Case Study: Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging
   - 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 38
Introduction to Supercomputing
Jan Verschelde, 18 November 2016
1. an Application Case Study
   - magnetic resonance imaging
   - 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 disruptive breakthrough.
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

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

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
spiral trajectory, gridding to enable FFT

Reconstructing MR Images

Cartesian Scan Data

Spiral Scan Data

Gridding

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 Intl 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

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

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

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

an Application Case Study
- magnetic resonance imaging
- 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 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

Acquire Data

Compute $F^Hd$

Compute $F^HF + \lambda W^HW$

$(F^HF + \lambda W^HW)\rho = F^Hd$

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

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.
Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging
   - 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 $F^Hd$

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.
a first version of the kernel

```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;

    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
   - 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.
__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;
    }
}
Advanced MRI Reconstruction

1. an Application Case Study
   - magnetic resonance imaging
   - 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
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;
    }
}
```

In the new kernel, the \( n \)-th element will be computed by the \( n \)-th thread.
a new kernel

```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;

    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.
1. **an Application Case Study**
   - magnetic resonance imaging
   - 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] = rFHDn; iFHD[n] = iFHDn;
}
```

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

1. an Application Case Study
   - magnetic resonance imaging
   - 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.

```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);
}
```
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:

```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
   - 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 $\text{__cos}$ and $\text{__sin}$ provide a much higher throughput.

The $\text{__cos}$ and $\text{__sin}$ 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.


- The IMPATIENT MRI Toolset, open source software available at [http://impact.crhc.illinois.edu/mri.php](http://impact.crhc.illinois.edu/mri.php).