky , and kz is m , which is independent of `threadIdx`. This means that all threads in a warp will be accessing the same element of kx , ky , and kz . This is the ideal accessing pattern for cached constant memory: every time an element is brought into the cache, it will be used at least by all threads in a warp which is 32 in G80. This means that for every 32 accesses to the constant memory, at least 31 of them will be served by the cache. This allows the cache effectively eliminate about 96% of the accesses to the constant memory. Better yet, each time when a constant is accessed from the cache, it can be broadcast to all the threads in a warp. This means that no delays are incurred due to any bank conflicts in the access to the cache. This makes constant memory as efficient as registers when accessing k-space elements.

There is, however, a technical issue involved in placing the k-space elements into the constant memory. Recall that constant memory has a capacity of 64KB. However, the size of the k-space samples can be much larger, in the order of hundreds of thousands or even millions. A typical way of working around the limitation of constant memory capacity is to breakdown a large data set to be placed into the constant memory into 64KB chunks. The developer must re-organize the kernel so that the kernel will be invoked multiple times, with each invocation of the kernel consuming only a 64KB chunk of the large data set. This turns out to be quite easy for the cmpFHD kernel.

```

__constant__ float  kx_c[CHUNK_SIZE],
                   ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];
...

__ void main() {

    int i;

    for (i = 0; i < M/CHUNK_SIZE; i++);
        cudaMemcpy(kx_c, &kx[i*CHUNK_SIZE], 4*CHUNK_SIZE,
                   cudaMemcpyHostToDevice);
        cudaMemcpy(ky_c, &ky[i*CHUNK_SIZE], 4*CHUNK_SIZE,
                   cudaMemcpyHostToDevice);
        cudaMemcpy(kz_c, &kz[i*CHUNK_SIZE], 4*CHUNK_SIZE,
                   cudaMemcpyHostToDevice);

    cmpFHD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>>
        ();

}

```

Figure 7.12 Chunking k-space data to fit into constant memory.

A careful examination of the loop in Figure 7.11 reveals that all threads will be sequentially marching through the k-space arrays. For large data sets, the loop in the kernel simply iterates more times. This means that we can divide up the loop into sections, with each section processing a chunk of the k-space elements that fit into the 64KB capacity of the constant memory. The host code now invokes the kernel multiple times. Each time the host invokes the kernel, it places a new chunk into the constant memory before calling the kernel function. This is illustrated in Figure 7.12.

In Figure 7.12, the cmpFHD kernel is called from a loop. The code assumes that kx, ky, and kz are in the host memory. The dimension of these kx, ky, and kz are given by M. At each iteration, the host code calls the cudaMemcpy() function to transfer a chunk of the k-space data into the device constant memory. The kernel is then invoked to process the chunk. Note that when M is not a perfect multiple of CHUNK_SIZE, the host code will need to have an additional round of cudaMemcpy and one more kernel invocation to finish the remaining k-space data.

```

__global__ void cmpFhD(float* rPhi, iPhi, phiMag,
                     x, y, z, rMu, iMu, int M) {

    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

    for (m = 0; m < M; m++) {
        float expFhD = 2*PI*(kx[m]*xn_r+ky[m]*yn_r+kz[m]*zn_r);

        float cArg = cos(expFhD);
        float sArg = sin(expFhD);

        rFhDn_r += rMu[m]*cArg - iMu[m]*sArg;
        iFhDn_r += iMu[m]*cArg + rMu[m]*sArg;
    }
    rFhD[n] = rFhDn_r; iFhD[n] = iFhDn_r;
}

```

Figure 7.13 Revised F^Hd kernel to use constant memory

Figure 7.13 shows the revised kernel that accesses the k-space data from constant memory. Note that pointers to kx, ky, and kz are no longer in the parameter list of the kernel function. Since we cannot use pointers to access variables in the constant memory, the kx_c, ky_c, and kz_c arrays are accessed as global variables declared under `__constant__` keyword as shown Figure 7.12. By accessing these elements from the constant cache, the kernel now has effectively only 4 global memory accesses to the rMu and iMu arrays. The compiler will typically recognize that the four array accesses are made to only two locations. It will only perform two global accesses, one to rMu[m] and one to iMu[m]. The values will be stored in temporary register variables for use in the other two. This makes the final number of memory accesses to 2. The computer to memory access ratio is down to 14:2, or 7:1. This is still not quite the desired 10:1 ratio but is sufficiently high that the memory bandwidth limitation is no longer the only factor that limits performance. As we will see, we can perform a few other optimizations that make computation more efficient and further improve performance.

