CHAPTER

14 Application case study—non-Cartesian magnetic resonance imaging

An introduction to statistical estimation methods

CHAPTER OUTLINE

Application case studies teach computational thinking and practical programing techniques in a concrete manner. They also help demonstrate how the individual techniques fit into a top-to-bottom application 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 that has traditionally been constrained by the limited capabilities of the main stream computing systems. We show that parallel execution not only speeds up the existing approaches, but also allows the applications experts to pursue an approach that has been known to provide benefit but was previously ignored due to their excessive computational requirements. *This approach represents an increasingly important class of computational methods that derive statistically optimal estimation of unknown values from a very large amount of observational data.* We 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 different types of memories, steer around limitations of the hardware, validate results, and assess the impact of performance improvements.

Programming Massively Parallel Processors. DOI: http://dx.doi.org/10.1016/B978-0-12-811986-0.00014-5

14.1 BACKGROUND

Magnetic resonance imaging (MRI) is commonly used by the medical community to safely and noninvasively 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. Intuitively, the reconstruction phase *estimates* the shape and texture of the tissues based on the observation k-space data collected from the scanner.

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 early 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 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 breakthrough.

The reader is referred to MRI textbooks such as Liang and Lauterbur [LL 1999] for the physics principles behind MRI. For this case study, we will focus on the computational complexity in the reconstruction phase and how the complexity is 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. Eq. (14.1) shows a formulation that relates the k-space samples to the reconstructed image for a class of reconstruction methods.

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

In Eq. (14.1), $\widehat{m}(\mathbf{r})$ is the reconstructed image, $s(\mathbf{k})$ is the measured k-space data, and $W(\mathbf{k})$ is the weighting function that accounts for nonuniform sampling. That is, $W(\mathbf{k})$ decreases the influence of data from k-space regions where a higher density of samples points are taken. For this class of reconstructions, $W(\mathbf{k})$ can also serve as an *apodization* filtering function that reduces the influence of noise and reduces artifacts due to finite sampling.

If data are acquired at uniformly spaced Cartesian grid points in the k-space under ideal conditions, then the $W(\mathbf{k})$ weighting function is a constant and can thus be factored out of the summation in Eq. (14.1). Furthermore, with uniformly spaced Cartesian grid samples, the exponential terms in (1) are uniformly spaced in the k-space. As a result, the reconstruction of $\widehat{m}(\mathbf{r})$ becomes an inverse Fast Fourier Transform (iFFT) on $s(\mathbf{k})$, an extremely efficient computation method. A collection

of data measured at such uniformly spaced Cartesian grid points is referred to as a *Cartesian scan trajectory*. Fig. 14.1A depicts a Cartesian scan trajectory. In practice, Cartesian scan trajectories allow straightforward implementation on scanners and are widely used in clinical settings today.

Although the iFFT reconstruction of Cartesian scan data is computationally efficient, non-Cartesian scan trajectories often have advantage in reduced sensitivity to patient motion, better ability to provide self-calibrating field inhomogeneity information, and reduced requirements on scanner hardware performance. As a result, non-Cartesian scan trajectories like spirals (shown in Fig. 14.1C), radial lines (also known as projection imaging) and rosettes have been proposed to reduce motion-related artifacts and address scanner hardware performance limitations. These improvements have recently allowed the reconstructed image pixel values to be used for measuring subtle phenomenon such as tissue chemical anomalies before they become anatomical pathology. Fig. 14.2 shows such an MRI reconstruction-based measurement that generates a map of sodium, a heavily regulated substance in normal human tissues. The information can be used to track to tissue health in stroke and cancer treatment processes. Because sodium is much less abundant than water molecules in human tissues, reliable measure of sodium levels requires a higher SNR through higher number of samples and thus needs to mitigate the extra scan time with non-Cartesian scan trajectories.

Image reconstruction from non-Cartesian trajectory data presents both challenges and opportunities. The main challenge arises from the fact that the exponential terms are no longer uniformly spaced; the summation does not have the form of a Fast Fourier Transform (FFT) anymore. Therefore, one can no longer perform reconstruction by directly applying an iFFT to the k-space samples. 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. 14.1B). For example, a convolution approach to gridding takes a k-space data point, convolves it with a gridding convolution mask, and

