Many-core Processors for Science and Engineering Applications

Lecture 7: Application Case Study - Quantitative MRI Reconstruction

Objective

- To learn about computational thinking skills through a concrete example
  - Problem formulation
  - Designing implementations to steer around limitations
  - Validating results
  - Understanding the impact of your improvements
- A top to bottom experience!

The Illinois MRI Acceleration Team

Faculty

- Zhi-Pei Liang
- Brad Sutton
- Keith Thulborn
- Ian Atkinson

- Wen-mei Hwu
- John Stone

Students

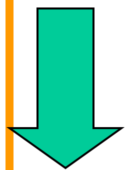
- Justin Haldar
- Yue Zhuo
- Fan Lam

- Sam Stone
- Deepthi Nandakumar
- Xiao-Long Wu
- Nady Obeid
- Haoran Yi
- Stephanie Tsao

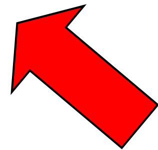
MRI Pipeline

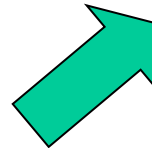
Speedup can enable new applications

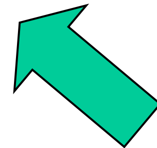
Interpretation



Data Acquisition







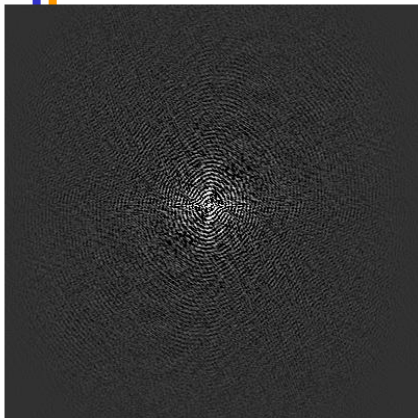
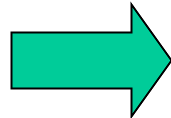
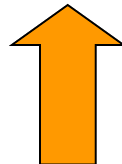
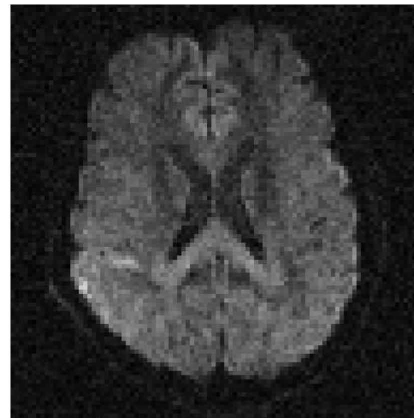