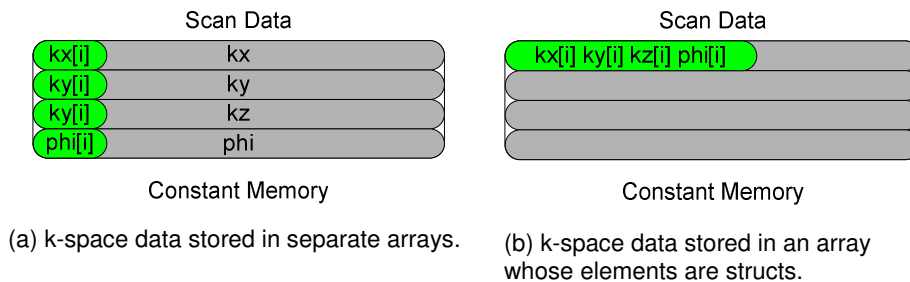


Figure 7.14 Effect of k-space data layout on constant cache efficiency.

If we ran the code in Figures 12 and 13, we would have found out that the performance enhancement was not as high as we expected. As it turns out, the code shown in these

figures does not result in as much memory bandwidth reduction as we expected. The reason is that the constant cache does not perform very well for the code. This has to do with the design of the constant cache and the memory layout of the k-space data. As shown in Figure 14, each constant cache entry is designed to store multiple consecutive words. This design reduces the overhead of bookkeeping in the hardware design. If multiple data elements that are used by each thread are not in consecutive words, as illustrated in Figure 14(a), they will end up taking up multiple cache entries. Due to cost constraints, the constant cache has only a very small number of entries. As shown in Figure 12 and 13, the k-space data is stored in three arrays: `kx_c`, `ky_c`, and `kz_c`. During each iteration of the loop, three entries of the constant cache is needed to hold the three k-space element being processed. Since different warps can be at very different iterations, they may require many entries altogether. As it turns out, the G80 cache capacity was not sufficient to provide sufficient number of entries for all the warps active in an SM.

```

struct kdata {
    float x, float y, float z;
} k;

__constant__ struct kdata k_c[CHUNK_SIZE];
...

__ void main() {

    int i;

    for (i = 0; i < M/CHUNK_SIZE; i++);
        cudaMemcpy(k_c, k, 12*CHUNK_SIZE, cudaMemcpyHostToDevice);

        cmpFHD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>>
            ();

    }

```

Figure 7.15 adjusting k-space data layout to improve cache efficiency

The problem of inefficient use of cache entries has been well studied in the literature and can be solved by adjusting the memory layout of the k-space data. The solution is illustrated in Figure 14(b) and the code based on this solution in Figures 15. Rather than having the x, y, and z components of the k-space data to be stored in three separate arrays, the solution stores these components in an array whose elements are structs. The declaration of the array is shown on top of Figure 15. By storing the x, y, and z components in the three fields of an array element, the developer forces these components to be stored in consecutive locations of the constant memory. Therefore, all three components used by an iteration can now fit into one cache entry, reducing the number of entries needed to support the execution of all the active warps. Note that since we have only one array to hold all k-space data, we can just use one `cudaMemcpy` to copy the entire chunk to the device constant memory. The size of the transfer is adjusted from `4*CHUNK_SIZE` to `12*CHUNK_SIZE` to reflect the transfer of all the three components in one `cudaMemcpy` call.

```

__global__ void cmpFhD(float* rPhi, iPhi, phiMag,
    x, y, z, rMu, iMu, int M) {

    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

    for (m = 0; m < M; m++) {
        float expFhD = 2*PI*(k[m].x*xn_r+k[m].y*yn_r+k[m].z*zn_r);

        float cArg = cos(expFhD);
        float sArg = sin(expFhD);

        rFhDn_r += rMu[m]*cArg - iMu[m]*sArg;
        iFhDn_r += iMu[m]*cArg + rMu[m]*sArg;
    }
    rFhD[n] = rFhDn_r; iFhD[n] = iFhDn_r;
}

