CS 677: Parallel Programming for Many-core Processors
Lecture 6

Instructor: Philippos Mordohai

Webpage: mordohai.github.io

E-mail: Philippos.Mordohai@stevens.edu

Homework Assignment 3

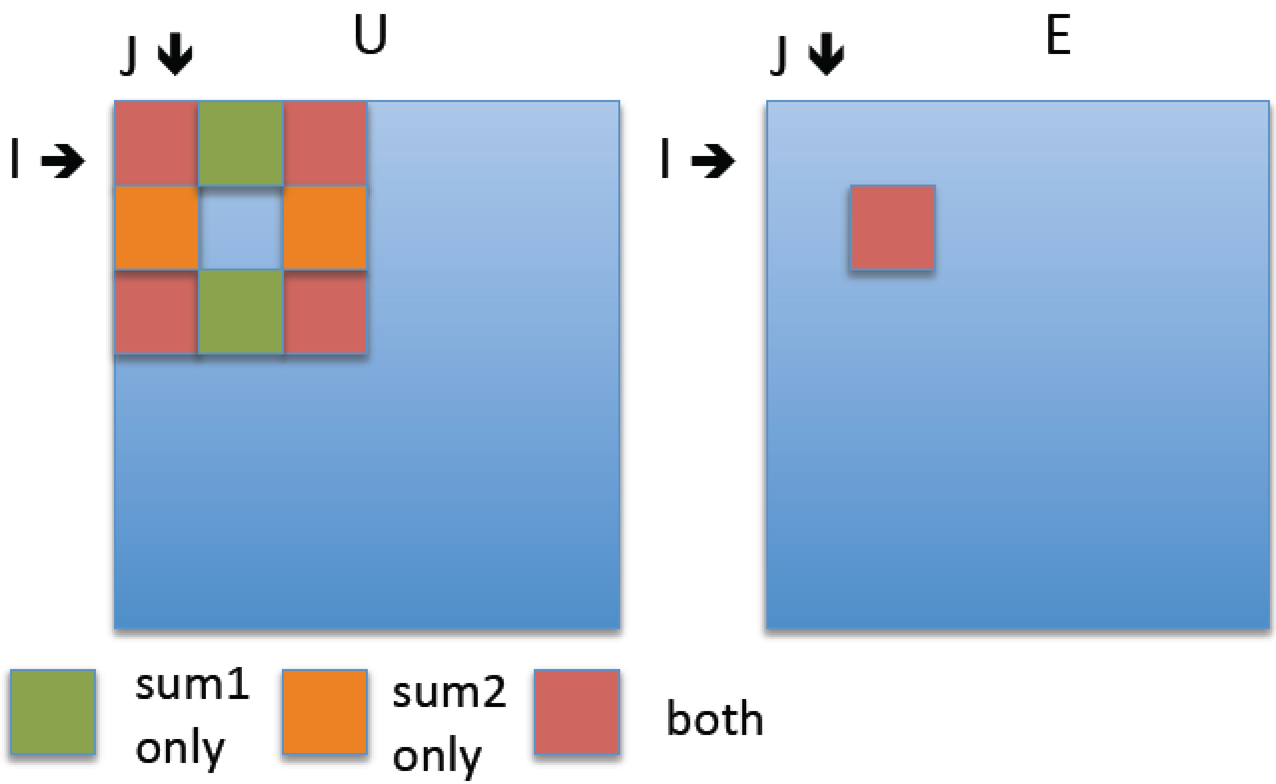
- Apply Sobel filter on (grayscale) images

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Homework Assignment 4: CPU Version

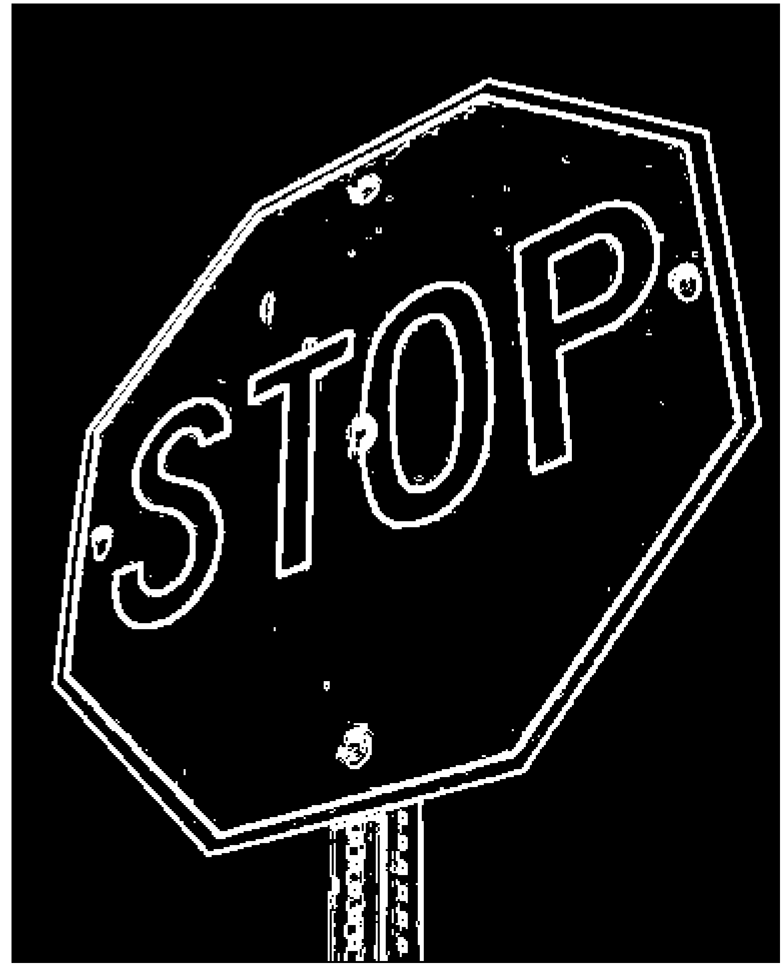
```
for (i = 1; i < ImageNRows - 1; i++)
  for (j = 1; j < ImageNCols -1; j++)
  {
      sum1 = u[i-1][j+1] - u[i-1][j-1]
           + 2 * u[i][j+1] - 2 * u[i][j-1]
           + u[i+1][j+1] - u[i+1][j-1];
      sum2 = u[i-1][j-1] + 2 * u[i-1][j]
           + u[i-1][j+1] – u[i+1][j-1]
           - 2 * u[i+1][j] - u[i+1][j+1];
      magnitude = sum1*sum1 + sum2*sum2;
      if (magnitude > THRESHOLD)
           e[i][j] = 255;
      else
           e[i][j] = 0;
}
```

Homework Assignment 4



- Compute magnitude of filter response $G_x^2 + G_y^2$ and output:
  - 0 if magnitude below threshold
  - 255 if magnitude above threshold
  - 0 pixel is within 1 pixel of image border

Example Output



Open Questions

- Memory bandwidth
- 1D vs. 2D block structure
  - Fetching of pixels at block boundaries
- I prefer solutions without padding, but you can pad for a 10% penalty

- Solutions using global memory only will receive little credit

The PPM Image Format

- PPM is a very simple format
- Each image file consists of a header followed by all the pixel data
- Header