Image Reconstruction

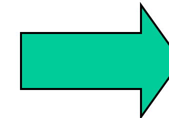


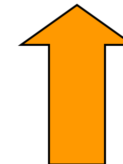


GPU

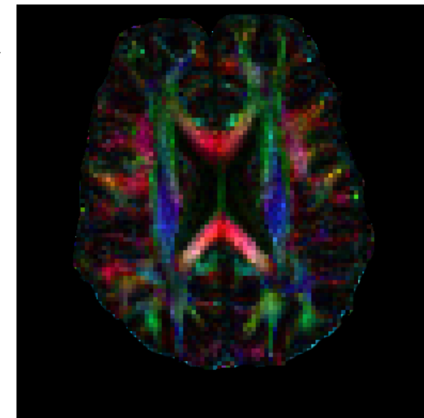


Parameter Estimation





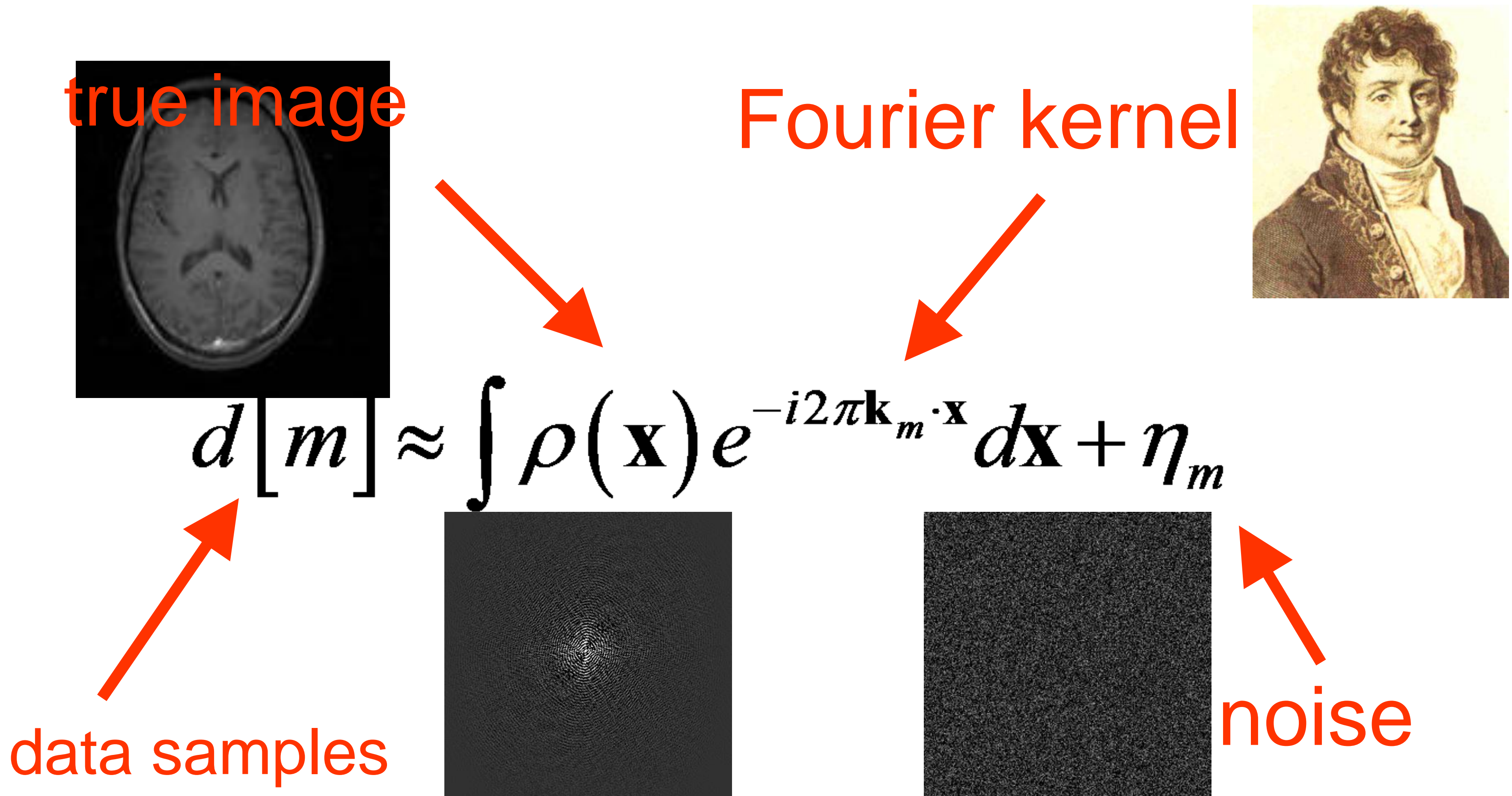
GPU



© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

MRI data in Fourier Space

- Ignoring several effects, MRI image and signal are a Fourier transform pair



© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Discretization

- Infinite dimensional variables are inconvenient for computation

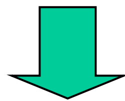
$$\rho(\mathbf{x}) = \sum_{n=1}^{N} \rho_n \varphi(\mathbf{x} - \mathbf{x}_n)$$

Finite dimensional image representation

voxel basis function

$$d[m] \approx \int \rho(\mathbf{x}) e^{-i2\pi \mathbf{k}_m \cdot \mathbf{x}} d\mathbf{x} + \eta_m$$

Integral equation



$$\mathbf{d} = \mathbf{F}\boldsymbol{\rho} + \boldsymbol{\eta}$$

Matrix equation

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Very Large Matrix Equations

$$\mathbf{d} = \mathbf{F}\boldsymbol{\rho} + \boldsymbol{\eta}$$

- Typical 2D images: N of ρ =256x256
- Typical 3D images: N of ρ =256x256x256

If thinking in megapixels – this is a low res camera

F Matrix entries are complex floats, so storage of matrix (single precision):