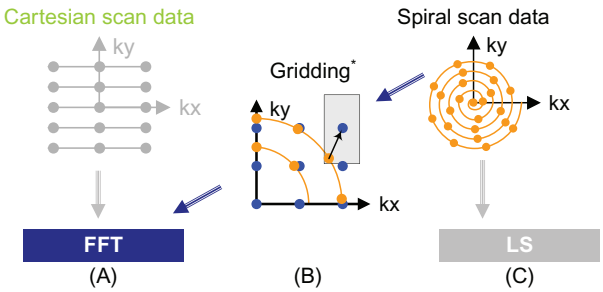
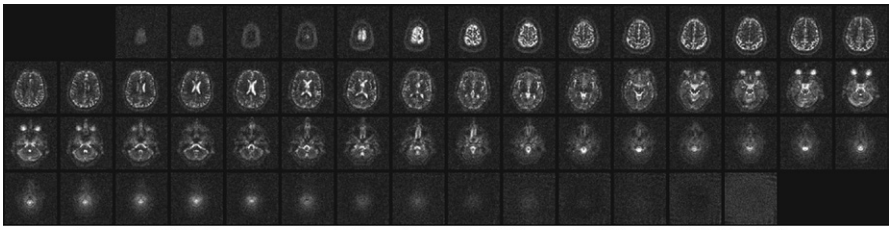


FIGURE 14.1

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. Note: *Based on Fig 1 of Lustig et al. Fast Fourier Transform for Iterative MR Image Reconstruction, IEEE Int'l Symp. on Biomedical Imaging, 2004.



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

FIGURE 14.2

Non-Cartesian k-space sample trajectory and accurate linear-solver-based reconstruction enable new capabilities with exciting medical applications. The improved SNR enables reliable collection of in-vivo concentration data on chemical substance such as sodium in human tissues. The variation or shifting of sodium concentration gives early signs of disease development or tissue death. For example, the sodium map of a human brain shown in this figure can be used to give early indication of brain tumor tissue responsiveness to chemotherapy protocols, enabling individualized medicine.

accumulates the results on a Cartesian grid. As we have seen in Chapter 7, Parallel patterns: convolution, convolution is quite computationally intensive and is an important pattern for massively parallel computing. Accelerating convolution gridding computation with parallel computing facilitates the application of the current FFT approach to non-Cartesian trajectory data. Since we will be examining two convolution-style applications in the next two chapters, we will not cover the approach here.

In this chapter, we will cover an iterative, *statistically optimal* image reconstruction method which can accurately model imaging physics and bound the noise error in the resulting image pixel values. Such statistically optimal methods are gaining importance in the wake of big data analytics. However, such iterative reconstruction methods have been impractical for large-scale 3D problems due to their excessive computational requirements compared to gridding. Recently, these reconstructions have become viable in clinical settings when accelerated on graphics processing unit (GPUs). In particular, we will show that an iterative reconstruction algorithm that used to take hours using a high-end sequential central processing unit (CPUs) to reconstruct an image of moderate resolution now takes only minutes using both CPUs and GPUs, a delay acceptable in clinical settings.

14.2 ITERATIVE RECONSTRUCTION

Haldar and Liang proposed a linear-solver-based iterative reconstruction algorithm for non-Cartesian scan data, as shown in Fig. 14.1C. 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, computationally expensive. The reconstruction time on high-end sequential CPUs has been

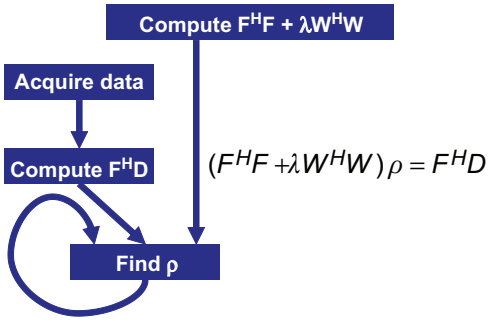


FIGURE 14.3

An iterative linear-solver-based approach to reconstructing non-Cartesian k-space sample data.

hours for moderate-resolution images and thus impractical in clinical use. We use this as an example of innovative methods that have required too much computation time to be considered practical. We will show that massive parallelism can reduce the reconstruction time to the order of a minute so that one can deploy the new imaging capabilities such as sodium imaging in clinical settings.

Fig. 14.3 shows a solution of the quasi-Bayesian estimation problem formulation of the iterative 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.

On the surface, the computational solution to the problem formulation in Fig. 14.3 should be very straightforward. It involves matrix–matrix multiplications and addition ($F^HF+ \lambda W^HW$), matrix–vector multiplication (F^HD), matrix inversion $(F^HF+ \lambda W^HW)^{-1}$, and finally matrix–matrix multiplication $((F^HF+ \lambda W^HW)^{-1}*F^HD)$. However, the sizes of the matrices make this straightforward approach 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, there are 128^3 columns in F with N elements in each column where N is the number of k-space samples used. Obviously, F is extremely large. Such massive dimensions are commonly encountered in big-data analytics, when one tries to use iterative-solver methods to estimate the major contributing factors of a massive amount of noisy observational data.