P6
comment 1
comment 2
.
#comment n
rows columns maxvalue
pixels

P3 means ASCII file
P6 means binary (most practical)

See filereading code in homework zip file

Use Gimp or IrfanView to manipulate images and convert between formats

Reading the Header

```
fp = fopen(filename, "rb");
…
int num = fread(chars, sizeof(char), 1000, fp);
if (chars[0] != 'P' || chars[1] != '6')
{
   fprintf(stderr, "ERROR  file '%s' does not
      start with \"P6\"  I am expecting a binary
      PPM file\n", filename);
   return NULL;
}
```

check for “P6” in first line

Reading the Header (cont)

```
unsigned int width, height, maxvalue;
char *ptr = chars+3; // P 6 newline
if (*ptr == '#') // comment line!
{
      ptr = 1 + strstr(ptr, "\n");
}
num = sscanf(ptr, "%d\n%d\n%d",
            &width, &height, &maxvalue);
fprintf(stderr, "read %d things   width %d  height %d
      maxval %d\n", num, width, height, maxvalue);
*xsize = width;
*ysize = height;
*maxval = maxvalue;
```

skip over comments by looking for # in first column

Reading the Data

```
// allocate buffer to read the rest of the file into
int bufsize =  3 * width * height * sizeof(unsigned char);
if ((*maxval) > 255) bufsize *= 2;
unsigned char *buf = (unsigned char *)malloc( bufsize );

…

long numread = fread(buf, sizeof(char), bufsize, fp);

…

int pixels = (*xsize) * (*ysize);
for (int i=0; i<pixels; i++)
       pic[i] = (int) buf[3*i];  // red channel
return pic; // success
```

Application Case Study – Advanced 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

Acknowledgements

Sam S. Stone[§], Haoran Yi[§], Justin P. Haldar[†], Deepthi Nandakumar, Bradley P. Sutton[†],
Zhi-Pei Liang[†], Keith Thulburin[*]

[§]Center for Reliable and High-Performance Computing

[†] Beckman Institute for Advanced Science and Technology

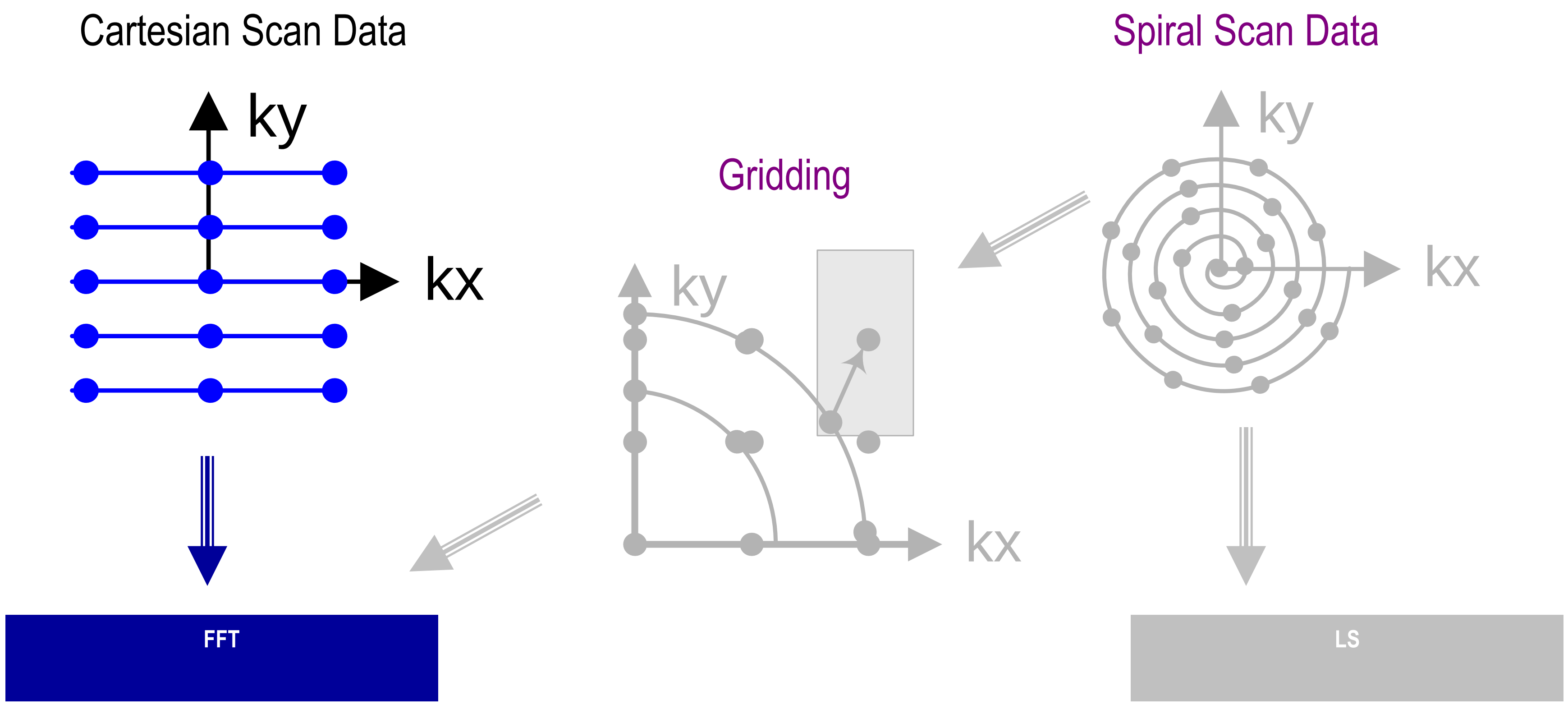
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

* *University of Illinois, Chicago Medical Center*

Overview

- Magnetic resonance imaging
- Non-Cartesian Scanner Trajectory
- Least-squares (LS) reconstruction algorithm
- Optimizing the LS reconstruction on the G80
  – Overcoming bottlenecks
  – Performance tuning
- Summary

