CHAPTER 17

Iterative magnetic resonance imaging reconstruction

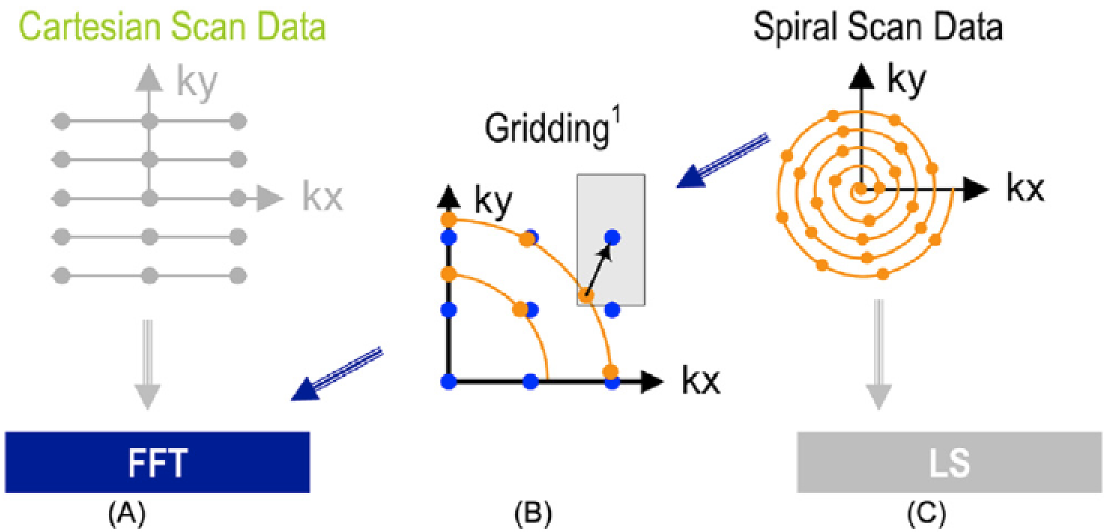
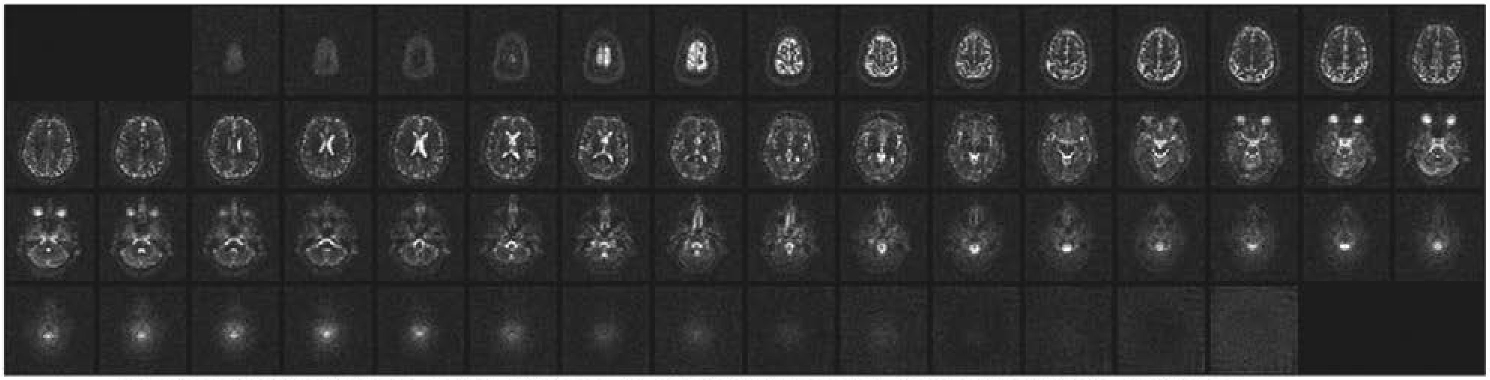


FIGURE 17.1

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 a linear solver—based reconstruction.



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

FIGURE 17.2

Non-Cartesian k-space sample trajectory and accurate linear solver–based reconstruction enable new capabilities with exciting medical applications.