```

Figure 7.16 Adjusting the k-space data memory layout in the F^Hd kernel

With the new data structure layout, we also need to revise the kernel so that the access is done according to the new layout. The new kernel is shown in Figure 7.16. Note that $k_x[m]$ has become $k[m].x$, $k_y[m]$ has become $k[m].y$, and so on.

```

__global__ void cmpFhD(float* rPhi, iPhi, phiMag,
    x, y, z, rMu, iMu, int M) {

    int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

    float xn_r = x[n]; float yn_r = y[n]; float zn_r = z[n];
    float rFhDn_r = rFhD[n]; float iFhDn_r = iFhD[n];

    for (m = 0; m < M; m++) {
        float expFhD = 2*PI*(k[m].x*xn_r+k[m].y*yn_r+k[m].z*zn_r);

        float cArg = __cos(expFhD);
        float sArg = __sin(expFhD);

        rFhDn_r += rMu[m]*cArg - iMu[m]*sArg;
        iFhDn_r += iMu[m]*cArg + rMu[m]*sArg;
    }
    rFhD[n] = rFhDn_r; iFhD[n] = iFhDn_r;
}

```

Figure 7.17 using hardware `__sin()` and `__cos()` functions.

Step 3: Using hardware trigonometry functions

As we discussed in Chapter 5, CUDA offers hardware mathematic functions that offer much higher throughput than their software counter parts. These functions are implemented in the SFU (Super Function Units) in G80. The procedure for using these functions is quite easy. In the case of the `cmpFhD` kernel, what we need to do is to change the calls to `sin` and `cos` functions into their hardware versions: `__sin` and `__cos`. Because, these functions are called in a heavily executed loop body, we expect that the change will result in very significant performance improvement. The resulting `cmpFhD` kernel is shown in Figure 7.17.

A.N. Netravali and B.G. Haskell, Digital Pictures: Representation, Compression, and Standards (2nd Ed), Plenum Press, New York, NY (1995).

$$\begin{aligned}
 MSE &= \frac{1}{mn} \sum_i \sum_j (I(i, j) - I_0(i, j))^2 & A_s &= \frac{1}{mn} \sum_i \sum_j I_0(i, j)^2 \\
 PSNR &= 20 \log_{10} \left(\frac{\max(I_0(i, j))}{\sqrt{MSE}} \right) & SNR &= 20 \log_{10} \left(\frac{\sqrt{A_s}}{\sqrt{MSE}} \right)
 \end{aligned}$$

Figure 7.18 metrics used to validate the accuracy of hardware functions
 I_0 is perfect image. I is reconstructed image.
 PSNR is Peak signal-to-noise ratio, SNR is Signal-to-noise ratio.

We need to, however, be careful about the reduced accuracy when switching from software functions to hardware functions. As we discussed in Chapter 6, Hardware implementation currently have slightly less accuracy than software libraries. In the case of MRI, we need to make sure that the hardware implementation passes test cases that measure the Signal to Noise Ratio (SNR) of the resulting image, as shown in Figure 7.18. The testing process involves a “perfect” image (I_0). We then use a reverse process to generate a corresponding

“scanned” k-space data that is actually synthesized. The synthesized scanned data is then processed by the proposed reconstruction system to generate a reconstructed image (I). The value of the voxels in the images are then plugged into the MSE, PSNR, and SNR formula in Figure 7.18.

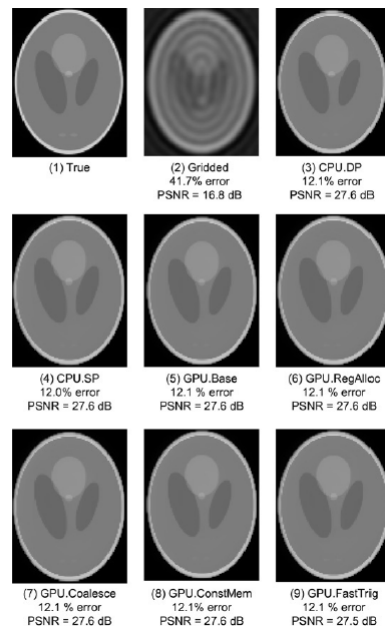


Figure 7.19 Validation of floating-point precision and accuracy of the different FHD implementations.

The criteria for passing the test depend on the application that the image is intended for. In our case, we worked with the MRI clinical experts to ensure that the SNR changes due to hardware functions is well within the accepted limits for their applications. In applications where the images are used by physicians to form impression of injury or disease evaluation, one also needs to have visual inspection of the image quality. Figure 7.19 shows the visual comparison of the original “true” image. It then shows that the SNR achieved CPU double precision and single precision implementation both achieve 27.6 dB, an well acceptable level for the application. A visual inspection also shows that the reconstructed image indeed correspond well with the original image.

