GPU Teaching Kit

Accelerated Computing

Module 15 - Application Case Study – Advanced MRI Reconstruction

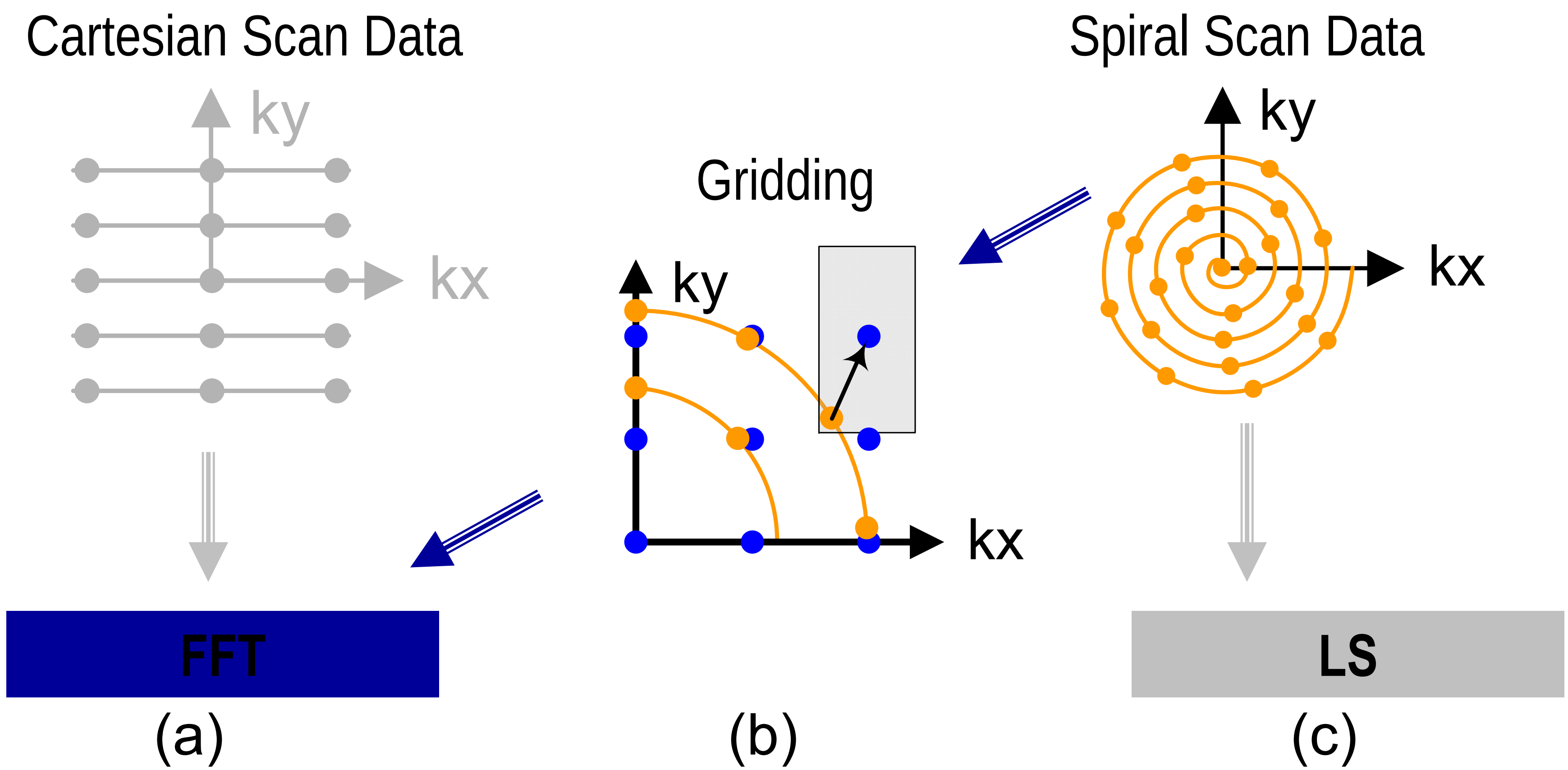
Lecture 15.1 - Advanced MRI Reconstruction