2D: dimension of F is $(256X256)^2$ ~ 34 GB

3D: $(256X256X256)^3$ ~ 2 PB

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Reconstructing Fourier Data

- Two main approaches:
  - Riemann approximation of the continuous inverse FT:

density compensation

$$\hat{\boldsymbol{\rho}} = \mathbf{F}^H (\mathbf{w} \odot \mathbf{d}) = \sum_{m=1}^{M} w_m d[m] e^{i2\pi \mathbf{k}_m \cdot \mathbf{x}}$$

  - Solve a regularized inverse problem, e.g.,

$$\hat{\boldsymbol{\rho}} = \arg\min_{\boldsymbol{\rho}} \|\mathbf{F}\boldsymbol{\rho} - \mathbf{d}\|_2^2 + R(\boldsymbol{\rho})$$

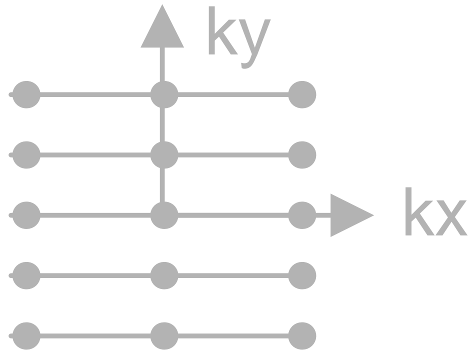
Solutions often derived by solving one or more matrix inversions, e.g.,

$$\hat{\boldsymbol{\rho}} = (\mathbf{F}^H\mathbf{F} + \mathbf{H})^{-1} \mathbf{F}^H \mathbf{d}$$

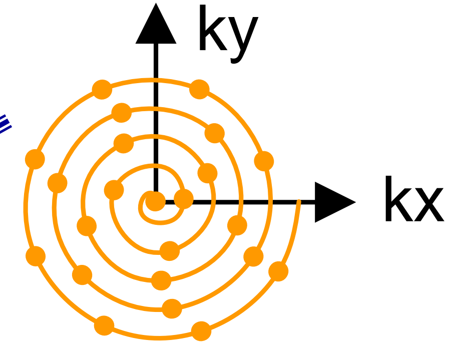
© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Reconstructing MR Images

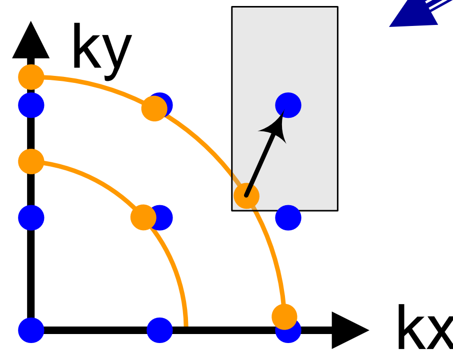
Cartesian Scan Data



Spiral Scan Data



Gridding[1]



FFT

LS

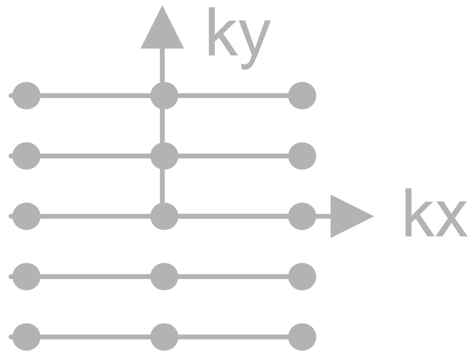
Spiral scan data + Gridding + FFT:
Fast scan, fast reconstruction, better images

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

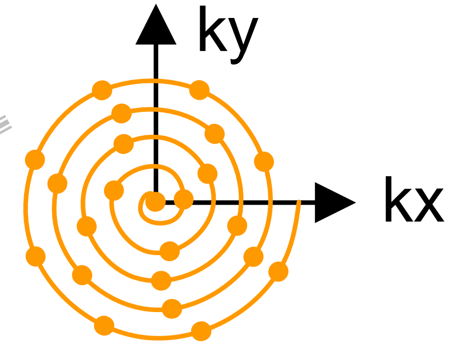
© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Reconstructing MR Images