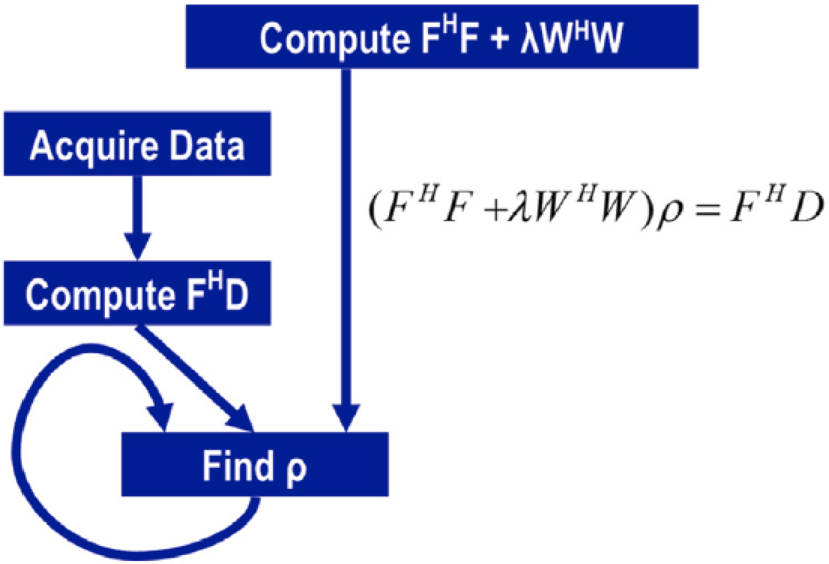


FIGURE 17.3

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

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

FIGURE 17.4

Computation of $F^H D$.

```
#define FHD_THREADS_PER_BLOCK 1024
__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++) {
     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(&iFhD[n],iMu[m]*cArg + rMu[m]*sArg);
  }
}
```

FIGURE 17.5

First version of the F^HD kernel.

```
for (int 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 (int m = 0; m < M; m++) {
   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);
      rFhD[n] +=  rMu[m]*cArg - iMu[m]*sArg;
      iFhD[n] +=  iMu[m]*cArg + rMu[m]*sArg;
   }
}
```

FIGURE 17.6

Loop fission on the F^HD computation.

```
#define MU_THREADS_PER_BLOCK 1024
__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 17.7

The cmpMu kernel.

```
#define FHD_THREADS_PER_BLOCK 1024
__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(&iFhD[n],iMu[m]*cArg + rMu[m]*sArg);
  }
}
```

FIGURE 17.8

Second option of the $F^H D$ kernel.

```
for (int n = 0; n < N; n++) {
   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 17.9

Loop interchange of the F^HD computation.

```
#define FHD_THREADS_PER_BLOCK 1024
__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 17.10

Third option of the FHD kernel.

```
#define FHD_THREADS_PER_BLOCK 1024
__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;
               // assign frequently accessed coordinate and output
               // elements into registers
  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 17.11

Using registers to reduce memory accesses in the $F^H D$ kernel.

```
__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(kz_c,&kz[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 17.12

Host code sequence for chunking k-space data to fit into constant memory.

```
#define FHD_THREADS_PER_BLOCK 1024
__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 17.13

Revised F^HD kernel to use constant memory.

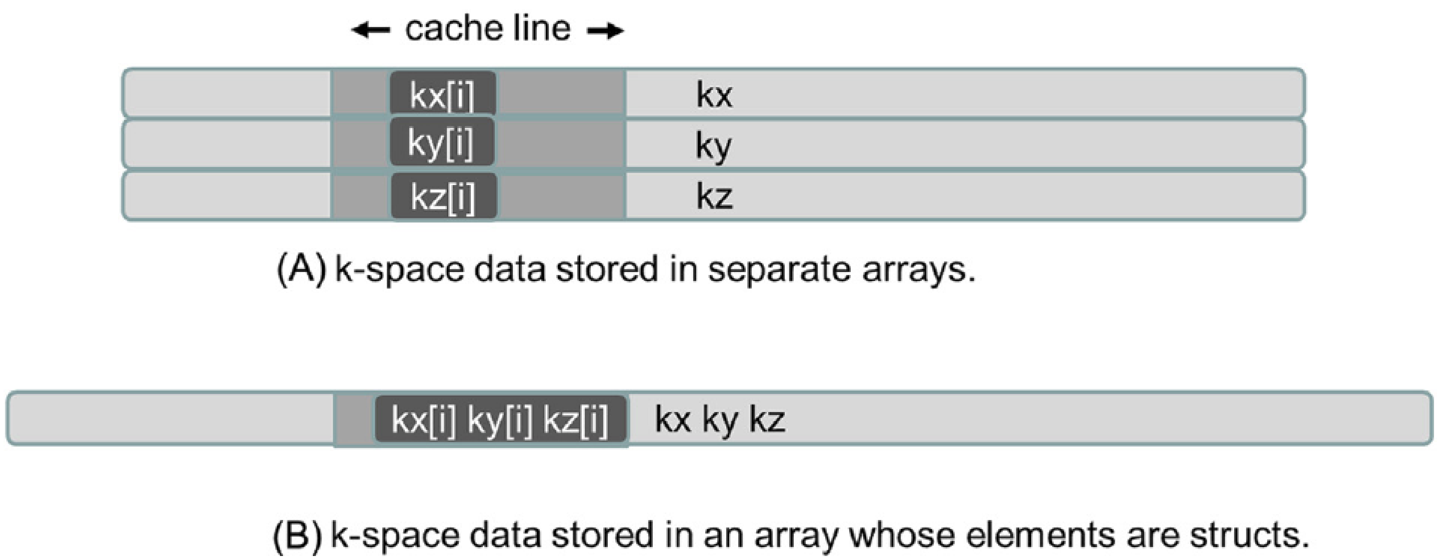


FIGURE 17.14

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.

```
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 17.15

Adjusting k-space data layout to improve cache efficiency.