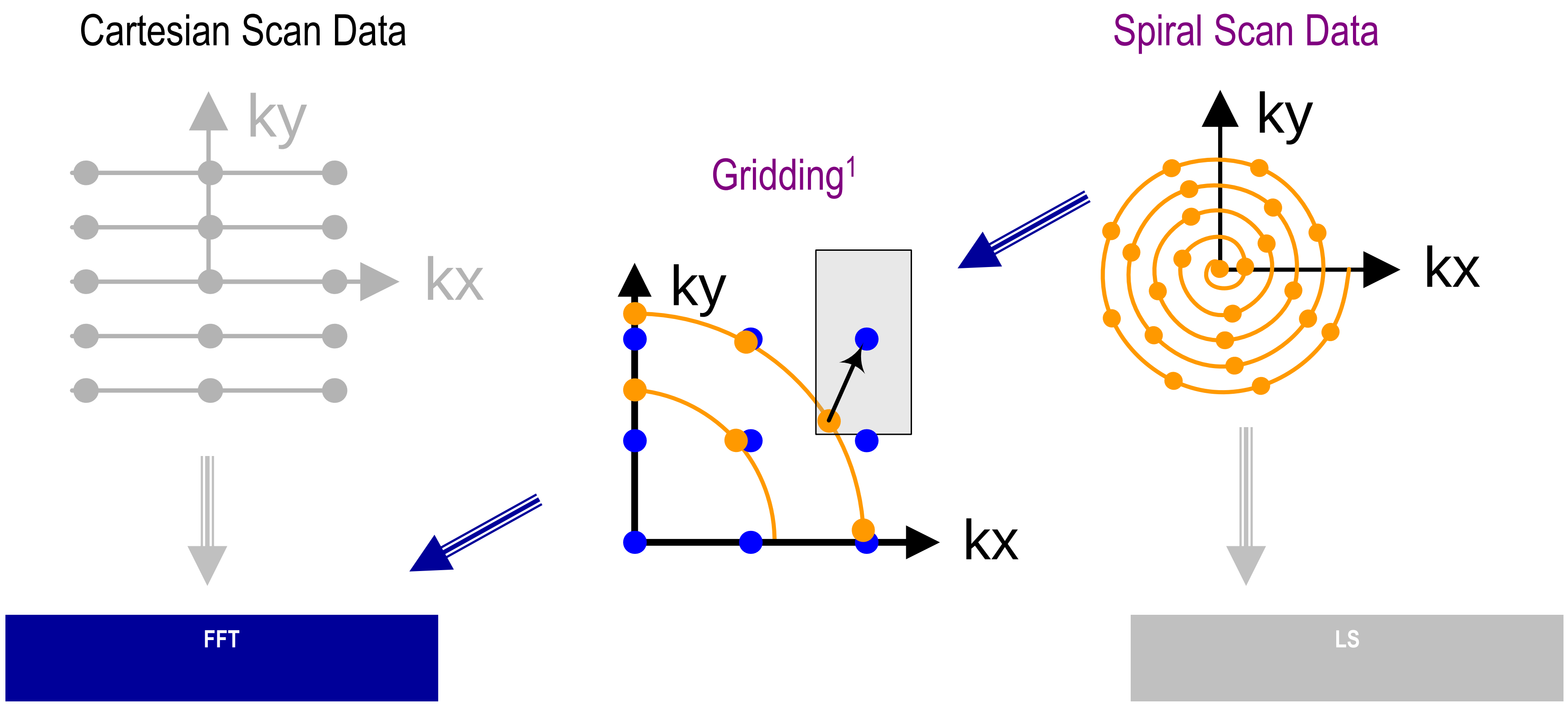
Reconstructing MR Images



Cartesian scan data + FFT:

Slow scan, fast reconstruction, images may be poor

Reconstructing MR Images

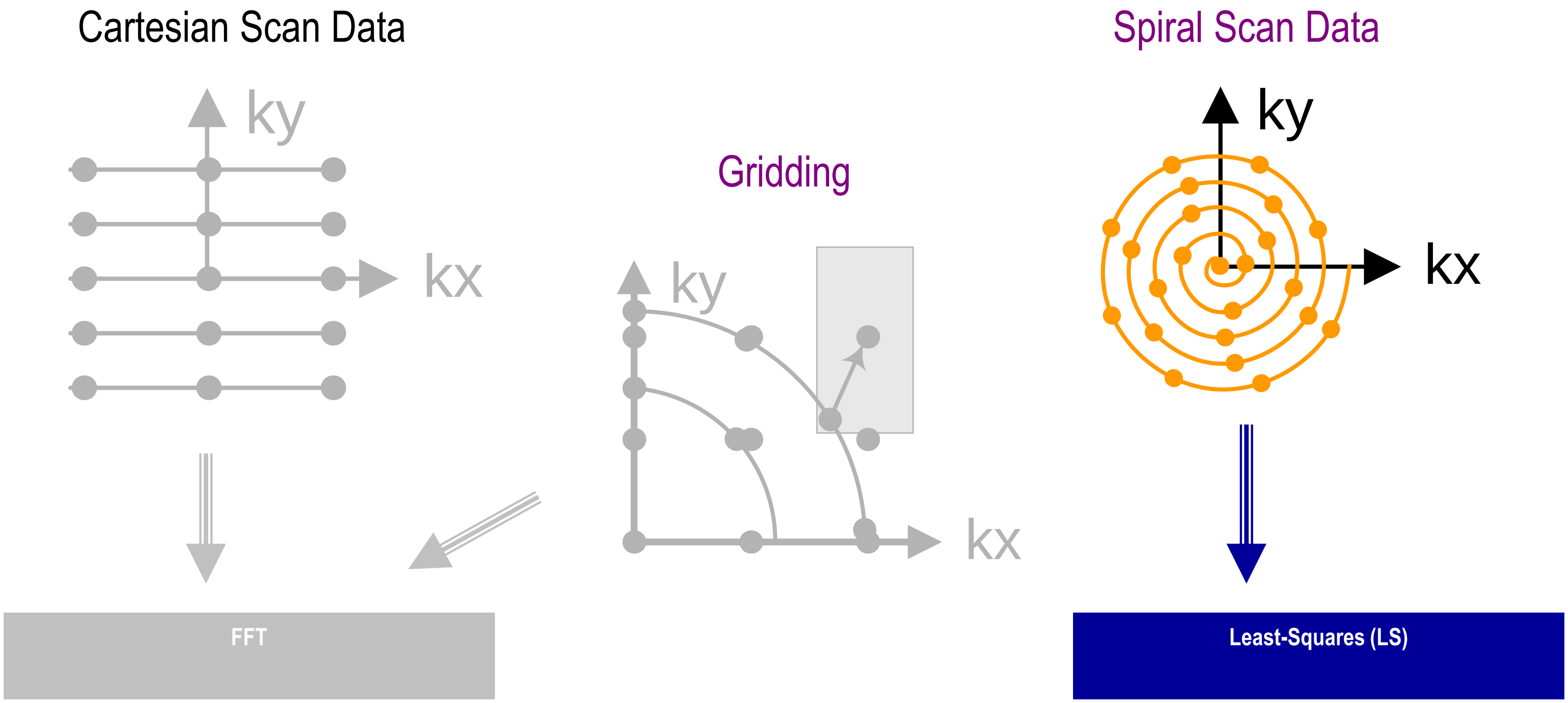


Spiral scan data + Gridding + FFT:

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

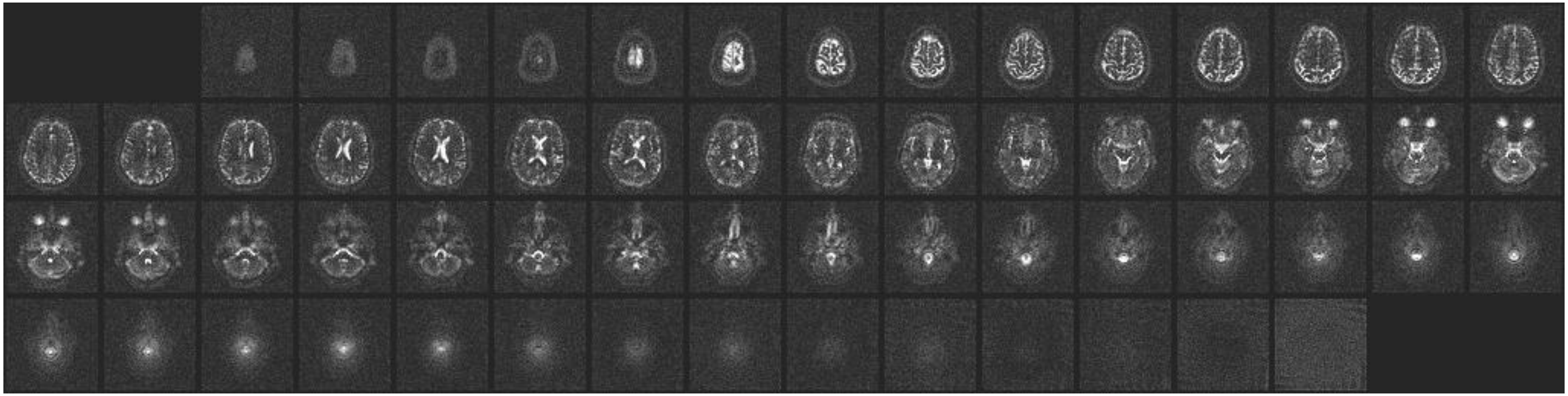
Reconstructing MR Images



Spiral scan data + LS

Superior images at expense of significantly more computation

An Exciting Revolution - Sodium Map of the Brain



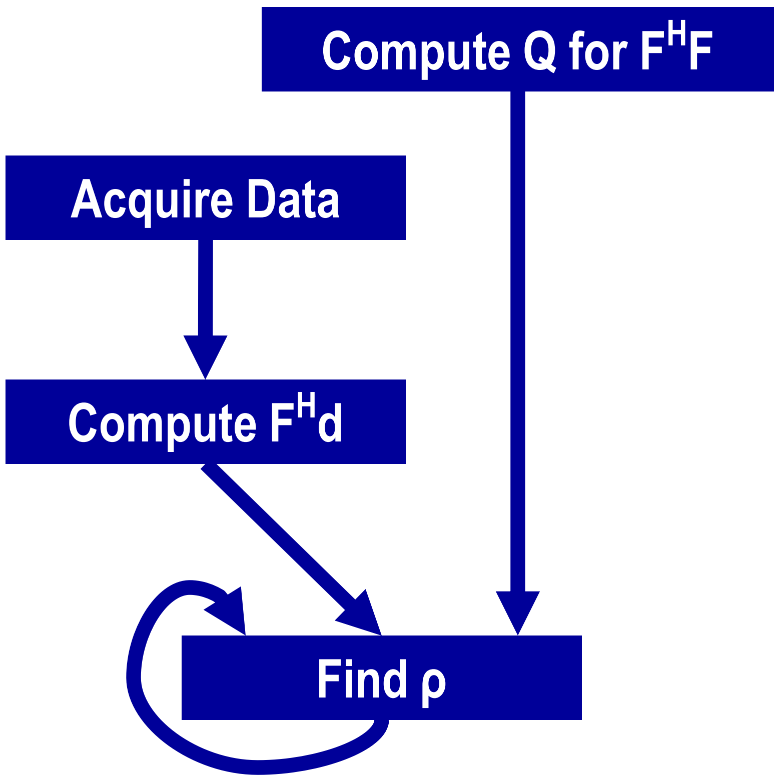
- Images of sodium in the brain
  - Very large number of samples for increased SNR
  - Requires high-quality reconstruction

- 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

$$(F^H F + W^H W)\rho = F^H d$$



- $F^H F$ depends only on scanner configuration
- $W^H W$ incorporates prior information, such as anatomical constraints
- $F^H d$ depends on scan data
- ρ vector containing voxel values of reconstructed image - found using linear solver
  - 99.5% of the reconstruction time for a single image is devoted to computing $F^H d$
  - computing Q is even more expensive, but depends only on the scanner configuration and can be amortized

Least-Squares Reconstruction

- The solution is:

$$\rho = (F^H F + W^H W)^{-1} F^H d$$

- but for a relatively low-res reconstruction of 128^3 voxels, the inverted matrix contains well over four trillion complex-valued elements
- Use conjugate gradient to solve

Least-Squares Reconstruction

$$(F^H F + W^H W)\rho = F^H d$$

- $W^H W$ is sparse
- $F^H F$ has convolutional structure
  - each descending diagonal from left to right is constant
- Efficient FFT-based matrix multiplication is possible
  - Out of scope for CS 677

Least-Squares Reconstruction

- What has to be computed is the Q matrix which depends only on the scan trajectory, but not the scan data

$$Q(x_n) = \sum_{m=1}^{M} |\varphi(k_m)|^2 \, e^{(i2\pi k_m \cdot x_n)}$$

- where:
  - k_m is the location of the m^{th} sample
  - x_n is the n^{th} voxel
  - φ() is the Fourier transform of the voxel basis function

Least-Squares Reconstruction

- What also needs to be computed is the vector $F^H d$ which depends on the data

$$[F^H d]_n = \sum_{m=1}^{M} \varphi^*(k_m) d(k_m) e^{(i 2\pi k_m \cdot x_n)}$$

- These two equations look similar but the computation of Q requires oversampling by a factor of 2 in each dimension
  – Q is O(8MN) and $F^H d$ is O(MN)

Least-Squares Reconstruction - Complexity

- Q: 1-2 days on CPU
- $F^H d$: 6-7 hours on CPU
- ρ: 1.5 minutes on CPU

- Therefore, accelerate Q and $F^H d$ computations

Q v.s. F^HD

```
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

25

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
- Voxel data
  - N = # voxels
  - x, y, z = input 3D voxel data
  - rFhD, iFhD= output voxel data
- Complexity is O(MN)
- Inner loop
  - 14 FP MUL or ADD ops
  - 2 FP trig ops (12-13 FL OPs)
  - 12 loads, 2 stores

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;
  }
}
```

1. Each thread executes an iteration of the outer loop
   => Problem: Each thread is trying to accumulate a partial sum to rFhD and iFhD (requires a reduction)
2. Each thread executes an iteration of the inner loop.
   - Avoids the reduction problem
   - But now each thread is doing very little work
   - We need one grid for each outer loop iteration.
   - Performance limited by overheads for launching M grids and writing 2N values to global memory for each grid

One Possibility (Wrong)

```
__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;
  }
}
```

One Possibility (Wrong) - 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;
  }
}
```

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

```
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];
}
```

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 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;
  }
}
```

Problem: Each thread is trying to accumulate a partial sum to rFhD and iFhD

<pre>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</pre>
	<pre>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</pre>

