Chapter-14
Application case study— non-Cartesian magnetic resonance imaging
An introduction to statistical estimation methods

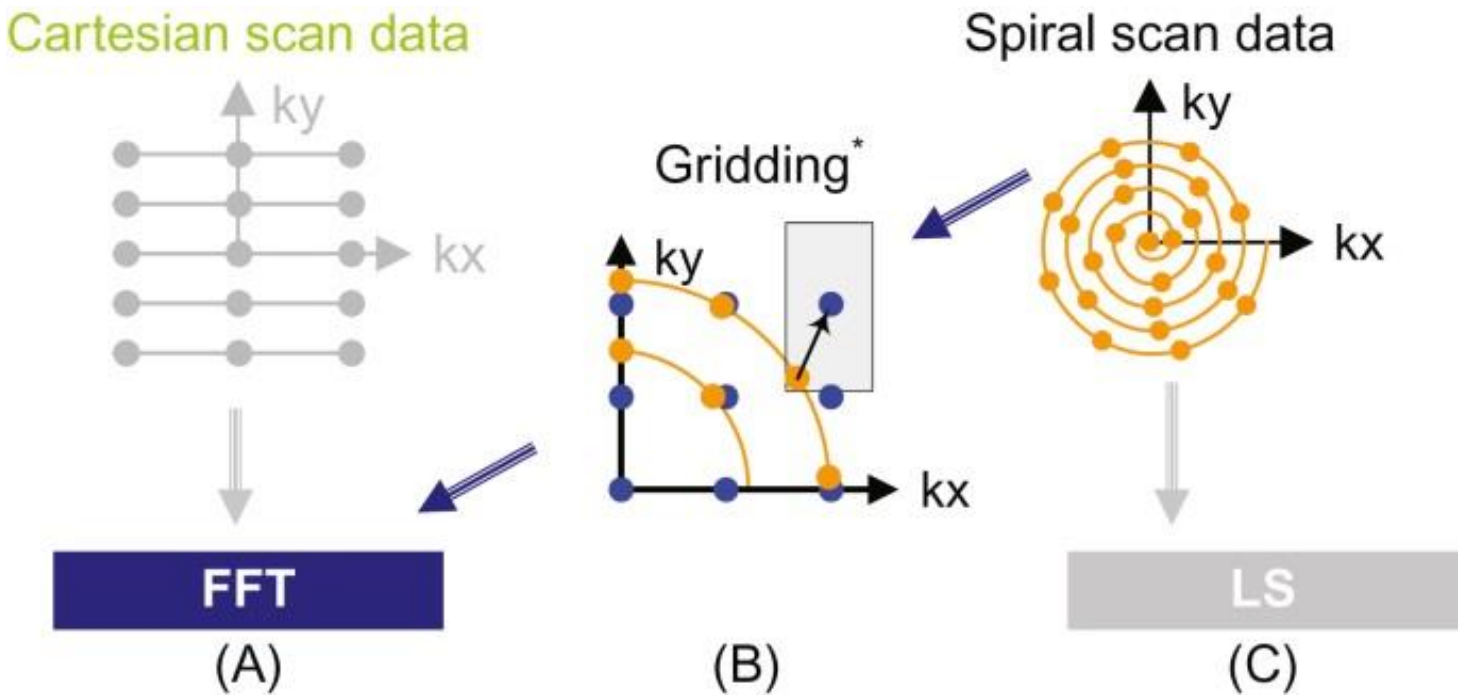
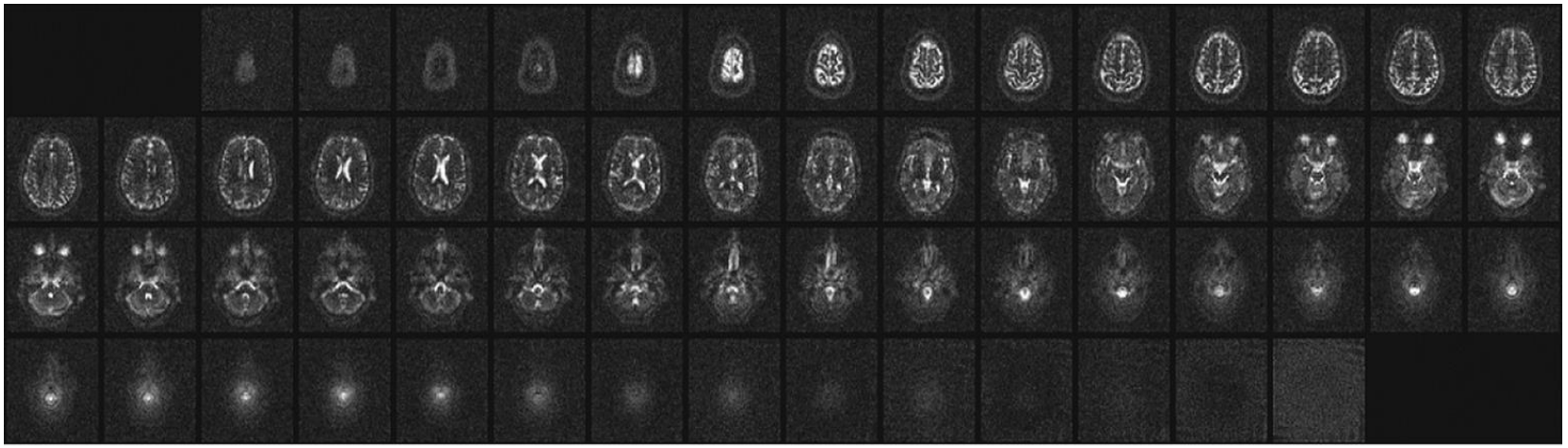