The advantage of iterative reconstruction compared to a simple bilinear interpolation gridding/FFT is also obvious in Figure 7.19. The image reconstructed with the simple gridding/FFT has an SNR of only 16.8 dB, substantially lower than that of the iterative reconstruction method. A visual inspection of the gridding/FFT image shows that there is severe artifact that can significantly impact the usability of the image for diagnosis purposes.

When we move from double precision to single precision arithmetic on the CPU, there was no measurable degradation of SNR, which remains at 27.6 dB. When we move the trig function from software library to the hardware units, we observed a negligible degradation

of SNR, from 27.6 dB to 27.5 dB. A visual inspection shows that the reconstructed image does not have significant artifact compared to the original image.

Step 4: Experimental Performance Tuning

Up to this point, we have not determined the appropriate values for the configuration parameters for the kernel. For example, we need to determine the best number of threads for each block. On one hand, using a large number of threads in a block is needed to reach thread capacity of each SM given that up to eight blocks can be assigned to each SM. On the other hand, having more threads in each block increases the register usage of each block and can reduce the number of blocks that can fit into an SM. Some possible values of number of threads per block are 32, 64, 128, 256, and 512. One can also consider non-power-of-two numbers.

One also needs to determine the number of scan points per grid. All the scan point data consumed by a grid must fit into the 64KB constant memory. This is 16K single precision. Since each scan point requires three single-precision floating point data, we can have up to 4K scan points if we want to use power-of-two scan points in each grid for convenient loop control. Some possible numbers are 32, 64, 128, 256, 1024, 2048, and 4096.

Another kernel configuration parameter is the number of times one should unroll the body of the `for` loop. On one hand, unrolling the loop can reduce the number of overhead instructions, and potentially reduce the number of clock cycles to process each k-space sample data. On the other hand, too much unrolling can increase the usage of registers and reduce the number of blocks that can fit into an SM.

Note that these configurations are not independent of each other. Increasing one parameter value can potentially use the resource that could be used to increase another parameter value. As a result, one needs to evaluate these parameters jointly in an experimental manner. That is, one may need to change the source code for each joint configuration and measure the run time. There can be a large number of source code versions to try. In the case of F^HD, the performance improves about 20% by systematically searching all the combinations and choosing the one with the best measured runtime, as compared to a heuristic tuning search effort that explore some promising trends.

7.4. Final Evaluation

To obtain a reasonable baseline, we implemented two versions of FHD on the CPU. Version CPU.DP uses double-precision for all floating-point values and operations, while version CPU.SP uses single-precision. Both CPU versions are compiled with Intel's `icpc` (version 10.1) using flags `-O3 -msse3 -axT -vec-report3 -fp-model fast = 2`, which (1) vectorizes the algorithm's dominant loops using instructions tuned for the Core 2 architecture, and (2) links the trigonometric operations to fast, approximate functions in the math library. Based on experimental tuning with a smaller data set, the inner loops are unrolled by a factor of four and the scan data is tiled to improve locality in the L1 cache.

	Q		F ^H d		Total	
Reconstruction	Run Time (m)	GFLOP	Run Time (m)	GFLOP	Linear Solver (m)	Recon. Time (m)
Gridding + FFT (CPU, DP)	N/A	N/A	N/A	N/A	N/A	0.39
LS (CPU, DP)	4009.0	0.3	518.0	0.4	1.59	519.59
LS (CPU, SP)	2678.7	0.5	342.3	0.7	1.61	343.91
LS (GPU, Naïve)	260.2	5.1	41.0	5.4	1.65	42.65
LS (GPU, CMem)	72.0	18.6	9.8	22.8	1.57	11.37
LS (GPU, CMem, SFU, Exp)	7.5 357X	178.9	1.5 228X	144.5	1.69	3.19 108X

Figure 7.20 Summary of Performance improvements

Each GPU version of F^Hd is compiled using `nvcc -O3` (CUDA version 1.1) and executed on a 1.35 GHz Quadro FX 5600. The Quadro card is housed in a system with a 2.4 GHz dual-socket, dual-core Opteron 2216 CPU. Each core has a 1 MB L2 cache. The CPU versions use pthreads to execute on all four cores of 2.66 GHz Core 2 Extreme quad-core CPU, which has peak theoretical capacity of 21.2 GFLOPS per core and a 4 MB L2 cache. The CPU versions perform substantially better on the Core 2 Extreme quad-core than on the dual-socket, dual-core Opteron.