Loop interchange of the F^HD computation

A Third Option for the FHd 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];
   float iMu_reg = iMu[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;
  }
}
```

From C to CUDA: Step 2 Getting around Memory Bandwidth Limitations

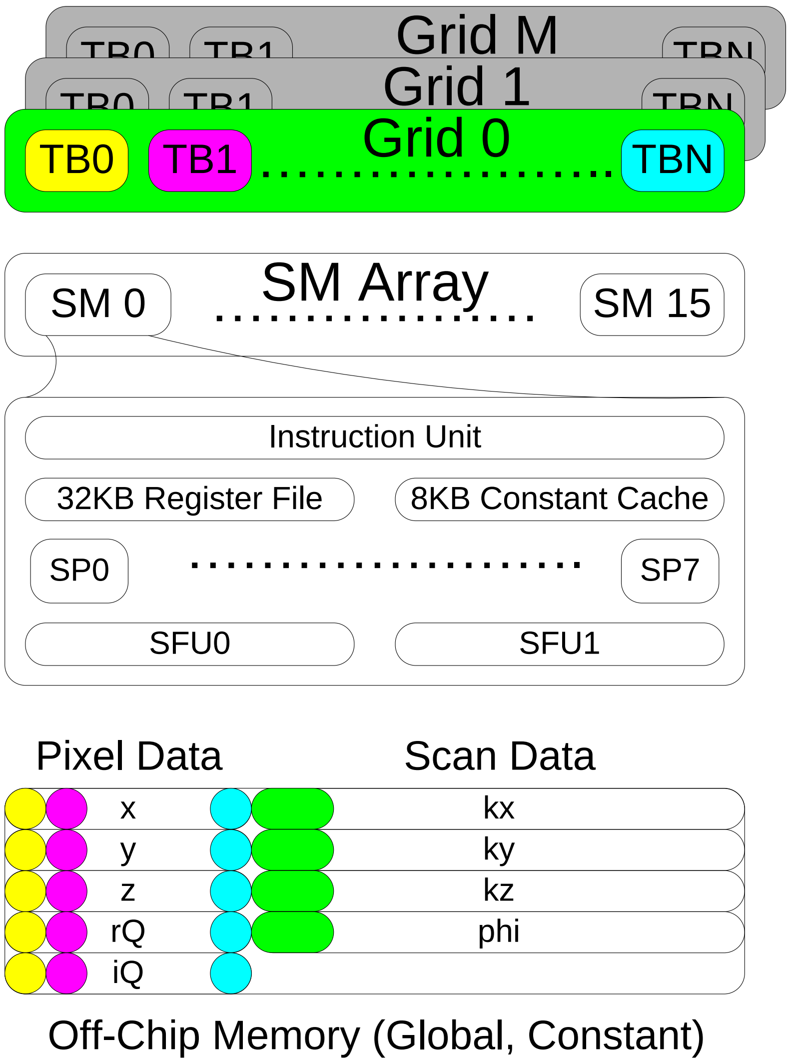
- Using registers
- Using constant memory

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;
}
```

Compute-to-memory access ratio 14:7 (inside the loop)
Was 14:14 before (approx.)

Tiling of Scan Data



LS reconstruction uses multiple grids

- Each grid operates on all scan data
- Each grid operates on a distinct subset of voxels
- Each thread in the same grid operates on a distinct voxel

Thread n operates on voxel n:

```
for (m = 0; m < M/32; m++) {
   exQ = 2*PI*(kx[m]*x[n] +
               ky[m]*y[n] +
               kz[m]*z[n])
   rQ[n] += phi[m]*cos(exQ)
   iQ[n] += phi[m]*sin(exQ)
}
```

Using Constant Memory

- All threads access scan data (kx, ky, kz) in the same order
- Threads don't modify scan data

- Put scan data in constant memory
  - Limited to 64kB (larger than shared memory)
  - But cached, for every 32 accesses to constant memory, at least 31 will be cached (96% reduction in time, no bank conflicts – broadcast mode to all threads in warp)

Chunking k-space Data to Fit into Constant Memory

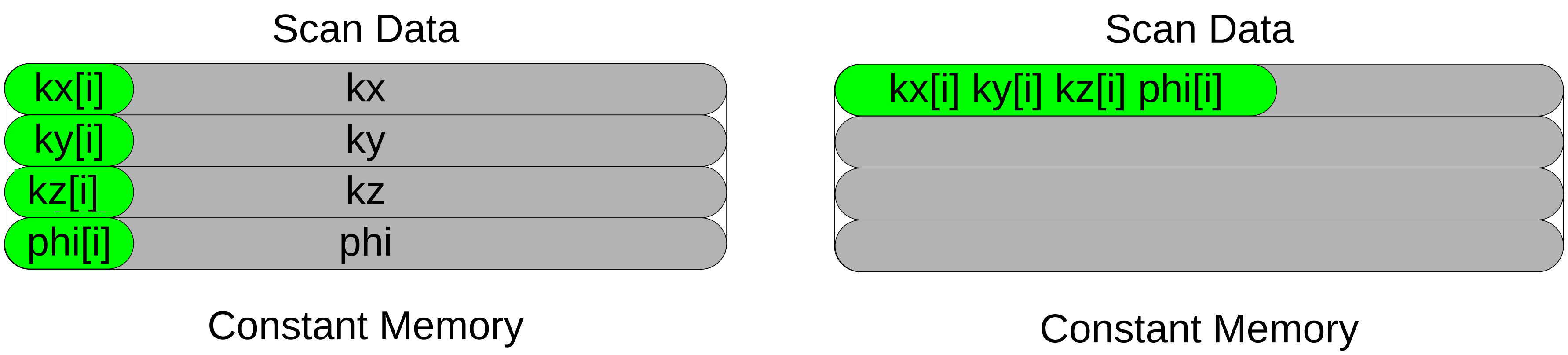
```
__constant__ float  kx_c[CHUNK_SIZE],
                    ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];
…
void main() {

  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(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,
               int CHUNK_SIZE);
  }
  /* 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_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;
}
```

kx_c, ky_c and kz_c are no longer arguments but global variables

Compute-to-memory access ratio 14:4 (inside the loop)
Can be 14:2 if compiler stores rMu[m] and iMu[m] in temporary registers



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

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

Effect of k-space data layout on constant cache efficiency.

- The previous implementations leads to bad (slow) performance
- Each constant cache entry is designed to store multiple consecutive words
- There are very few such entries – insufficient for all active warps in an SM
- Solution: use array of struct (contrary to last week’s advice)

```
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>>>
             ();

  }
```