FIGURE 14.1: Scanner k-space trajectories and their associated reconstruction strategies: (A) Cartesian trajectory with FFT reconstruction, (B) Spiral (or non-Cartesian trajectory in general) followed by gridding to enable FFT reconstruction, (C) spiral (non-Cartesian) trajectory with linear solver based 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.

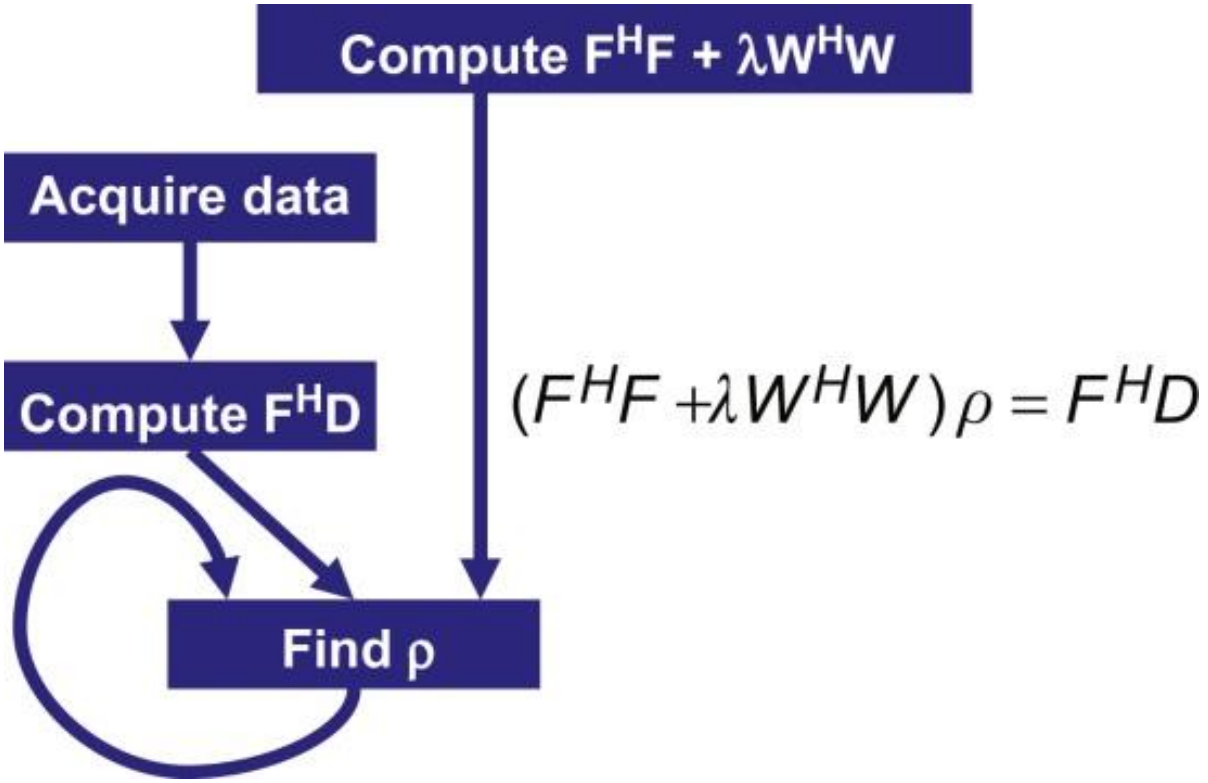


FIGURE 14.3: An iterative linear-solver-based approach to reconstructing non-Cartesian k-space sample data.

(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 FHD. (A) Q computation, (B) FHD computation.

```
__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 FHD kernel. The kernel will not execute correctly due to conflicts between threads in writing into rFhD and iFhD arrays.

(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 FHD computation. (A) FHD 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.