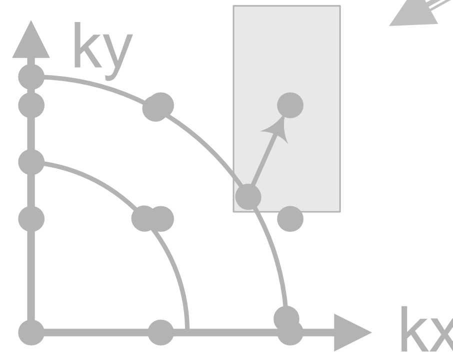
Cartesian Scan Data



Spiral Scan Data



Gridding



FFT

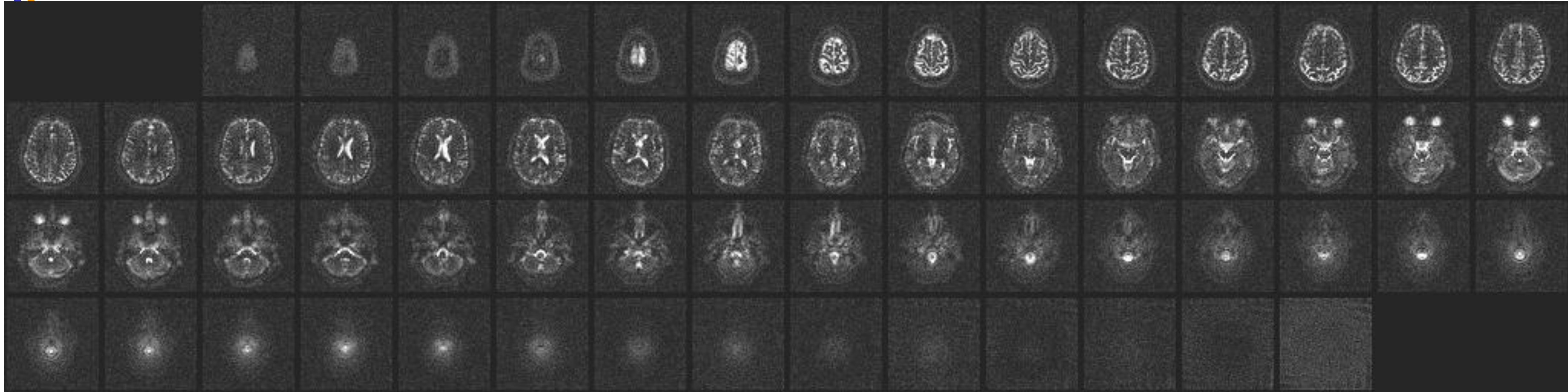
Least-Squares (LS)

Spiral scan data + LS

Superior images at expense of significantly more computation

© David Kirk/NVIDIA and Wen-mei W. Hwu

An Exciting Revolution - Sodium Map of



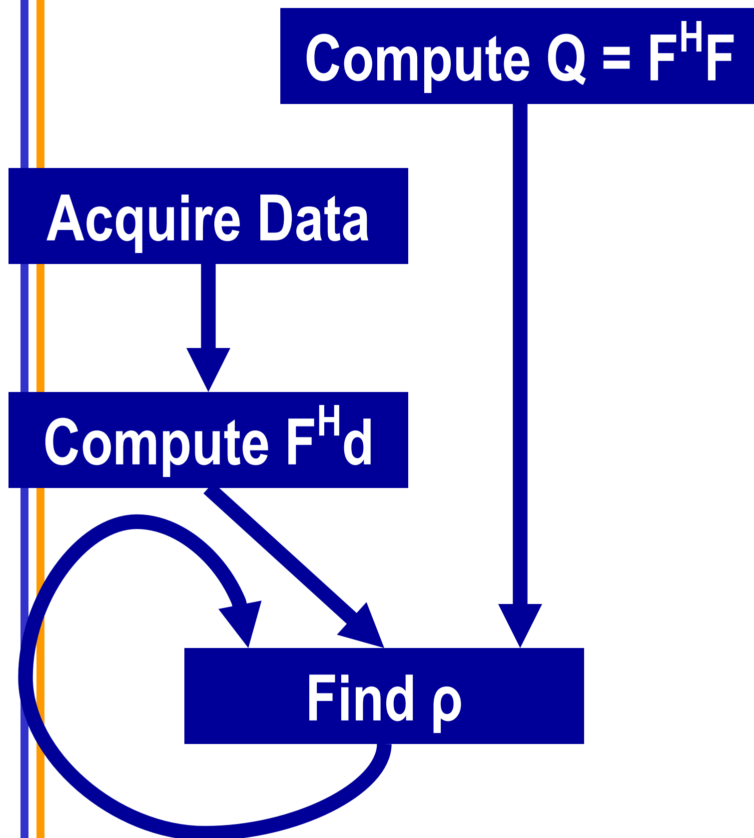
- Images of sodium in the brain
  - Very large number of samples for increased SNR
  - Requires high-quality reconstruction

- Enables study of brain-cell viability before anatomic changes occur in stroke and cancer treatment – within days!