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 (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;
}
```

Adjusting the k-space data memory layout in the $F^{H}d$ kernel

From C to CUDA: Step 3
Where are the potential bottlenecks?

Bottlenecks

- Memory Bandwidth
  - See previous slides
- Trig operations
- Overhead (branches, address calculations)
  - These are important due to short inner loop

Trigonometric Operations

- Use SFUs (Super Function Units)
  - __sin and __cos are implemented as hardware instructions
    - Require 4 cycles (vs. 12 and 13 FLOP for software versions)
    - Reduced accuracy
- Performance: from 22.8 GFLOPS to 92.2 GFLOPS

Address Calculations

- Last bottleneck: Overhead of branches and address calculations
- Solution: Loop unrolling and experimental tuning
  - Loop unrolling factors (1,2,4,8,16)
  - Also experimentally tuned the number of threads per block and the number of scan points per grid (see following slides)
- Performance:179 GFLOPS (Q), 145 GFLOPS ($F^H d$)

Experimental Methodology

- Reconstruct a 3D image of a human brain[1]
  - 3.2 M scan data points acquired via 3D spiral scan
  - 256K voxels
- Compare performance of several reconstructions
  - Gridding + FFT reconstruction[1] on CPU (Intel Core 2 Extreme Quadro)
  - LS reconstruction on CPU (double-precision, single-precision)
  - LS reconstruction on GPU (NVIDIA GeForce 8800 GTX)
- Metrics
  - Reconstruction time: compute $F^H d$ and run linear solver
  - Run time: compute Q or $F^H d$

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

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 applications are resistant to approximations; others are very sensitive