The sizes of the matrices involved are so large that the matrix operations involved in a direct solution of the equation in Fig. 14.3 using methods such as Gaussian

elimination discussed in Chapter 6, Numerical considerations, are practically intractable. An iterative method for matrix inversion, such as the conjugate gradient (CG) algorithm, is therefore preferred. The CG algorithm reconstructs the image by iteratively solving the equation in Fig. 14.3 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 $F^HF + \lambda W^HW$ 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^HW, and matrix F^HF is Toeplitz that enables efficient matrix–vector multiplication via the FFT. Stone et al. [SHT 2008] present a GPU accelerated method for calculating Q, a data structure that allows us to quickly calculate matrix–vector multiplication involving F^HF without actually calculating F^HF itself. The calculation of Q can take days on a high-end CPU core. It only needs to be done once for a given trajectory and can be used for multiple scans.

The matrix–vector multiply to calculate F^HD takes about one order of magnitude less time than Q but can still take about three hours for a 128^3-voxel reconstruction on a high-end sequential CPU. Recall that D is the vector of data samples from the scanner. Thus, F^HD needs to be computed for every image acquisition; it is desirable to reduce the computation time of F^HD to minutes.[1] We will show the details of this process. As it turns out, the core computational structure of Q is identical to that of F^HD; Q just has much larger data structure dimensions. As a result, the same methodology can be used to accelerate the computation of both.

The "find ρ" step in Fig. 14.3 performs the actual CG based on F^HD. As we explained earlier, precalculation of Q makes this step much less computationally intensive than F^HD, accounting for only less than 1% of the execution of the reconstruction of each image on a sequential CPU. As a result, we will leave the CG solver out of the parallelization scope and focus on F^HD in this chapter. We will however, revisit its status at the end of the chapter.

14.3 COMPUTING F^HD

Fig. 14.4 shows a sequential C implementation of the computations for the core step of computing a data structure for matrix–vector multiplications between F^H*F and ρ (referred to as Q computation in Fig. 14.4A) during the iterative CG solution process without explicitly calculating F^HF and that for computing F^HD (Fig. 14.4B). It should be clear from a quick glance at Fig. 14.4A and Fig. 14.4B that the core step of Q and F^HD have identical loop 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 Q involves a much larger `m`, since it implements a matrix–matrix multiplication as opposed to a matrix–vector

[1] Note that the FHD computation can be approximated with gridding and can run in a few seconds, with perhaps reduced quality of the final reconstructed image.

(A)

```
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);
  }
}
```

(B)

```
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;
  }
}
```

FIGURE 14.4

Computation of Q and F^HD. (A) Q computation, (B) F^HD computation.

multiplication; thus it incurs a much longer execution time. Thus it suffices to discuss one of them from the parallelization perspective. We will focus on F^HD, since this is the one that will need to be run for each data acquisition.

A quick glance at Fig. 14.4B shows that the C implementation of F^HD is an excellent candidate for acceleration because it exhibits substantial data parallelism. The algorithm first computes the real and imaginary components of Mu (`rMu` and `iMu`) at each sample point in the k-space, it then computes the real and imaginary components of F^HD at each voxel in the image space (M is the total number of k-space samples and N is the total number of voxels in the reconstructed image). The value of F^HD at any voxel depends on the values of all k-space sample points. However, no voxel elements of F^HD depend on any other elements of F^HD. Therefore, all elements of F^HD 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 by the preceding statements 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^HD, 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 5, Performance considerations, 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 floating-point arithmetic to floating-point trigonometry functions is only 13:2. Thus, GPU-based implementation must tolerate or avoid stalls due to long-latency and low-throughput of `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 a CUDA kernel.

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 Fig. 14.4B can be executed in parallel, we can simply convert the outer loop into a CUDA kernel by mapping its iterations to CUDA threads. Fig. 14.5 shows a kernel from such a straightforward conversion. Each thread implements an iteration of the original outer loop. That is, we use each thread to calculate the contribution of one *k*-space sample to all F^HD elements. The original outer loop has M iterations, and M can be in the millions. We obviously need to have a large number of 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 (in terms of number of blocks) and `FHD_THREADS_PER_BLOCK` for block size (in terms of number of threads) for kernel invocation. Within the kernel, each thread calculates

```
__global__ void cmpFhD(float* rPhi, iPhi, rD, iD,
    kx, ky, kz, x, y, z, rMu, iMu, rFhD, iFhD, 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 (int n = 0; n < N; n++) {
    floatexpFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);

    floatcArg = cos(expFhD);  floatsArg = sin(expFhD);

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

FIGURE 14.5

First version of the F^HD kernel. The kernel will not execute correctly due to conflicts between threads in writing into rFhD and iFhD arrays.

the original iteration of the outer loop that it is assigned to cover using the familiar 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 innovation would be 65,536/512= 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 Fig. 14.5 exploits ample parallelism, it suffers from a major problem: all threads write into all `rFhD` and `iFhD` voxel elements. This means that the kernel must use 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. As we have seen in Chapter 9, Parallel patterns: parallel histogram computation, heavy use of atomic operation on global memory data can seriously reduce the performance of kernel. Furthermore, the size of the rFhD and iFhD arrays make privatization infeasible. We need to explore other options.

The other option is to use each thread to calculate one `FhD` value from all k-space samples. In order to do so, we need to first swap the inner loop and the outer loop so that each of the new outer loop iterations processes one `FhD` element. That is, each of the new outer loop iterations will execute the new inner loop that accumulates the contribution of all k-space samples to the `FhD` element handled by the outer loop iteration. This transformation of the loop structure is called *loop interchange*. It requires a perfectly nested loop, meaning that there is no statement between the outer `for`-loop statement and the inner `for`-loop statement. This is however, not true for the FhD code in Fig. 14.4B. We need to find a way to move the calculation of `rMu` and `iMu` elements out of the way.

From a quick inspection of Fig. 14.6A which is a replicate of Fig. 14.4B, we see that the F^HD calculation can be split into two separate loops, as shown in Fig. 14.6B 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 itself. As shown in Fig. 14.6B, 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 relative 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^HD. 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.

With loop fission, the F^HD 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 `FHD` elements based on the `rMu` and `iMu` elements calculated in the first step. Each step can now be

(A)