Objective

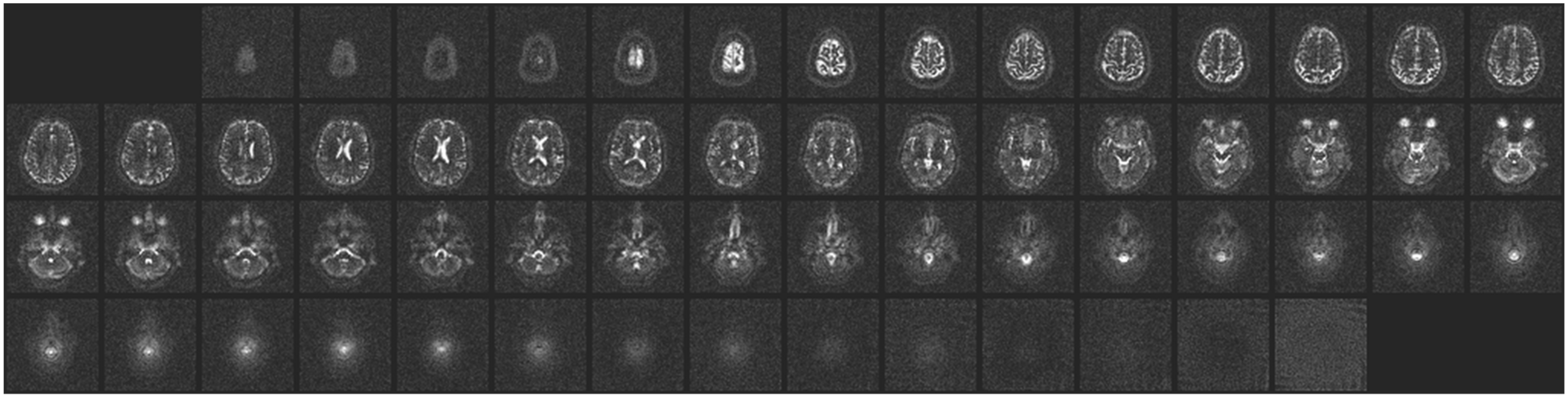
- To learn how to apply parallel programming techniques to an application
  - Determining parallelism structure
  - Loop transformations
  - Memory layout considerations
  - Validation