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

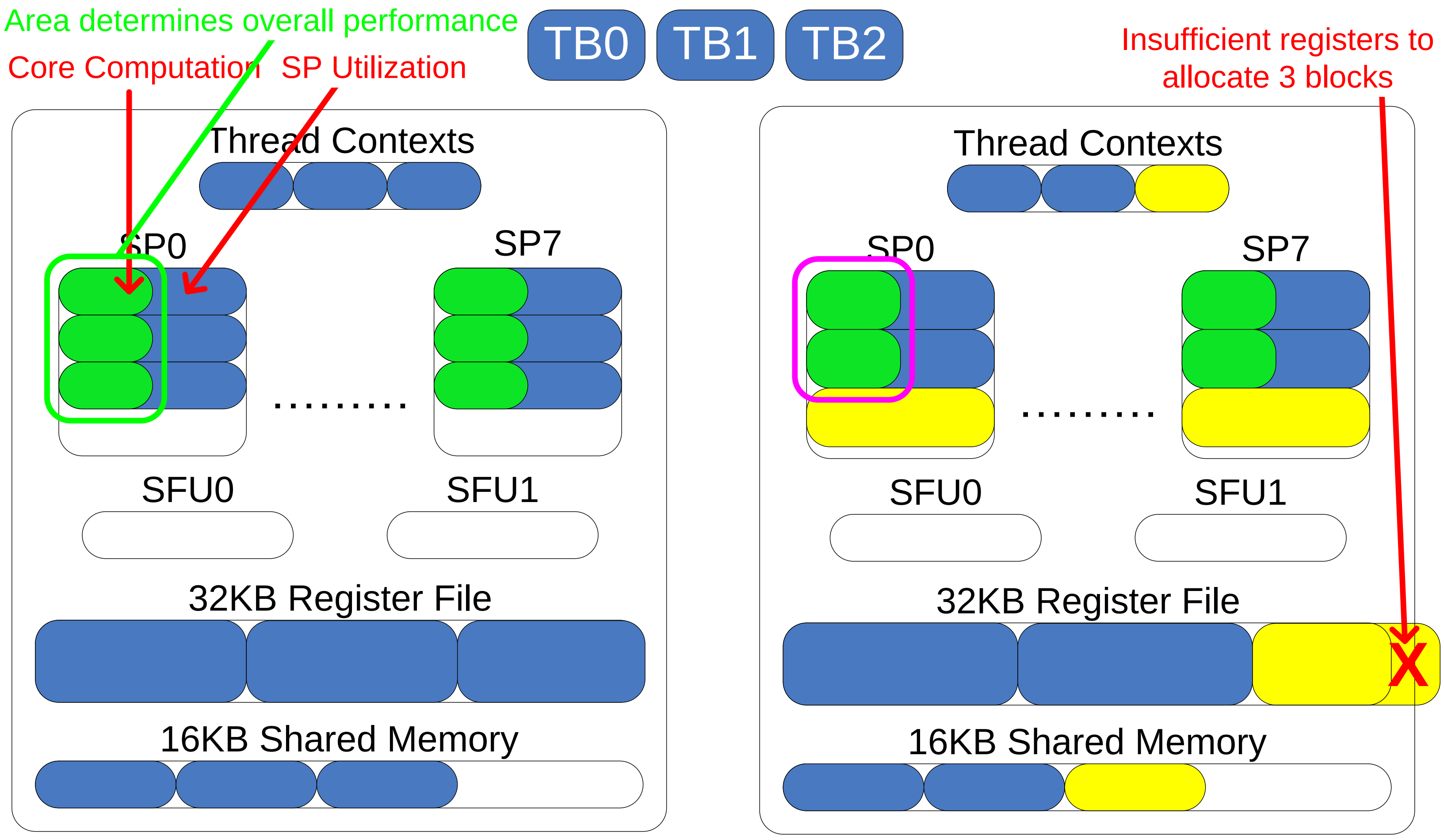
$$PSNR = 20\log_{10}\left(\frac{\max(I_0(i,j))}{\sqrt{MSE}}\right) \qquad SNR = 20\log_{10}\left(\frac{\sqrt{A_s}}{\sqrt{MSE}}\right)$$

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

Experimental Tuning: Tradeoffs

- In the Q kernel, three parameters are natural candidates for experimental tuning
  – Loop unrolling factor (1, 2, 4, 8, 16)
  – Number of threads per block (32, 64, 128, 256, 512)
  – Number of scan points per grid (32, 64, 128, 256, 512, 1024, 2048)
- Cannot optimize these parameters independently
  – Resource sharing among threads (register file, shared memory)
  – Optimizations that increase a thread's performance often increase the thread's resource consumption, reducing the total number of threads that execute in parallel
- Optimization space is not linear
  – Threads are assigned to SMs in large thread blocks
  – Causes discontinuity and non-linearity in the optimization space

Experimental Tuning: Example

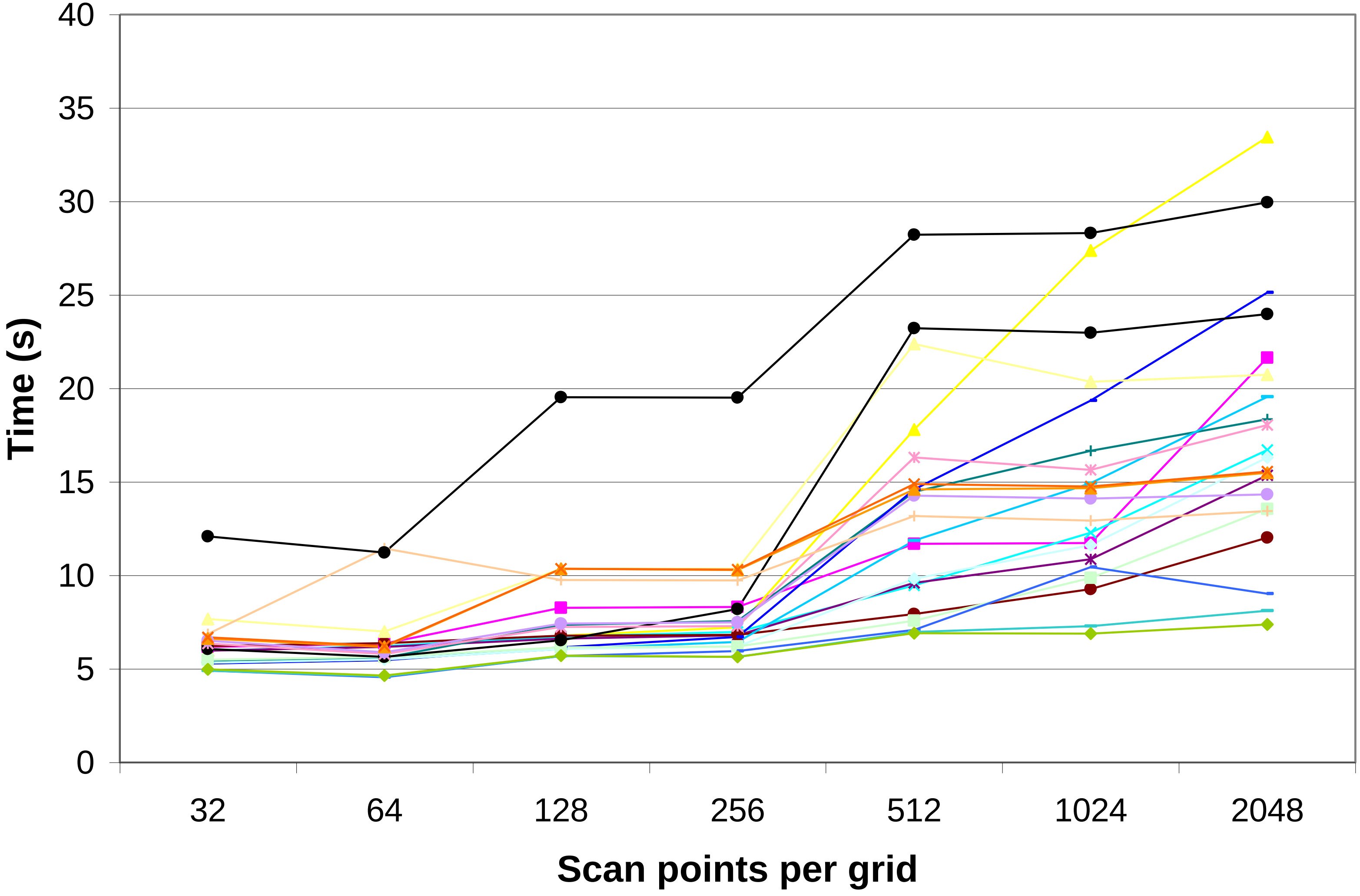


(a) Pre-“optimization”

(b) Post-“optimization”

Increase in per-thread performance, but fewer threads:
Lower overall performance

Experimental Tuning: Scan Points Per Grid

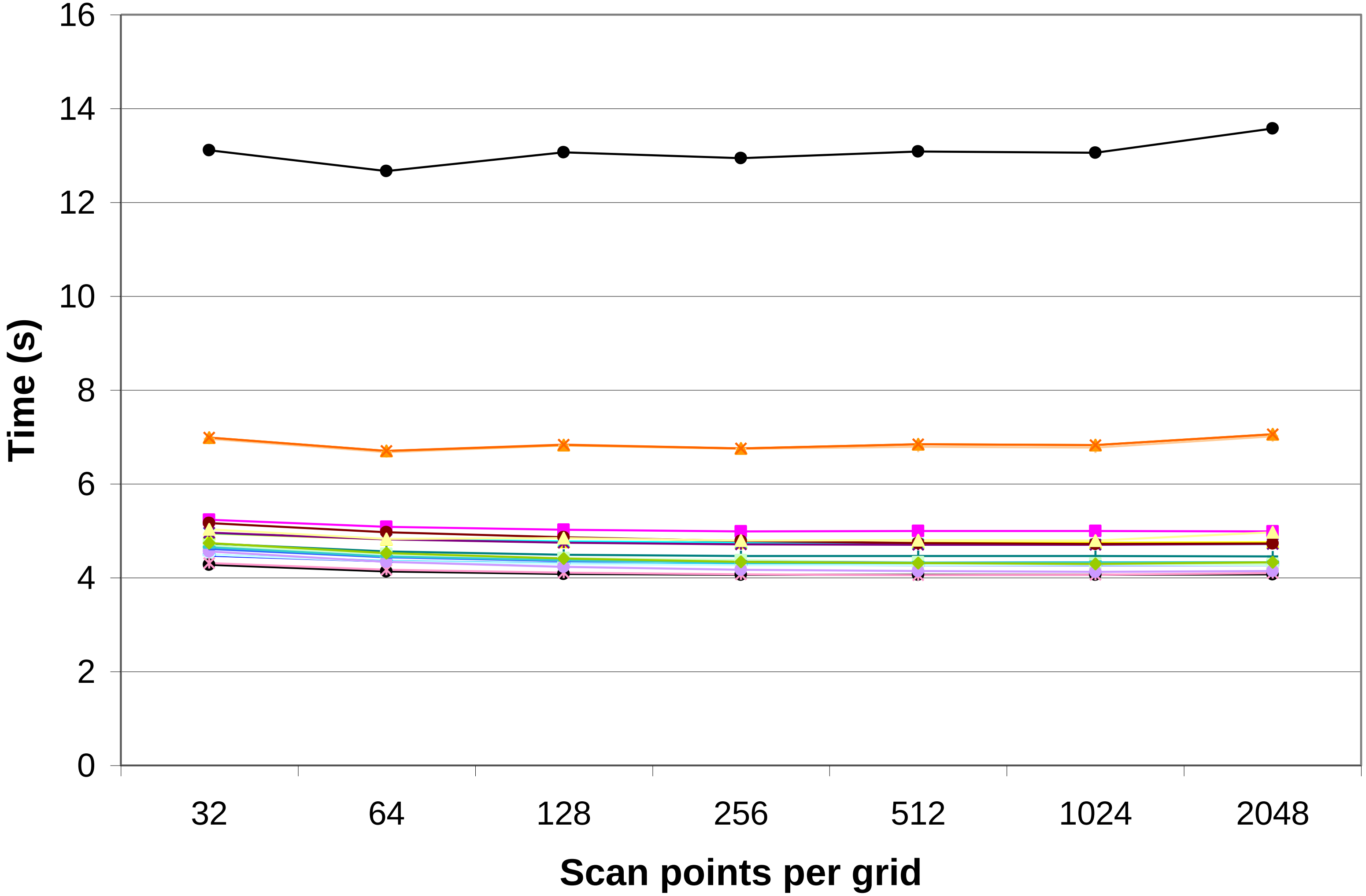


Experimental Tuning: Scan Points Per Grid

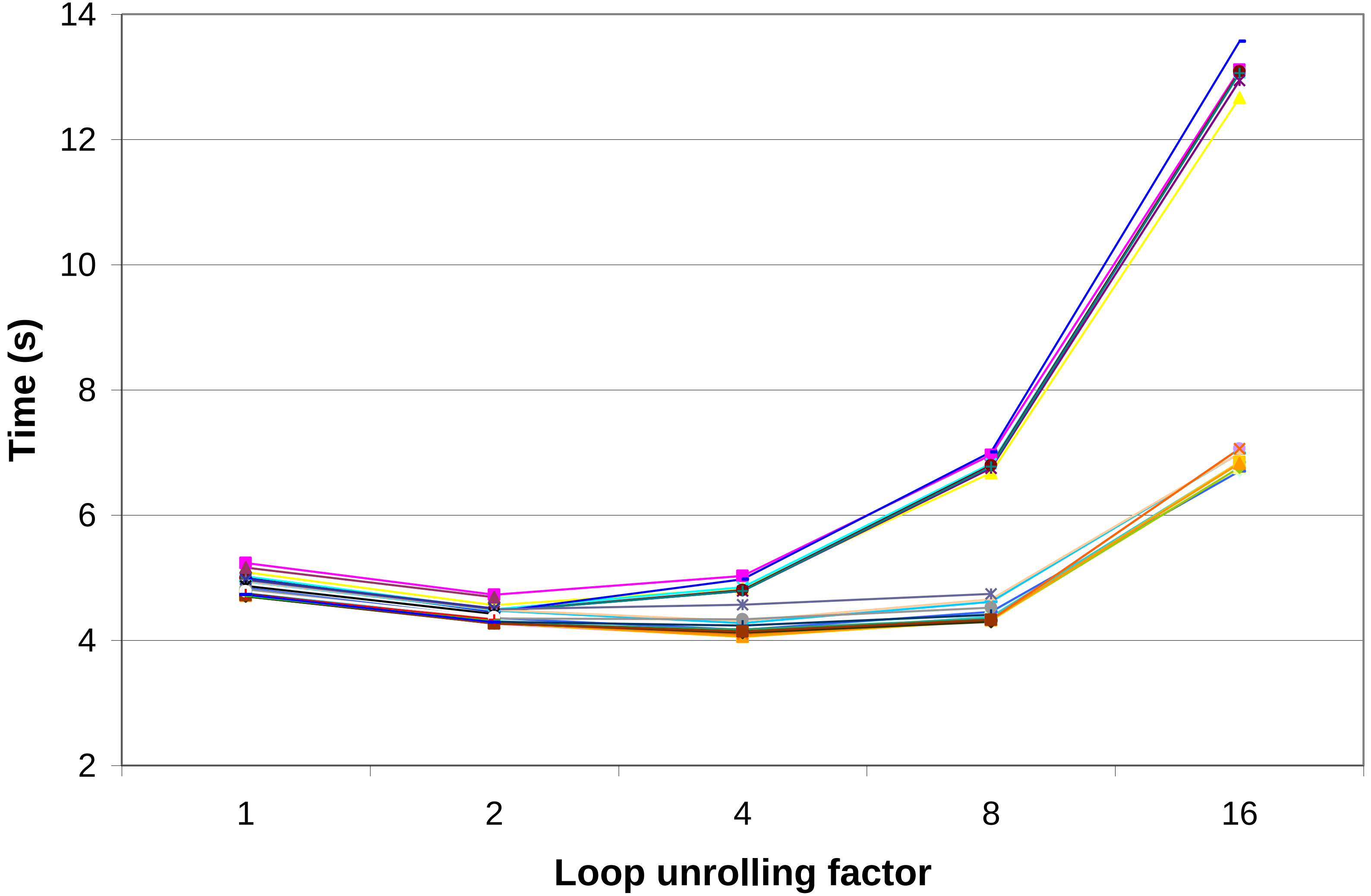
- Each line in previous plot represents a combination of loop unrolling factor and threads per block
- The y-axis represents runtime, so lower is better

- Runtime tends to increase as the number of scan points per grid increases
- That's counter-intuitive. Why would performance get worse as the amount of data processed by each kernel increased?