```
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)

```
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;
  }
}
```

FIGURE 14.6

Loop fission on the F^HD computation. (A) F^HD computation, (B) after loop fission.

```
__global__ void cmpMu(float* rPhi, iPhi, rD, iD, rMu, iMu)
{
  int m = blockIdx.x*MU_THREAEDS_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 14.7

cmpMu kernel.

converted into a CUDA kernel. The two CUDA kernels will execute sequentially with respect to each other. 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 Fig. 14.7 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 512 threads in each

block, we will need to use multiple blocks to allow the large number of threads. This can be accomplished by having a number of threads in each block, specified by `MU_THREADS_PER_BLOCK` in Fig. 14.4C, and by employing `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 65,536/512 = 128 blocks. This is done by defining `MU_THREADS_PER_BLOCK` as 512 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 2562nd (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 Fig. 14.6B 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 largest 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.

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 Fig. 14.5. That is, each thread will write into all `rFhD` and `iFhD` elements, thus creating an extremely large number of conflicts between threads. As in the case of Fig. 14.5, the code in Fig. 14.8 requires 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 pair of `rFhD` and `iFhD` elements. This option requires us to interchange the inner and outer loops and then use each thread to implement an iteration of the new outer loop. The transformation is shown in Fig. 14.9. Loop interchange is necessary because the loop being implemented by the CUDA threads must be the outer loop. Loop interchange makes each of the new outer loop iteration to process a pair of `rFhD` and `iFhD` elements. Loop interchange is permissible here because all iterations of both levels of loops are

```
__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 (int 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);

    atomicAdd(&rFhD[n], rMu[m]*cArg - imu[m]*sArg);
    atomicAdd(&rFhD[n], iMu[m]*cArg + imu[m]*sArg);
  }
}
```

FIGURE 14.8

Second option of the F^HD kernel.

(A)

```
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;
  }
}
```

(B)

```
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;
  }
}
```

FIGURE 14.9

Loop interchange of the F^HD computation. (A) Before loop interchange, (B) after loop interchange.

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 of voxels in the reconstructed image, the N value can be very large for higher-resolution images. For a 128^3 image, 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

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

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

  for (int 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 14.10

Third option of the FHD kernel.

`iFhD` elements since every thread has a unique n value. 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.

The kernel derived from the interchanged loops is shown in Fig. 14.10. The threads are organized as a two-level structure. The outer loop has been stripped away; each thread covers an iteration of the outer (n) loop, where n is equal to `blockIdx.x*FHD_THREADS_PER_BLOCK + 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 with a number of threads in each block, specified by a global constant `FHD_THREADS_PER_BLOCK`. Assuming that `N` is the variable that stores the number of voxels in the reconstructed image, `N/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 65,536/512 = 128 blocks. This is done by assigning 512 to `FHD_THREADS_PER_BLOCK` and using `FHD_THREADS_PER_BLOCK` as block size and `N/FHD_THREADS_PER_BLOCK` as grid size during kernel innovation.

STEP 2: GETTING AROUND THE MEMORY BANDWIDTH LIMITATION

The simple cmpFhD kernel in Fig. 14.10 will result in limited speedup due to memory bandwidth limitations. A quick 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 multiply, 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 our analysis in Chapter 5, Performance considerations.

We can immediately improve the compute to memory access ratio by assigning some of the array elements to automatic variables. As we discussed in Chapter 5, Performance considerations, the automatic variables will reside in registers, thus converting reads and writes to the global memory into reads and writes to on-chip registers. A quick review of the kernel in Fig. 14.10 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 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 Fig. 14.11. By increasing the number of registers used by five for each thread, we have reduced the memory access done in each iteration from 14 to 7. Thus, we have increased the compute to memory access ratio from 13:14 to 13:7. This is a good improvement and a good use of the precious register resource.

Recall that the register usage can limit the occupancy, number of blocks that can run in an streaming multiprocessor (SM). By increasing the register usage by 5 in the kernel code, we increase the register usage of each thread block 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 65,536 registers among all blocks assigned to it (in SM Version 3.5 or higher), we need be careful, as any further increase of register usage can begin to limit the number of blocks that can be assigned to an SM. Fortunately, the register usage is not a limiting factor to parallelism for this kernel.

We want to further improve the compute to memory access ratio to something closer to 10 by eliminating more 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` are accessed in each iteration of the loop in Fig. 14.11. This means that we cannot load a k-space element into a register and expect to access that element off a register through all the iterations. So, 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.

An analysis of the loop in Fig. 14.11 reveals that the k-space elements are indeed excellent candidates for constant memory. The index used for accessing kx, ky, and kz is m. We know that m is independent of `threadIdx`, which implies that all threads in a warp will be accessing the same element of `kx`, `ky`, and `kz`. This is an ideal access pattern for cached constant memory: every time an element is brought into the cache, it will be used at least by all 32 threads in a warp for a current generation device. 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 to effectively eliminate 96% or more of