```
__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 FHD 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 FHD computation. (A) Before loop interchange, (B) after loop interchange.

```
__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.

```
__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 FHD 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(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.

```
__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 FHD kernel to use constant memory.

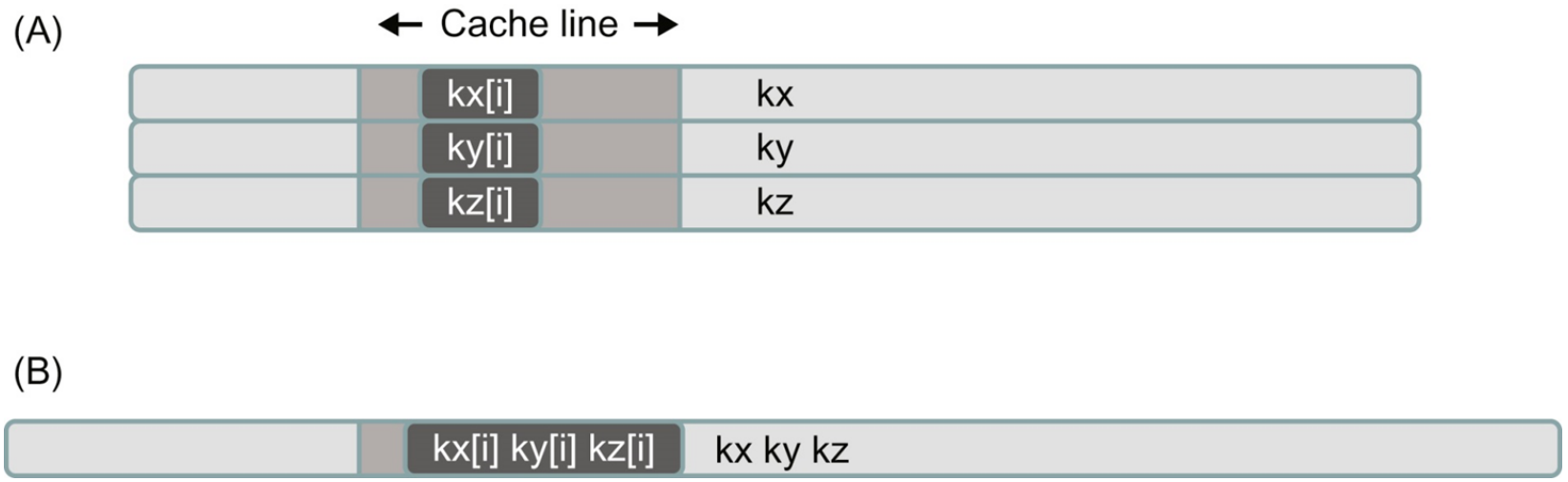


FIGURE 14.14: Effect of k-space data layout on constant cache efficiency. (A) k-space data stored in separate arrays, (B) k-space data stored in an array whose elements are structs.

```
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.

```
__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 FHD 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.

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

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

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.

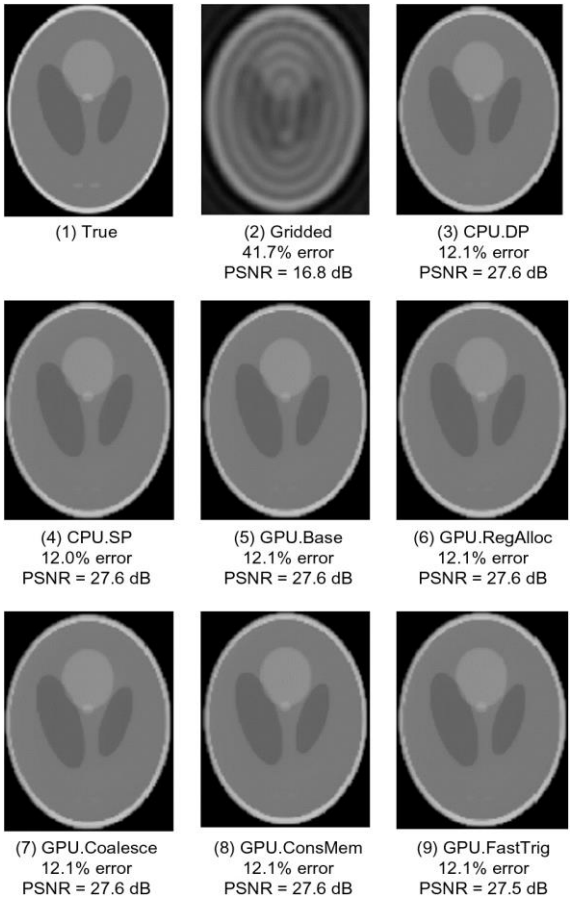


FIGURE 14.19: Validation of floating-point precision and accuracy of the different FHD implementations.