```
__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 17.16

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

```
#define FHD_THREADS_PER_BLOCK 1024
__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 17.17

Using hardware __sin() and __cos() functions.

$$MSE = \frac{1}{mn}\sum_{i}\sum_{j}(I(i,j) - I_0(i,j))^2 \qquad PSNR = 20\log_{10}\left(\frac{\max(I_0(i,j))}{\sqrt{MSE}}\right)$$

FIGURE 17.18

Metrics used to validate the accuracy of hardware functions. I_0 is the perfect image. I is the reconstructed image. PSNR is peak signal-to-noise ratio.

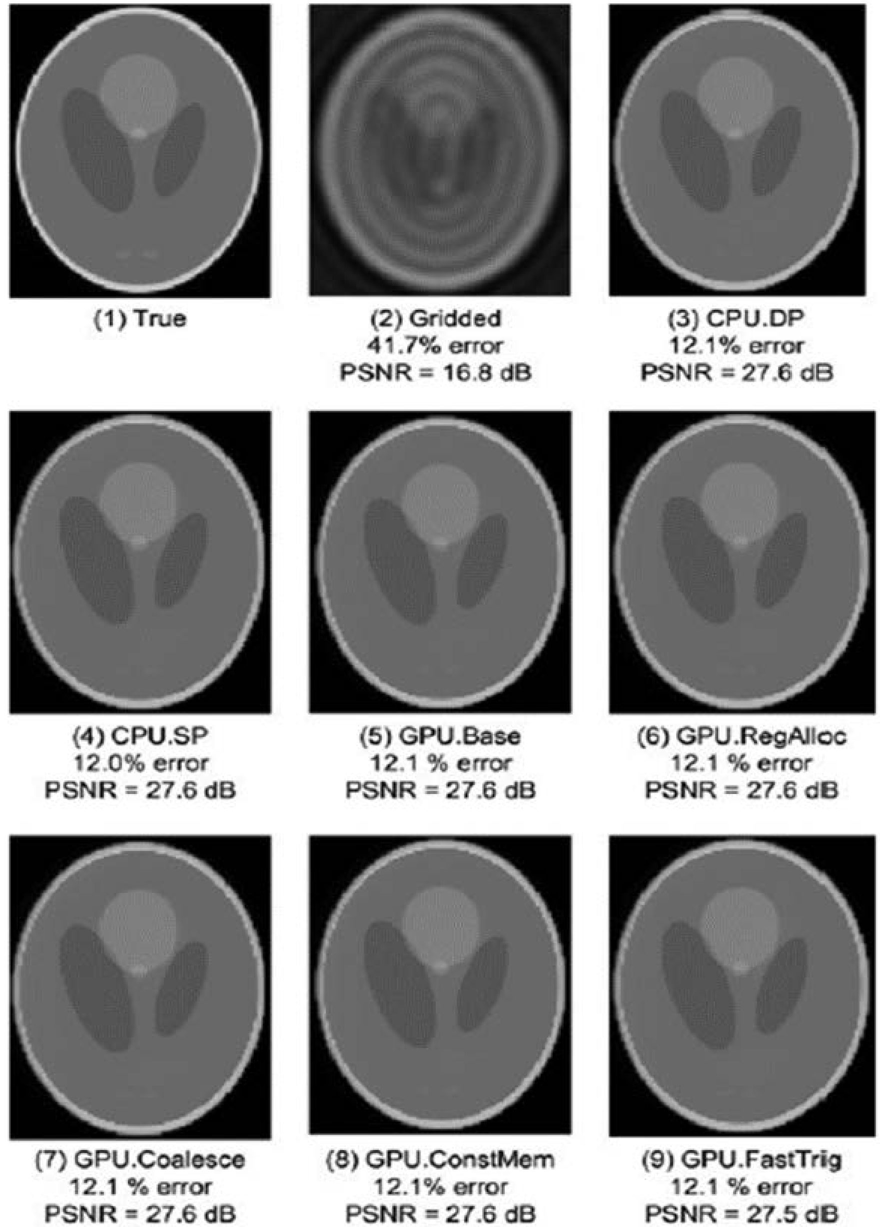


FIGURE 17.19

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