# Case Study: Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction
2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

MCS 572 Lecture 37
Introduction to Supercomputing
Jan Verschelde, 20 November 2024

# Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction

2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

# magnetic resonance imaging

Magnetic Resonance Imaging (MRI) is a safe and noninvasive probe of the structure and function of tissues in the body.

MRI consists of two phases:

1. Acquisition or scan: the scanner samples data in the spatial-frequency domain along a predefined trajectory.
2. Reconstruction of the samples into an image.

Limitations: noise, imaging artifacts, long acquisition times.

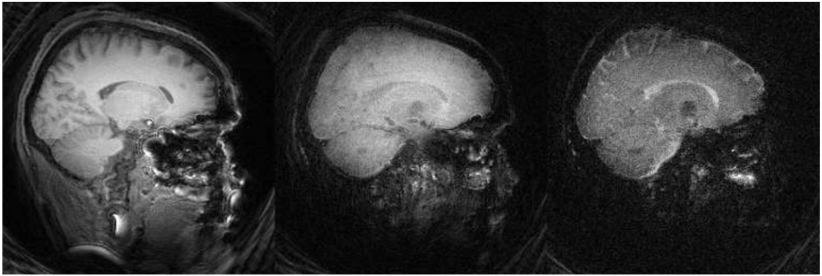
Three often conflicting goals:

- Short scan time to reduce patient discomfort.
- High resolution and fidelity for early detection.
- High signal-to-noise ratio (SNR).

Massively parallel computing provides a disruptive breakthrough.

# reconstructed images

S.S. Stone et al. / J. Parallel Distrib. Comput. 68 (2008) 1307–1318



(a) Gridded Reconstructions. Left to right: GRE, SE1, and SE2.

Main computational result [S.S. Stone et al., 2008]:

- reconstruction time of 23 min on a quad-core CPU,
- reduced to just over 1 min on Quadro FX 5600.

# problem formulation

The reconstructed image $m(\mathbf{r})$ is

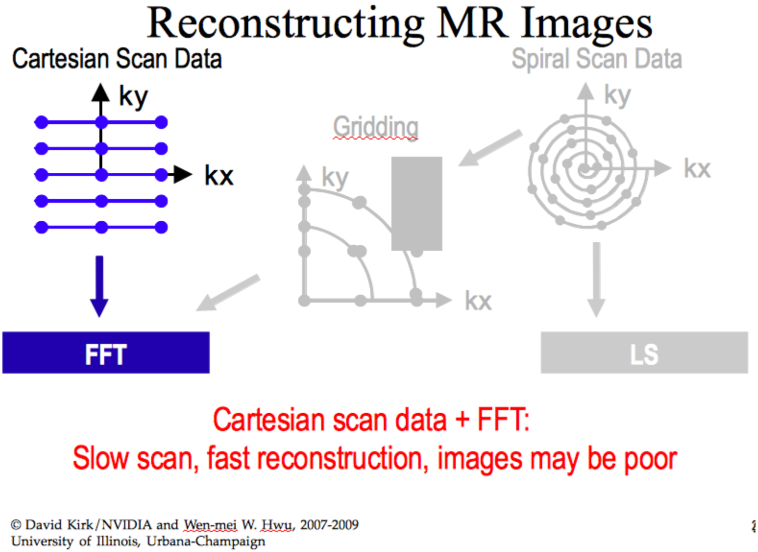
$$\widehat{m}(\mathbf{r}) = \sum_j W(\mathbf{k}_j) s(\mathbf{k}_j) e^{i2\pi \mathbf{k}_j \cdot \mathbf{r}}$$

where

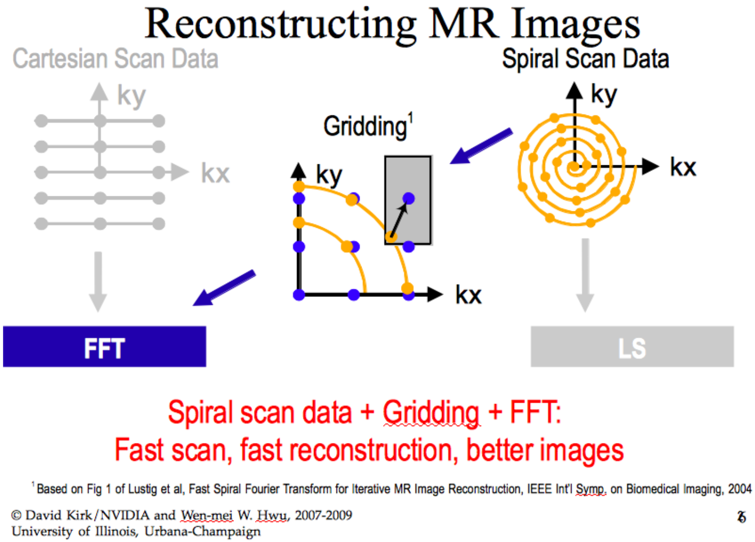
- $s(\mathbf{k})$ is the measured $k$-space data; and
- $W(\mathbf{k})$ is the weighting function to account for nonuniform sampling.

