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, 12 April 2023
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 (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


(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.
The reconstructed image $m(r)$ is

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

where

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

The reconstruction is an inverse fast Fourier transform on $s(k)$. 
Cartesian trajectory with FFT reconstruction

Reconstructing MR Images

Cartesian Scan Data

Spiral Scan Data

Gridding

FFT

LS

Cartesian scan data + FFT:
Slow scan, fast reconstruction, images may be poor

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
University of Illinois, Urbana-Champaign
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

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009
University of Illinois, Urbana-Champaign
spiral trajectory with linear solver reconstruction

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, 2007-2009
University of Illinois, Urbana-Champaign
sodium is much less abundant than water

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.

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

\[ \hat{\rho} = \arg \min_\rho \left\{ ||F\rho - d||^2_2 + ||W\rho||^2_2 \right\}, \]

where

- \( \hat{\rho} \) 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

\[ \hat{\rho} = \left( F^H F + W^H W \right)^{-1} F^H d. \]
an iterative linear solver

\[ (F^H F + \lambda W^H W) \rho = F^H d \]
three primary computations

The advanced reconstruction algorithm consists of

1. $$Q(x_n) = \sum_{m=1}^{M} |\phi(k_m)|^2 e^{i2\pi k_m \cdot x_n}$$
   where $$\phi(\cdot)$$ is the Fourier transform of the voxel basis function.

2. $$\left[F^H d\right]_n = \sum_{m=1}^{M} \phi^*(k_m) d(k_m) e^{i2\pi k_m \cdot x_m}$$

3. The conjugate gradient solver performs the matrix inversion to solve $$\left(F^H F + W^H W\right) \rho = F^H d$$.

The calculation for $$F^H d$$ is an excellent candidate for acceleration on the GPU because of its substantial data parallelism.
**structured matrices**

Concerning the left of

\[
\left( F^H F + W^H W \right) \rho = F^H d
\]

observe that

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

It is faster to compute \( F^H F \rho \) via matrix-vector multiplications, instead of forming the matrix \( F^H F \).

Therefore, the GPU acceleration has its focus on \( F^H d \).
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 $F^H d$

\begin{verbatim}
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;
}
}
\end{verbatim}

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

Consider the beginning of the loop:

```c
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_BLOK + 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. 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

```c
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:

```c
__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

```c
__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 \( \cos(\text{expFHd}) \) and \( \sin(\text{expFHd}) \).
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

```c
__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:

```c
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;
    }
}
```

```c
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.
__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

```c
__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] = rFHD[n]; iFHD[n] = iFHD[n];
}
```

Consider the improved Compute to Memory Access (CGMA) ratio.
an Application Case Study
- magnetic resonance imaging (MRI)
- iterative reconstruction

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.

```c
__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);
}
```
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:

```c
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 \texttt{cos} and \texttt{sin} as implemented in software, the hardware versions \texttt{\_\_cos} and \texttt{\_\_sin} provide a much higher throughput.

The \texttt{\_\_cos} and \texttt{\_\_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.
This lecture is based on Chapter 8 (first edition; or Chapter 11 for the second edition) in the book of Kirk & Hwu.