All reconstructions use the GPU version of the linear solver, which executes 60 iterations on the Quadro FX 5600. Two versions of Q were computed on the Core 2 Extreme, one using double-precision and the other using single-precision. The singleprecision Q was used for all GPU-based reconstructions and for the reconstruction involving CPU.SP, while the double-precision Q was used only for the reconstruction involving CPU.DP. As the computation of Q is not on the reconstruction's critical path, we give Q no further consideration.

To facilitate comparison of the iterative reconstruction with a conventional reconstruction, we also evaluated a reconstruction based on bilinear interpolation gridding and inverse FFT. Our version of the gridded reconstruction is not optimized for performance, but it is already quite fast.

All reconstructions are performed on sample data obtained from a simulated, three-dimensional, non-Cartesian scan of a phantom image. There are 284,592 sample points in the scan data set, and the image is reconstructed at 1,283 resolution, for a total of 221 voxels. In the first set of experiments, the simulated data contains no noise. In the second set of experiments, we added complex white Gaussian noise to the simulated data. When determining the quality of the reconstructed images, the percent error and peak signal-to-noise ratio metrics are used. The percent error is the root-mean-square (RMS) of the voxel error divided by the RMS voxel value in the true image (after the true image has been sampled at 1283 resolution).

The data (runtime, GFLOPS, and images) were obtained by reconstructing each image once with each of the implementations of the F^Hd algorithm described above. There are two exceptions to this policy. For GPU.Tune and GPU.Multi, the time required to compute F^Hd is so small that run-

time variations in performance may become non-negligible. Therefore, for these configurations we computed $F^H d$ three times and reported the average performance.

As shown in Figure 7.20, the total reconstruction time for the test image using bilinear interpolation gridding followed by inverse FFT takes only less than One minute on a high-end sequential CPU. This confirms that there is little value in parallelizing this traditional reconstruction strategy. It is, however, obvious from Figure 7.19(2) that the resulting image exhibits an unacceptable level of artifact.

The LS (CPU, DP) row shows the execution timing of reconstructing the test image using double-precision floating-point arithmetic on the CPU. The timing shows that the core step (Q) of calculating $F^H * F + \lambda W^H W$. The first observation is that the Q computation for a moderate resolution image based on a moderate sized data sample takes an unacceptable amount of time (more than 65 hours) on the CPU for setting up the system for a patient. Note that this time is eventually reduced to 6.5 minutes on the GPU with all the optimizations described in Section 7.3. The second observation is that the total reconstruction time of each image requires more than 8 hours, with only 1.59 minutes spent in the linear solver. This validates our decision to focus our parallelization effort on $F^H d$.

The LS(CPU, SP) row shows that we can reduce the execution time significantly when we convert the computation from double-precision floating-point arithmetic to single-precision on the CPU. This is because the SSE instructions have higher throughput, i.e., calculate more data elements per clock cycle when executing in single-precision mode. The execution times, however, are still unacceptable for practical use.

The LS(GPU, naïve) row shows that a straightforward CUDA implementation can achieve a speedup about 10 times for Q and 8 times for the reconstruction of each image. This is a good speedup but the resulting execution times are still unacceptable for practical use.

The LS(GPU, Cmem) row shows that significant further speedup is achieved by using registers and constant cache to get around the global memory bandwidth limitations. These enhancements achieve about 4X speedup over the naïve CUDA code! This shows the importance of achieving good compute to memory ratios in CUDA kernels. These enhancements bring the CUDA code to about 40X speedup over the single precision CPU code.

The LS(GPU, CMem, SPU, exp) row shows the use of hardware trigonometry functions and experimental tuning together results in dramatic further speedup. A separate experiment, not shown here, shows that most of the speedup comes from hardware trigonometry functions. The total speedup over CPU single-precision code is very impressive: 357X for Q and 108X for the reconstruction of each image.

An interesting observation is that in the end, the linear solver actually takes more time than $F^H d$. This is because we have accelerated $F^H d$ dramatically (228X). What used to be close to 100% of the per image reconstruction time now accounts for less than 50%. Any further

acceleration will now require acceleration of the linear solver, a much more difficult type of computation for massively parallel execution.

DRAFT