```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
       kx, ky, kz, 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 (int 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] = rFhD_r; iFhD[n] = iFhD_r;
}
```

FIGURE 14.11

Using registers to reduce memory accesses in the F^HD kernel.

the accesses to the global 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 makes constant memory almost as efficient as registers for accessing k-space elements.[2]

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 64 kB. 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 into chunks or 64 kB or smaller. The developer must reorganize the kernel so that the kernel will be invoked multiple times, with each invocation of the kernel consuming only a chunk of the large data set. This turns out to be quite easy for the cmpFhD kernel.

A careful examination of the loop in Fig. 14.11 reveals that all threads will sequentially march through the k-space arrays. That is, all threads in the grid access the same k-space element during each iteration. 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 64 kB capacity of the constant memory.[3] The host code now invokes the kernel

[2] The reason why a constant memory access is not exactly as efficient as a register access is that a memory load instruction is still needed for access the constant memory.

[3] Note not all accesses to read-only data are as favorable for constant memory as what we have here. In Chapter 12, Parallel patterns: graph search, we present a case where threads in different blocks access different elements in the same iteration. This more diverged access pattern makes it much harder to fit enough of the data into the constant memory for a kernel launch.

```
__constant__ float   kx_c[CHUNK_SIZE],
                     ky_c[CHUNK_SIZE],
                     kz_c[CHUNK_SIZE];
…
void main() {

  for (int i = 0; i < M/CHUNK_SIZE; i++);
    cudaMemcpyToSymbol(kx_c,&kx[i*CHUNK_SIZE],4*CHUNK_SIZE,
                       cudaMemCpyHostToDevice);
    cudaMemcpyToSymbol(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
                       cudaMemCpyHostToDevice);
    cudaMemcpyToSymbol(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
                       cudaMemCpyHostToDevice);
    …
    cmpFHD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>>
       (rPhi, iPhi, phiMag, x, y, z, rMu, iMu, CHUNK_SIZE);
  }
  /* Need to call kernel one more time if M is not */
  /* perfect multiple of CHUNK SIZE */
}
```

FIGURE 14.12

Chunking k-space data to fit into constant memory.

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 Fig. 14.12. (For more recent devices and CUDA versions, a "`const__restrict__`" declaration of kernel parameters makes the corresponding input data available in the "read-only data" cache, which is a simpler way of getting the same effect as using constant memory.)

In Fig. 14.12, the `cmpFhD` kernel is called from a loop. The code assumes that `kx`, `ky`, and `kz` arrays are in the host memory. The dimension of kx, ky, and kz is given by `M`. At each iteration, the host code calls the `cudaMemcpyToSymbol()` 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 `cudaMemcpyToSymbol()` and one more kernel invocation to finish the remaining k-space data.

Fig. 14.13 shows a 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. The `kx_c`, `ky_c`, and `kz_c` arrays are accessed as global variables declared under `__constant__` keyword as shown in Fig. 14.12. By accessing these elements from the constant cache, the kernel now has effectively only four 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 reduced to two. The compute to memory access ratio is up to 13:2.