Non-Cartiesian MRI Scan

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

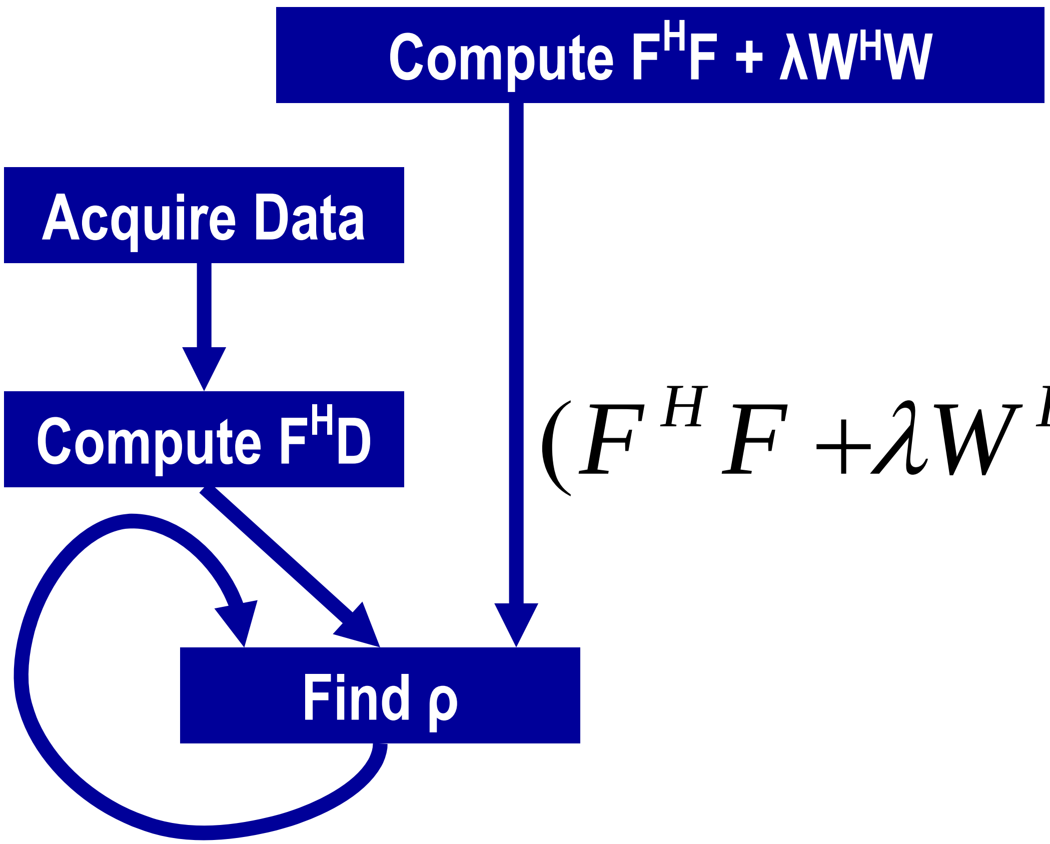


Non-Cartesian Scan



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

An Iterative Solver Based Approach to Image Reconstruction



$$(F^H F + \lambda W^H W)\rho = F^H D$$

Computation of Q and FHD

```
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

```
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^HD computation

First Version of the FHD Kernel.

```
__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 (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;
  }
}
```

Loop Fission

```
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;
  }
}
```

(a) F^HD computation

```
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;
  }
}
```

(b) after loop fission

cmpMu Kernel

```
__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];
}
```

Second Option of the FHD 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 (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;
  }
}
```

Loop Interchange of the FHD Computation

```
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
```

```
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
```

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;

  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;
  }
}
```

Using Registers to Reduce Memory Accesses

```
__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 (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;
}
```

Chunking k-space Data to Fit into Constant Memory

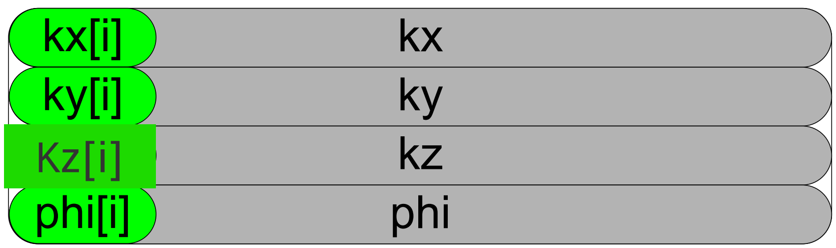
```
__constant__ float  kx_c[CHUNK_SIZE],
                    ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];
…
__ void main() {

  int i;
  for (i = 0; i < M/CHUNK_SIZE; i++);
    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 */
}
```

Revised FHD Kernel – 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 (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;
}
```

Constant Memory Layout Consideration

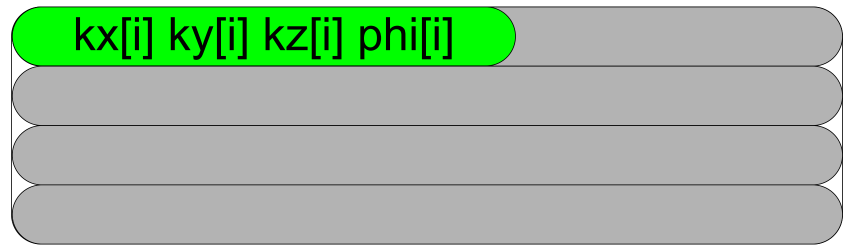
Scan Data



Constant Memory

(a) k-space data stored in separate arrays

Scan Data



Constant Memory

(b) k-space data stored in an array whose elements are structs.

- Storing k-space samples in three separate arrays requires 3 cache lines for each warp
  - Interleaving x, y, z values of the same sample in the same array reduces the cache line requirement to 1 per warp

Host Code with Adjusted Constant Memory Layout

```
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++);
    cudaMemcpyToSymbol(k_c,k,12*CHUNK_SIZE,
       cudaMemcpyHostToDevice);