The reconstruction is an inverse fast Fourier transform on $s(\mathbf{k})$.

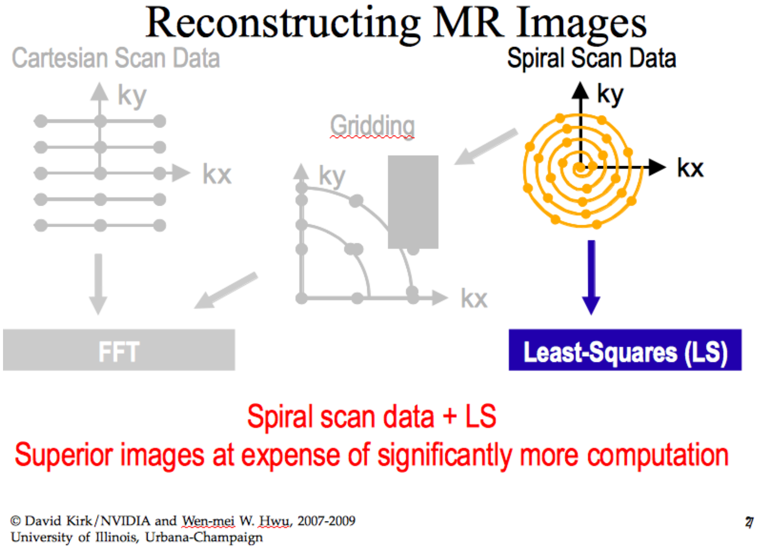
# Cartesian trajectory with FFT reconstruction



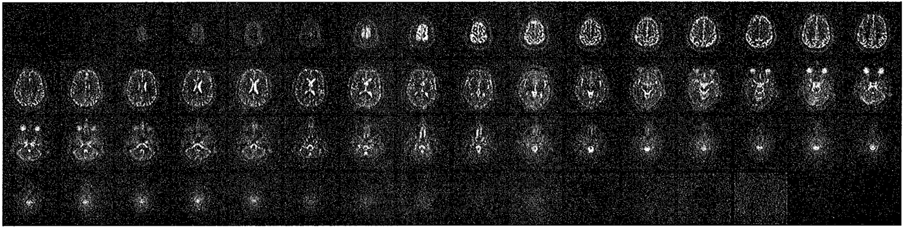
# spiral trajectory, gridding to enable FFT



# spiral trajectory with linear solver reconstruction



# sodium is much less abundant than water



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

**FIGURE 8.2**

The use of non-Cartesian *k*-space sample trajectory and accurate linear-solver-based reconstruction has resulted in new MRI modalities with exciting medical applications. The improved SNR allows reliable collection of *in vivo* concentration data on such chemical substances as sodium in human tissues. The variation or shifting of sodium concentration is an early sign of disease development or tissue death; for example, the sodium map of a human brain can provide an early indication of brain tumor tissue responsiveness to chemotherapy protocols, thus enabling individualized medicine.

# Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction

2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

# a linear least squares problem

A quasi-Bayesian estimation problem:

$$\widehat{\rho} = \arg \min_{\rho} \underbrace{||\mathbf{F}\rho - \mathbf{d}||_2^2}_{\text{data fidelity}} + \underbrace{||\mathbf{W}\rho||_2^2}_{\text{prior info}},$$