```
__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 (int m = 0; m < M; m++) {
    float expFhD =
       2*PI*(kx_c[m]*xn_r+ky_c[m]*yn_r+kz_c[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] = rFhD_r; iFhD[n] = iFhD_r;
}
```

FIGURE 14.13

Revised F^HD kernel to use constant memory.

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.

If we ran the code in Figs. 14.12 and 14.13, we would have found out that the performance enhancement was not as high as we expected for some devices. 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 Fig. 14.14A, each constant cache entry is designed to store multiple consecutive words. This design reduces the cost of constant cache hardware. When an element is brought into the cache, several elements around it are also brought into the cache. This is illustrated in shaded sections surrounding the `kx[i]`, `ky[i]`, and `kz[i]`, which is shown as dark boxes in Fig. 14.14. Three cache lines in the constant cache are needed to support the efficient execution of each iteration of a warp.

In a typical execution, we will have a fairly large number of warps that are concurrently executing on an SM. Since different warps can be at very different iterations, they may require many entries altogether. For example, if we define each thread block to have 512 threads and expect to assign three blocks to execute concurrently in each SM, we will have (512/32)*3 = 48 warps executing concurrently in an SM. If each of them requires a minimal of three cache lines in the constant cache to sustain efficient execution, in the worst case, we need a total of 48*3 = 144 cache

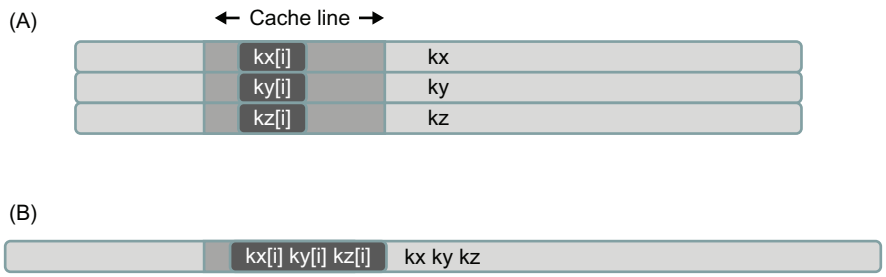


FIGURE 14.14

Effect of k-space data layout on constant cache efficiency. (A) k-space data stored in separate arrays, (B) k-space data stored in an array whose elements are structs.

lines. Even if we assume that on average, three warps will be executing at the same iteration and thus can share cache lines, we still need 48 cache lines. This is referred to as the working set of all the active warps.

Due to cost constraints, the constant caches of some devices have a small number of cache lines, say 32. When there are not enough cache lines to accommodate the entire working set, the data being accessed by different warps begin to compete with each other for the cache lines. By the time a warp moves to its next iteration, the next elements to be accessed have already been purged to make room for the elements accessed by other warps. As it turns out, the constant cache capacity in some devices indeed has insufficient number of entries to accommodate the entries for all the warps active in an SM. As a result, the constant cache fails to eliminate many of the global memory accesses.

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 Fig. 14.14B and the code based on this solution in Fig. 14.15. Rather than having the x, y, and z components of the k-space data stored in three separate arrays, the solution stores these components in an array whose elements comprise a `struct`. In the literature, this style of declaration is often referred to as *array of structs*. The declaration of the array is shown on top of Fig. 14.15. 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 of a warp 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. Assuming each k-space sample is a single precision floating-point number, the size of the transfer is adjusted from `4*CHUNK_SIZE` to `12*CHUNK_SIZE` to reflect the transfer of all three components in one `cudaMemcpyToSymbol` call.

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

__constant__ struct kdata k_c[CHUNK_SIZE];
…

void main() {

 for (int i = 0; i < M/CHUNK_SIZE; i++){
   cudaMemcpyToSymbol(k_c,k,12*CHUNK_SIZE,cudaMemCpyHostToDevice);

   cmpFhD<<<FHD_THREADS_PER_BLOCK, N/FHD_THREADS_PER_BLOCK>>>(…);

  }
 }
```

FIGURE 14.15

Adjusting k-space data layout to improve cache efficiency.

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 Fig. 14.16. Note that `kx[m]` has become `k[m].x`, `ky[m]` has become `k[m].y`, and so on. This small change to the code can result in significant enhancement of its execution speed on some devices.[4]

STEP 3: USING HARDWARE TRIGONOMETRY FUNCTIONS

CUDA offers hardware implementations of mathematic functions that provide much higher throughput than their software counterparts. These functions are implemented as hardware instructions executed by the SFU (special function units). 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()` (two "_" characters in front of the function name). These are intrinsic functions that are recognized by the compiler and translated into SFU instructions. Because these functions are called in a heavily executed loop body, we expect that the change will result in a very significant performance improvement. The resulting `cmpFhD` kernel is shown in Fig. 14.17.

[4]The reader might notice that the adjustment from multiple arrays to an array of structure is opposite to what is often done to global memory data. When adjacent threads in a warp access consecutive elements of an array of structure, it is much better to store the fields of the structure into multiple arrays so that the memory accesses are coalesced. The key difference here is that all threads in a warp are accessing the same elements.

```
__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 (int 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] = rFhD_r; iFhD[n] = iFhD_r;
}
```

FIGURE 14.16

Adjusting for the k-space data memory layout in the F^HD kernel.

```
__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 (int 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] = rFhD_r; iFhD[n] = iFhD_r;
}
```

FIGURE 14.17

Using hardware `__sin()` and `__cos()` functions.