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

Adjusted k-space data constant 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 (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;
}
```

Using Hardware __sin() and __cos()

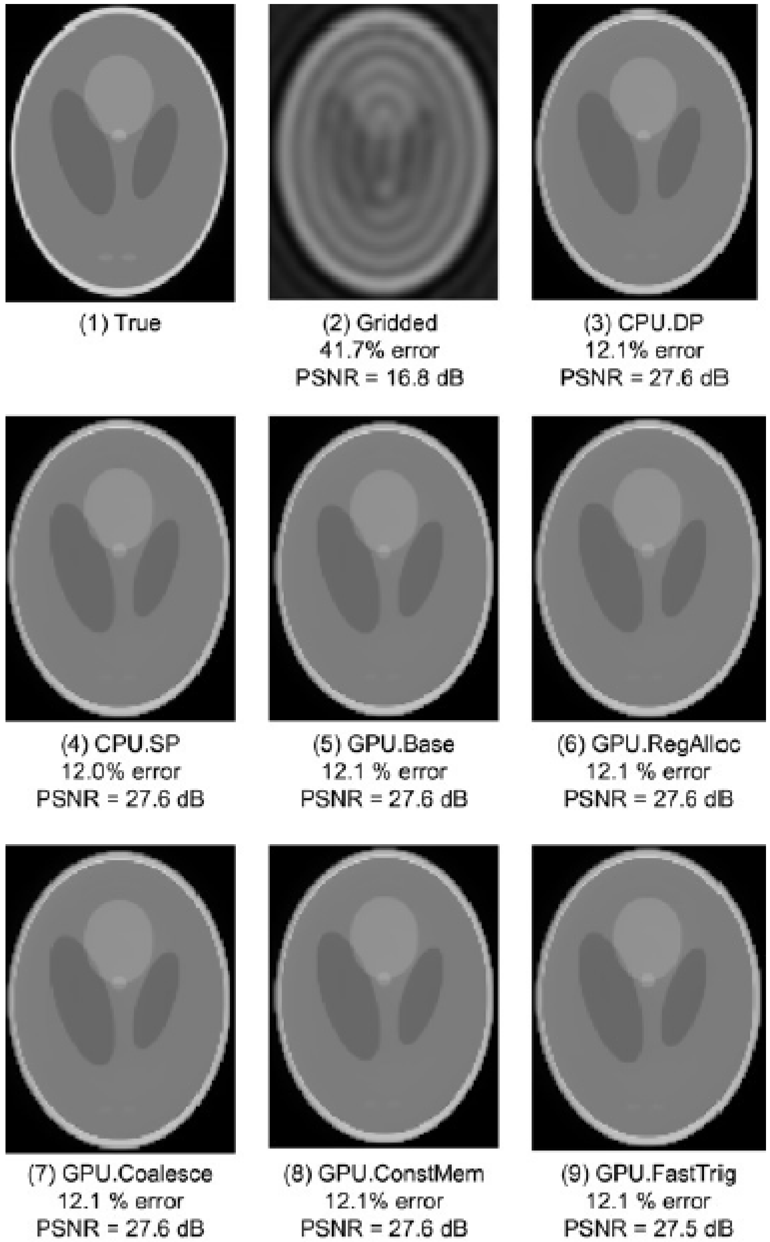
```
__global__ void cmpFhD(float* rPhi, iPhi, phiMag,
      x, y, z, rMu, iMu, int M) {

  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] = rFhD_r; iFhD[n] = iFhD_r;
}
```

Validating Reconstructed Image Using Peak Signal-to-Noise Ratio

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

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

I_0 is a known, "perfect" answer. This is typically done by creating k-space samples for a known image, producing a reconstructed image, and compare the two.



(1) True

(2) Gridded
41.7% error
PSNR = 16.8 dB

(3) CPU.DP
12.1% error
PSNR = 27.6 dB

(4) CPU.SP
12.0% error
PSNR = 27.6 dB

(5) GPU.Base
12.1 % error
PSNR = 27.6 dB

(6) GPU.RegAlloc
12.1 % error
PSNR = 27.6 dB

(7) GPU.Coalesce
12.1 % error
PSNR = 27.6 dB

(8) GPU.ConstMem
12.1% error
PSNR = 27.6 dB

(9) GPU.FastTrig
12.1 % error
PSNR = 27.5 dB

Title ???

Validation of floating-point precision and accuracy of the different $F^H D$ implementations.

(1) Is the known image answer (sometimes called a phantom image)

Note that all GPU optimized versions have comparable PSNR as (3) the CPU double-precision version

Slide 21

AS3 Comment from Joe Bungo - Should the slide have a title?
Andrew Schuh, 3/12/2016

Component and Whole-Application Speedup

	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

GPU Teaching Kit

Accelerated Computing