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

Least-Squares Reconstruction

$$F^H F\rho = F^H d$$



- Q depends only on scanner configuration
- $F^H d$ depends on scan data
- ρ found using linear solver
- Accelerate Q and $F^H d$ on G80
  - Q: 1-2 days on CPU
  - $F^H d$: 6-7 hours on CPU
  - ρ: 1.5 minutes on CPU

Algorithms to Accelerate

```
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;
  }
}
```

- Scan data
  - M = # scan points
  - kx, ky, kz = 3D scan data
- Pixel data
  - N = # pixels
  - x, y, z = input 3D pixel data
  - rFhD, iFhD= output pixel data
- Complexity is O(MN)
- Inner loop
  - 13 FP MUL or ADD ops
  - 2 FP trig ops
  - 12 loads, 2 stores

©David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

From C to CUDA: Step 1
What unit of work is assigned to each thread?

```
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;
  }
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

One Possibility

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

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

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

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

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

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

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

One Possibility - Improved

```
__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;
  float rMu_reg, iMu_reg;

  rMu_reg = rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
  iMu_reg = 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_reg*cArg - iMu_reg*sArg;
    iFhD[n] +=  iMu_reg*cArg + rMu_reg*sArg;
  }
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Back to the Drawing Board – Maybe map the n loop to threads?

```
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;
  }
}
```

```
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) $F^{H}d$ computation

```
for (m = 0; m < M; m++) {
  for (n = 0; n < N; n++) {

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

A Second Option for the cmpFHd Kernel

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

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

 for (m = 0; m < M; m++) {
    float rMu_reg =  rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
   float iMu_reg = iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[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_reg*cArg - iMu_reg*sArg;
    iFhD[n] +=  iMu_reg*cArg + rMu_reg*sArg;
  }
}
```

We do have another option.

```
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) $F^H d$ 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

A Separate 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];
}
```

```
__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;
  }
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

```
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
```

Figure 7.9 Loop interchange of the $F^H D$ computation

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

A Third Option of 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;
  }
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Using Registers to Reduce Global Memory Traffic

```
__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;
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Chunking k-space Data to Fit into Constant Memory

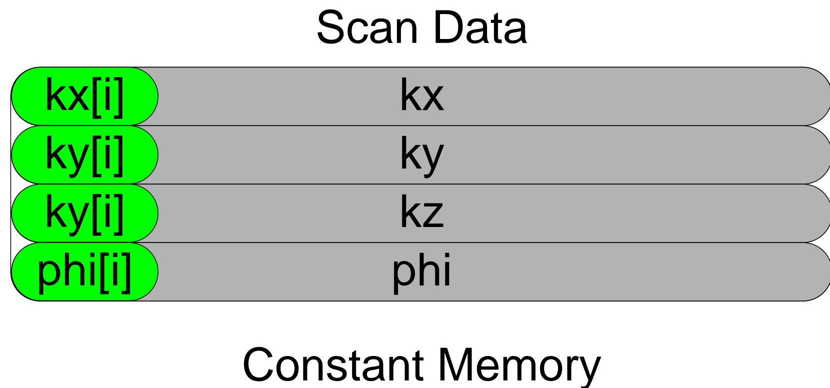
```
__constant__ float  kx_c[CHUNK_SIZE],
                    ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];
…
__ void main() {