However, we need to be careful about the reduced accuracy when switching from software functions to hardware functions. As we discussed in Chapter 6, Numerical considerations, hardware implementation currently has less accuracy than software libraries (the details are available in the CUDA C Programming

$$MSE = \frac{1}{mn}\sum_i \sum_j (I(i,\ j) - I_0(i,\ j))^2$$

$$PSNR = 20 \log_{10}\left(\frac{\max(I_0(i,\ j))}{\sqrt{MSE}}\right)$$

FIGURE 14.18

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.

Guide). In the case of MRI, we need to make sure that the hardware implementation passes provide enough accuracy, as defined in Fig. 14.18. The testing process involves a "perfect" image (I_0) of a fictitious object, sometimes referred to as a *phantom* object. We use a reverse process to generate a corresponding "scanned" k-space data that is synthesized. The synthesized scanned data is then processed by the proposed reconstruction system to generate a reconstructed image (I). The values of the voxels in the perfect and reconstructed images are then fed into the peak SNR (PSNR) formula in Fig. 14.18.

The criteria for passing the test depend on the application that the image is intended for. In our case, we worked with experts in clinical MRI to ensure that the PSNR changes due to hardware functions are well within the accepted limits for their applications. In applications where the images are used by physicians to form an impression of injury or evaluate a disease, one also needs to have visual inspection of the image quality. Fig. 14.19 shows the visual comparison of the original "true" image. It then shows that the PSNR achieved by CPU double precision and single precision implementation are both 27.6 dB, well above the acceptable level for the application. A visual inspection also shows that the reconstructed image indeed corresponds well with the original image.

The advantage of iterative reconstruction compared to a simple bilinear interpolation gridding/iFFT is also obvious in Fig. 14.19. The image reconstructed with the simple gridding/iFFT has a PSNR of only 16.8 dB, substantially lower than the PSNR of 27.6 dB achieved by the iterative reconstruction method. A visual inspection of the gridding/iFFT image in Fig. 14.19(2) shows that there are severe artifacts that can significantly impact the usability of the image for diagnostic purposes. These artifacts do not occur in the images from the iterative reconstruction method.

When we moved from double precision to single precision arithmetic on the CPU, there was no measurable degradation of PSNR, which remains at 27.6 dB. When we moved the trigonometry function from software library to the hardware units, we observed a negligible degradation of PSNR, from 27.6 dB to 27.5 dB. The slight loss of PSNR is within an acceptable range for the application. A visual inspection confirms that the reconstructed image does not have significant artifacts compared to the original image.

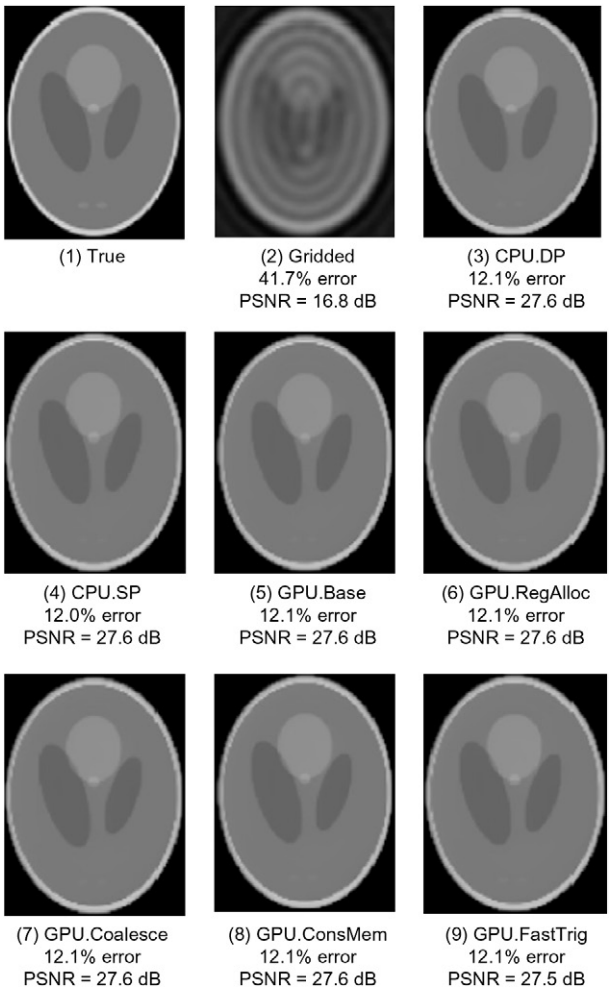


FIGURE 14.19

Validation of floating-point precision and accuracy of the different F^HD implementations.

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 optimal number of threads for each block. On one hand, using a large number of threads in a block is needed to fully utilize the thread capacity of each SM (given that sixteen blocks can be assigned to each SM at maximum). 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 could also consider nonpower-of-two numbers.