  - ➢ Conflicts in the constant cache (across different blocks)

Experimental Tuning: Scan Points Per Grid (Improved Data Layout)



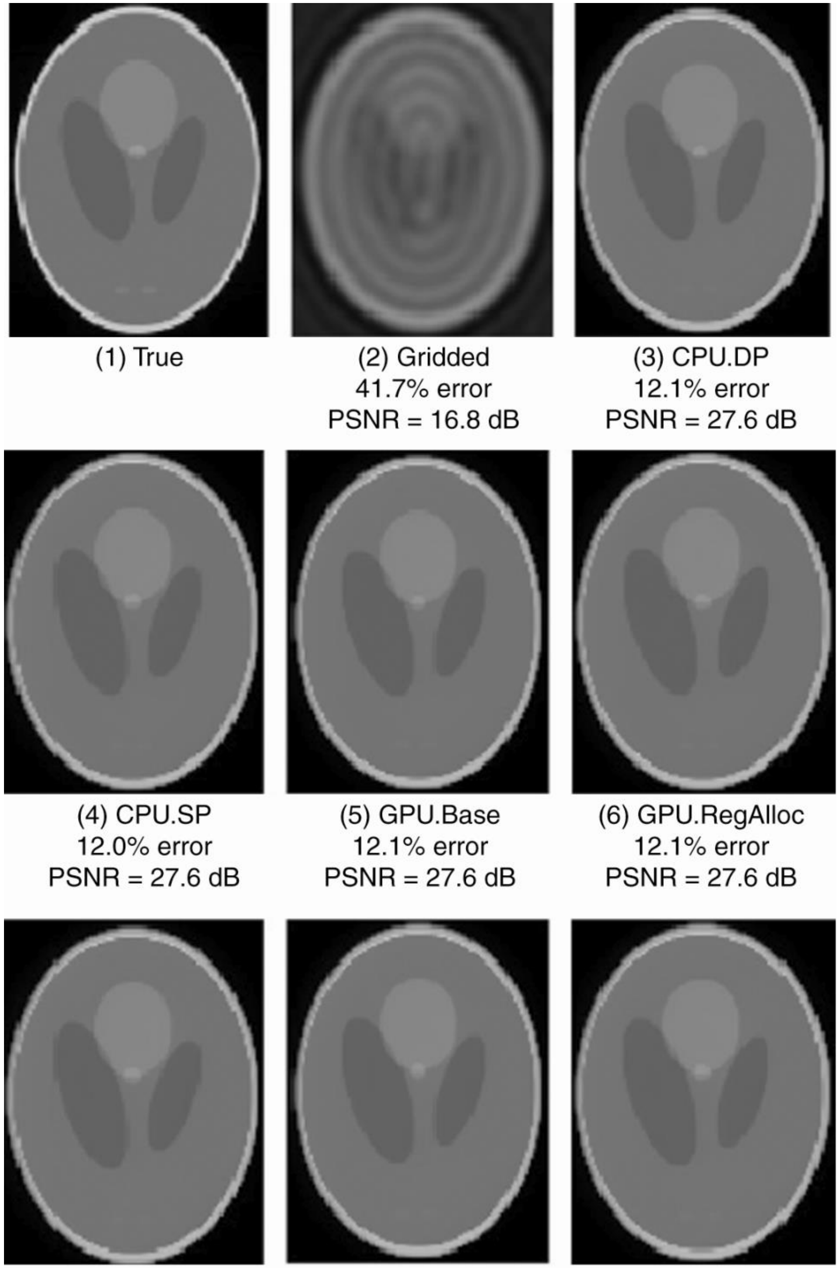
Experimental Tuning: Loop Unrolling Factor



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)

Quantitative Evaluation



(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

Summary of Results

	Q		F^Hd			
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

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
	357X		228X			108X

Timers

- Any timer can be used
  - Check resolution
- Important: many CUDA API functions are asynchronous
  - They return control back to the calling CPU thread prior to completing their work
  - All kernel launches are asynchronous
  - So are all memory copy functions with the `Async` suffix on the name

Synchronization

- Synchronize the CPU thread with the GPU by calling `cudaThreadSynchronize()` immediately before starting and stopping the CPU timer
- `cudaThreadSynchronize()` blocks the calling CPU thread until all CUDA calls previously issued by the thread are completed

Synchronization

- `cudaEventSynchronize()` blocks until a given event in a particular stream has been recorded by the GPU
  - Safe only in the default (0) stream
  - Fine for our purposes

CUDA Timer

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

kernel<<<grid,threads>>> ( d_odata, d_idata,
  size_x, size_y, NUM_REPS);

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop ); // after cudaEventRecord
cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Output

- `time` is in milliseconds
- Its resolution of approximately half a microsecond
- The timings are measured on the GPU clock
  - Operating system-independent