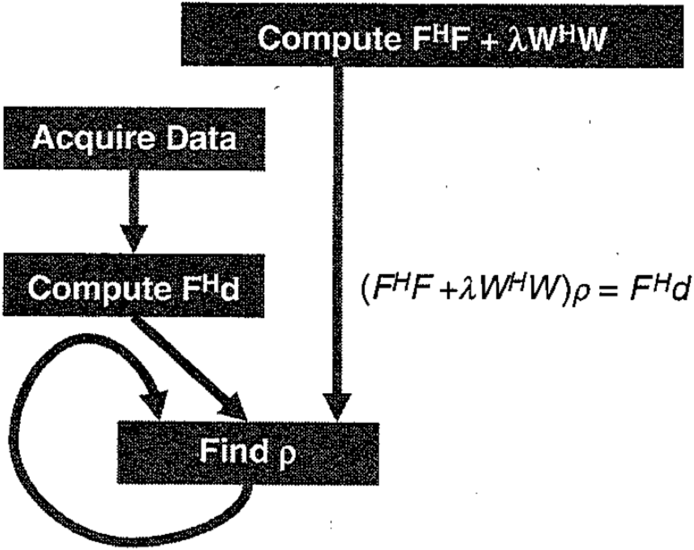
where

- $\widehat{\rho}$ contains voxel values for reconstructed image,
- the matrix **F** models the imaging process,
- **d** is a vector of data samples, and
- the matrix **W** incorporates prior information, derived from reference images.

The solution to this linear least squares problem is

$$\widehat{\rho} = \left( \mathbf{F}^H \mathbf{F} + \mathbf{W}^H \mathbf{W} \right)^{-1} \mathbf{F}^H \mathbf{d}.$$

# an iterative linear solver



# three primary computations

The advanced reconstruction algorithm consists of

1. $$Q(\mathbf{x}_n) = \sum_{m=1}^{M} |\phi(\mathbf{k}_m)|^2 e^{i2\pi \mathbf{k}_m \cdot \mathbf{x}_n}$$
   where $\phi(\cdot)$ is the Fourier transform of the voxel basis function.
2. $$\left[\mathbf{F}^H \mathbf{d}\right]_n = \sum_{m=1}^{M} \phi^*(\mathbf{k}_m)\mathbf{d}(\mathbf{k}_m) e^{i2\pi \mathbf{k}_m \cdot \mathbf{x}_m}$$
3. The conjugate gradient solver performs the matrix inversion to solve $\left(\mathbf{F}^H\mathbf{F} + \mathbf{W}^H\mathbf{W}\right)\rho = \mathbf{F}^H\mathbf{d}$.

The calculation for $\mathbf{F}^H\mathbf{d}$ is an excellent candidate for acceleration on the GPU because of its substantial data parallelism.

# structured matrices

Concerning the left of

$$\left( \mathbf{F}^H \mathbf{F} + \mathbf{W}^H \mathbf{W} \right) \rho = \mathbf{F}^H \mathbf{d}$$

observe that

1. the matrix $\mathbf{W}$ is sparse, and
2. $\mathbf{F}^H \mathbf{F}$ has a convolutional structure
   which enables efficient matrix multiplication via the FFT.

   It is faster to compute $\mathbf{F}^H \mathbf{F} \rho$ via matrix-vector multiplications, instead of forming the matrix $\mathbf{F}^H \mathbf{F}$.

Therefore, the GPU acceleration has its focus on $\mathbf{F}^H \mathbf{d}$.

# Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction

2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

# computing $\mathbf{F}^H\mathbf{d}$

```
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;
   }
}
```

Consider the Compute to Global Memory Access (CGMA) ratio.

# the CGMA ratio of the beginning of the loop

The Compute to Global Memory Access (CGMA) ratio is
the #floating-point operations performed for each memory access.

Consider the beginning of the loop:

```
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 each `m`, we count

1. 4 multiplications, one addition, one subtraction;
2. 10 memory accesses.

So the CGMA ratio is $6/10$.

# a first version of the 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;

   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;
   }
}
```

# Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction
2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

# splitting the outer loop

```
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;
   }
}
```

# a kernel for the first loop

We convert the first loop into a CUDA kernel:

```
__global__ void cmpMu ( float *rPhi,iPhi,rD,iD,rMu,iMu)
{
   int m = blockIdx * MU_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];
}
```

Because $M$ can be very big, we will have many threads.

For example, if $M = 65,536$, with 512 threads per block,
we have $65,536/512 = 128$ blocks.

To improve the CGMA ratio of 0.6,
use registers and/or shared memory.

# a kernel for the second loop

```
__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;
   }
}
```

For the computation of the CGMA ratio, assumptions must be made for the #floating-point operations for `cos(expFHd)` and `sin(expFHd)`.

# Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction

2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

# thread `m` computes `n`-th data element

```
__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;
   }
}
```

A conflict between *different* threads `i` and `j` arises
when they write in the *same* memory locations.