  int i;
  for (i = 0; i < M/CHUNK_SIZE; i++);
    cudaMemcpy(kx_c,&kx[i*CHUNK_SIZE],4*CHUNK_SIZE,
               cudaMemCpyHostToDevice);
    cudaMemcpy(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
               cudaMemCpyHostToDevice);
    cudaMemcpy(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, int M);
  }
  /* Need to call kernel one more time if M is not */
  /* perfect multiple of CHUNK SIZE */
}
```

Revised Kernel for Constant Memory

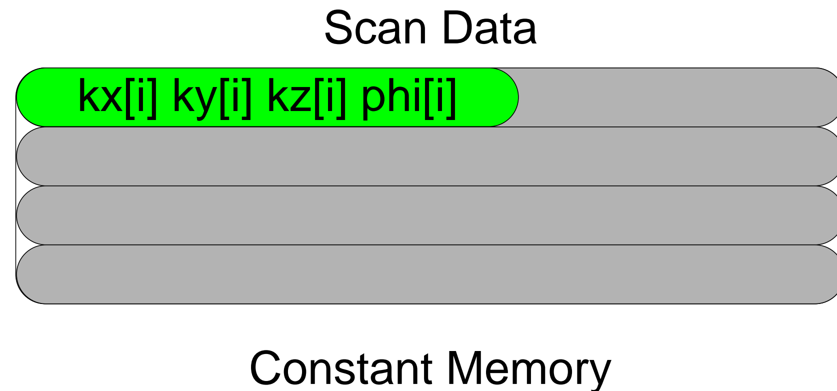
```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
      x, y, z, rMu, iMu, int M) {

  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[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;
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

K-space Data Layout in Constant Memory



(a) k-space data stored in separate arrays.



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

K-Space Layout Host Code

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

__constant__ struct kdata k_c[CHUNK_SIZE];
…

__ void main() {

 int i;

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

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

  }
```

K-Space Layout Kernel Code

```
__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;
}
```

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Sidebar: Estimating Off-Chip Loads with Const Cache

- How can we approximate the number of off-chip loads when using the constant caches?
- Given: 128 threads per block, 4 blocks per SM, 256 scan points per grid
- **Assume no evictions due to cache conflicts**
- 7 accesses to global memory per thread (x, y, z, rQ x 2, iQ x 2)
  - 4 blocks/SM * 128 threads/block * 7 accesses/thread = 3,584 global mem accesses
- 4 accesses to constant memory per scan point (kx, ky, kz, phi)
  - 256 scan points * 4 loads/point = 1,024 constant mem accesses
- Total off-chip memory accesses = 3,584 + 1,024 = 4,608
- Total FP arithmetic ops = 4 blocks/SM * 128 threads/block * 256 iters/thread * 10 ops/iter = 1,310,720
- FP arithmetic to off-chip loads: 284 to 1

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Sidebar: Effects of Approximations

- Avoid temptation to measure only absolute error ($I_0 - I$)
  - Can be deceptively large or small
- Metrics
  - PSNR: Peak signal-to-noise ratio
  - SNR: Signal-to-noise ratio
- Avoid temptation to consider only the error in the computed value
  - Some apps are resistant to approximations; others are very sensitive

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

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

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

(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

Validation of floating-point precision and accuracy of the different FHd implementations.

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Sidebar: Optimizing the CPU Implementation

- Optimizing the CPU implementation of your application is very important
  - Often, the transformations that increase performance on CPU also increase performance on GPU (and vice-versa)
  - The research community won't take your results seriously if your baseline is crippled
- Useful optimizations
  - Data tiling
  - SIMD vectorization (SSE)
  - Fast math libraries (AMD, Intel)
  - Classical optimizations (loop unrolling, etc)
- Intel compiler (icc, icpc)

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Summary of Results

	Q		$F^H d$			
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)	13.6	98.2	2.4	92.2	1.60	4.00
LS (GPU, CMem, SFU, Exp)	7.5	178.9	1.5	144.5	1.69	3.19

8X

© David Kirk/NVIDIA and Wen-mei W. Hwu
Urbana, Illinois, August 10-14, 2009

Summary of Results

	Q		$F^H d$			
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 (G80, Naïve)	260.2	5.1	41.0	5.4	1.65	42.65
LS (G80, CMem)	72.0	18.6	9.8	22.8	1.57	11.37
LS (G80, CMem, SFU)	13.6	98.2	2.4	92.2	1.60	4.00
LS (G80, CMem, SFU, Exp/layout)	7.5	178.9	1.5	144.5	1.69	3.19

357X

228X

108X

© David Kirk/NVIDIA and [obscured] . Hwu
Urbana, Illinois, August 10-14, 2009