Another kernel configuration parameter is the number of times one should unroll the body of the for-loop. This can be set using a "#pragma unroll" followed by the number of unrolls we want the compiler to perform on a 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 potentially increase the usage of registers and reduce the number of blocks that can fit into an SM.

Note that the effects of these configurations are not isolated from 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 only explores some promising trends. Ryoo, et al. present a Pareto-Optimal-Curve-based method to screen away most of the inferior combinations using [RRS 2008].

14.4 FINAL EVALUATION

To evaluation the advantage of each alternative approach, we can use a sample data set 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 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 PSNR 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.

To facilitate comparison of the iterative reconstruction with a conventional reconstruction, we also evaluated a reconstruction based on bilinear interpolation gridding and iFFT. Our version of the gridded reconstruction is not optimized for speed, but it is already quite fast. For example, the total reconstruction time for the test image using bilinear interpolation gridding followed by iFFT takes less than 1 minute on a high-end sequential CPU. It is, however, obvious from Fig. 14.19(2) that the resulting image exhibits an unacceptable level of artifacts. It should be noted that the quality of the reconstruction can be improved with more sophisticated convolutional gridding methods at increased computation cost.

The actual execution time of the reconstruction steps will of course vary across devices. Therefore, we will discuss the results in approximate terms. For the preparation of the system for each patient, the sequential Q computation for our experimental input and output takes tens of hours on a high-end CPU. This time is reduced to a few minutes on the GPU with all the optimizations described in Section 14.3.

The total reconstruction of each image time using a sequential F^HD implementation on a high-end CPU requires a few hours. This time is reduced to about 3 minutes using the final version of the cmpFhD kernel on a high-end GPU. A naïve implementation of the cmpFhD would result in a reconstruction time of about 30 minutes on a high-end GPU. There is about 10 × speed improvement going from the naïve version to the final version as discussed in Section 14.3.

An interesting observation is that in the end, the CG solver (the find ρ step in Fig. 14.3) can actually take more time than F^HD. This is because we have accelerated F^HD dramatically. Any further acceleration will now require acceleration of the CG solver. Before parallelization, F^HD used to account for nearly 100% of the execution time. After successful parallelization, it only accounts for about 50%. The other 50% is largely spent in the CG solver. This is a well-known phenomenon in parallelizing real applications. As some phases of the execution are accelerated by successful parallelization efforts, the execution time becomes dominated by other phases that used to account for insignificant portions of the execution.

14.5 EXERCISES

1. Loop fission splits a loop into two loops. Use the F^HD code in Fig. 14.4B and enumerate the execution order of the two parts of the outer loop body: (1) the statements before the inner loop and (2) the inner loop. (1) List the execution order of these parts from different iterations of the outer loop before fission. (2) List the execution order of these parts from the two loops after fission. Determine if the execution results will be identical. The execution results are identical if all data required by a part is properly generated and preserved for its consumption before that part executes, and the execution result of the part is not overwritten by other parts that should come after the part in the original execution order.

2. Loop interchange swaps the inner loop into the outer loop and vice versa. Use the loops from Fig. 14.9 and enumerate the execution order of the instances of loop body before and after the loop exchange. (1) List the execution order of the loop body from different iterations before loop interchange. Identify these iterations with the values of m and n. (2) List the execution order of the loop body from different iterations after loop interchange. Identify these iterations with the values of m and n. Determine if the (1) and (2) execution results will be identical. The execution results are identical if all data required by a part is properly generated and preserved for its consumption before that part executes, and the execution result of the part is not overwritten by other parts that should come after the part in the original execution order.

3. In Fig. 14.11, identify the difference between the access to x[] and kx[] in the nature of indices used. Use the difference to explain why it does not make sense to try to load kx[n] into a register for the kernel shown in Fig. 14.11.

4. During a meeting, a new graduate student told his advisor that he improved his kernel performance by using cudaMalloc() to allocate constant memory and using cudaMemcpy() to transfer read-only data from the CPU memory to the constant memory. If you were his advisor, what would be your response?

REFERENCES

Liang, Z. P., & Lauterbur, P. (1999). *Principles of magnetic resonance imaging: A signal processing perspective*. New York: John Wiley & Sons, Inc.

Ryoo, S., Ridrigues, C. I., Stone, S. S., Stratton, J. A., Ueng, Z., Baghsorkhi, S. S., et al. (2008). Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing* http://dx.doi.org/10.1016/j.jpdc.2008.05.011.

Stone, S. S., Haldar, J. P., Tsao, S. C., Hwu, W. W., Sutton, B. P., & Liang, Z. P. (2008). Accelerating advanced MRI reconstruction on GPUs. *Journal of Parallel and Distributed Computing* http://dx.doi.org/10.1016/j.jpdc.2008.05.013.

This page intentionally left blank