# loop interchange

To avoid conflicts between threads,
we interchange the inner and the outer loops:

```
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;
   }
}
```

```
for(n=0; n<N; n++)
{
   for(m=0; m<M; m++)
   {
      expFHd = 2*PI*(kx[m]*x[n]
                    +ky[m]*y[n]
                    +ky[m]*y[n]);
      cArg = cos(expFHd);
      sArg = sin(expFHd);
      rFHd[n] += rMu[m]*cArg
               - iMu[m]*sArg;
      rFHd[n] += iMu[m]*cArg
               + rMu[m]*sArg;
   }
}
```

In the new kernel, the `n`-th element will be computed by the `n`-th thread.

# a new kernel

```
__global__ void cmpFHd ( float* rPhi, iPhi, phiMag,
                kx, ky, kz, x, y, z, rMu, iMu, int M )
{
   int n = blockIdx.x*FHD_THREAD_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;
   }
}
```

For a $128^3$ image, there are $(2^7)^3 = 2,097,152$ threads.
For higher resolutions, e.g.: $512^3$, multiple kernels may be needed.

# Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction

2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

# 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_THREAD_PER_BLOCK + threadIdx.x;
   float xn = x[n]; float yn = y[n]; float zn = z[n];
   float rFHdn = rFHd[n]; float iFHdn = iFHd[n];
   for(m = 0; m < M; m++)
   {
      float expFHd = 2*PI*(kx[m]*xn+ky[m]*yn+kz[m]*zn);
      float cArg = cos(expFHd);
      float sArg = sin(expFHd);
      rFHdn += rMu[m]*cArg - iMu[m]*sArg;
      iFHdn += iMu[m]*cArg + rMu[m]*sArg;
   }
   rFHd[n] = rFHdn; iFHd[n] = iFHdn;
}
```

Consider the improved Compute to Memory Access (CGMA) ratio.

# Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction

2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

# chunking *k*-space data into constant memory

Using constant memory we use cache more efficiently.
Limited in size to 64KB, we need to invoke the kernel multiple times.

```
__constant__ float kx[CHUNK_SZ],ky[CHUNK_SZ],kz[CHUNK_SZ];
// code omitted ...
for(i = 0; k < M/CHUNK_SZ; i++)
{
   cudaMemcpy(kx,&kx[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemCpyHostToDevice);
   cudaMemcpy(ky,&ky[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemCpyHostToDevice);
   cudaMemcpy(kz,&kz[i*CHUNK_SZ],4*CHUNK_SZ,
              cudaMemCpyHostToDevice);
   // code omitted ...
   cmpFHD<<<FHd_THREADS_PER_BLOCK,
            N/FHd_THREADS_PER_BLOCK>>>
     (rPhi,iPhi,phiMag,x,y,z,rMu,iMu,M);
}
```

# adjusting the memory layout

Due to size limitations of constant memory and cache, instead of storing the components of *k*-space data in three separate arrays, we use an array of structs:

```
struct kdata
{
   float x, float y, float z;
}
__constant struct kdata k[CHUNK_SZ];
```

and then in the kernel we use `k[m].x`, `k[m].y`, and `k[m].z`.

# Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging (MRI)
   - iterative reconstruction

2. Acceleration on GPU
   - determining the kernel parallelism structure
   - loop splitting
   - loop interchange
   - using registers to reduce memory accesses
   - chunking data to fit into constant memory
   - using hardware trigonometry functions

# using hardware trigonometry functions

Instead of `cos` and `sin` as implemented in software, the hardware versions `__cos` and `__sin` provide a much higher throughput.

The `__cos` and `__sin` are implemented as hardware instructions executed by the special function units.

We need to be careful about a loss of accuracy.

The validation involves a "perfect" image:

- a reverse process to generate "scanned" data;
- metrics: mean square error & signal-to-noise ratios.

The last stage is the experimental performance tuning.

# references

This lecture is based on Chapter 8 (first edition; or Chapter 11 for the second edition) in the book of Kirk & Hwu.


- A. Lu, I.C. Atkinson, and K.R. Thulborn. **Sodium Magnetic Resonance Imaging and its Bioscale of Tissue Sodium Concentration**. *Encyclopedia of Magnetic Resonance*, John Wiley & Sons, 2010.
- S.S. Stone, J.P. Haldar, S.C. Tsao, W.-m.W. Hwu, B.P. Sutton, and Z.-P. Liang. **Accelerating advanced MRI reconstructions on GPUs**. *Journal of Parallel and Distributed Computing* 68(10): 1307–1318, 2008.
- The IMPATIENT MRI Toolset, open source software available at

`http://impact.crhc.illinois.edu/mri/mri_toolset.aspx`.