L13: Application Case Studies

Outline

- Application Case Studies
  - Advanced MRI Reconstruction
    Reading: Kirk and Hwu, Chapter 8 or From the sample book http://courses.ece.illinois.edu/ece498/al/textbook/Chapter7-MRI-Case-Study.pdf

Reconstructing MR Images

- Cartesian Scan Data
  - FFT
  - Gridding
  - Carteisan scan data + FFT: Slow scan, fast reconstruction, images may be poor

- Spiral Scan Data
  - Gridding
  - Least-Squares (LS)
  - Spiral scan data + LS: Superior images at expense of significantly more computation
Least-Squares Reconstruction

\[ F^H F \rho = F^H d \]

- \( Q \) is a data structure that allows us to compute \( F^H F \)
- \( Q \) depends only on scanner configuration
- \( F^H d \) depends on scan data
- Aim: find \( \rho \) using linear solver
- Accelerate \( Q \) and \( F^H d \) on GPU
  - \( Q \): 1-2 days on CPU
  - \( F^H d \): 6-7 hours on CPU
  - \( \rho \): 1.5 minutes on CPU

### Algorithms to Accelerate

- Scan data
  - \( M = \) # scan points
  - \( k_x, k_y, k_z = \) 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

### Step 1. Consider Parallelism to Evaluate Partitioning Options

For \( m = 0; m < M; m++ \) {
  \[
  \text{phiMap}[m] = rPhi[m]^*rPhi[m] + iPhi[m]^*iPhi[m];
  \]
  for \( n = 0; n < N; n++ \) {
    \[
    \text{expQ} = 2*PI*(k_x[m]*x[n] + k_y[m]*y[n] + k_z[m]*z[n]);
    \]
    \[
    rQ[n] += \text{phiMap}[m]*\cos(\text{expQ});
    \]
    \[
    iQ[n] += \text{phiMap}[m]*\sin(\text{expQ});
    \]
  }
}

(a) \( Q \) computation

Q v.s. \( F^H d \)

For \( m = 0; m < M; m++ \) {
  \[
  \text{rMu}[m] = rPhi[m]^*rD[m] + iPhi[m]^*iD[m];
  \]
  \[
  \text{iMu}[m] = rPhi[m]^*iD[m] - iPhi[m]^*rD[m];
  \]
  for \( n = 0; n < N; n++ \) {
    \[
    \text{expFhD} = 2*PI*(k_x[m]*x[n] + k_y[m]*y[n] + k_z[m]*z[n]);
    \]
    \[
    \text{cArg} = \cos(\text{expFhD});
    \]
    \[
    \text{sArg} = \sin(\text{expFhD});
    \]
    \[
    rFhD[n] += \text{rMu}[m]*\text{cArg} - \text{iMu}[m]*\text{sArg};
    \]
    \[
    iFhD[n] += \text{iMu}[m]*\text{cArg} + \text{rMu}[m]*\text{sArg};
    \]
  }
}

(b) \( F^H d \) computation

### What about \( M \) total threads?

Note: \( M \) is \( O(\text{millions}) \)

(Step 2) What happens to data accesses with this strategy?
One Possibility

```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;
    }
}
```

One Possibility

```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;
    }
}
```

One Possibility

```c
major flaw: This code does not work correctly! The accumulation needs to use atomic operation.
```

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

```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 (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;
    }
}
```

Loop Fission

```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 (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

```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;
    }
}
```

(b) after loop fission
A Separate cmpMu Kernel

- After loop fission
  - computation is in 2 steps
  - 2 parallel kernels executed one after the other
  - first step implemented below:

```c
__global__ void cmpMu(float* rPhi, iPhi, rD, iD, rMu, iMu)
{
    int m = blockIdx.x * MU_THREADED_PER_BLOCK + threadIdx.x;
    rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
    iMu[m] = rPhi[m]*iD[m] – iPhi[m]*rD[m];
}
```

---

Loop Interchange

- Possible since all iterations of both levels are independent

```c
for (m = 0; m < M; m++) {
    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;
    }
}
```

---

Step 2. New FHd kernel

```c
__global__ void cmpFHd(float* kx,ky,kz, x,y,z, rMu,iMu, int M)
{
    int n = blockIdx.x * FH_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;
    }
}
```
Step 3. Using Registers to Reduce Global Memory Traffic

```c
__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;
}
```

Still too much stress on memory! Note that kx, ky and kz are read-only and based on m

Tiling k-space data to fit into constant memory

```c
__constant__ float kx_c[CHUNK_SIZE], ky_c[CHUNK_SIZE], 
    kz_c[CHUNK_SIZE];

__global__ void main() {
    int i;
    for (i = 0; i < M/CHUNK_SIZE; i++) {
        cudaMemcpyToSymbol(kx_c,&kx[i*CHUNK_SIZE],4*CHUNK_SIZE);
        cudaMemcpyToSymbol(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE);
        cudaMemcpyToSymbol(kz_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE);
        cmpFHd<<<N/FHD_THREADS_PER_BLOCK, 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

```c
__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;
}
```
Sidebar: Cache-Conscious Data Layout

- kx, ky, kz, and phi components of same scan point have spatial and temporal locality
  - Prefetching
  - Caching
  - Old layout does not fully leverage that locality
  - New layout does fully leverage that locality

Adjusting K-space Data Layout

```c
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);
        cmpFHD<<<FHD_THREADS_PER_BLOCK,N/FHD_THREADS_PER_BLOCK>>>(...);
    }
}
```

Overcoming Mem BW Bottlenecks

- Old bottleneck: off-chip BW
  - Solution: constant memory
  - FP arithmetic to off-chip loads: 421 to 1
- New bottleneck: trig operations
  - 22.8 GFLOPS (FHD)
Using Super Function Units

- Old bottleneck: trig operations
  - Solution: SFUs
- Performance
  - 92.2 GFLOPS (F\textsuperscript{H}d)

New bottleneck: overhead of branches and address calculations

Sidebar: Effects of Approximations

- Avoid temptation to measure only absolute error (I - 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

\[ \text{MSE} = \frac{1}{mn} \sum_{i,j} (f(i,j) - I(i,j))^2 \]
\[ \text{PSNR} = 20 \log_{10} \left( \frac{\text{MAX}(f(i,j))}{\text{MSE}} \right) \]
\[ \text{SNR} = 20 \log_{10} \left( \frac{\text{A}}{\sqrt{\text{MSE}}} \right) \]

Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>GFLOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reconstruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run Time (m)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GFLOP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear Solver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run Time (m)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gridding + FFT</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CPU, DP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LS (CPU, DP)</td>
<td>4009</td>
<td>0.3</td>
</tr>
<tr>
<td></td>
<td>518.0</td>
<td>0.4</td>
</tr>
<tr>
<td></td>
<td>519.59</td>
<td>1.59</td>
</tr>
<tr>
<td>LS (CPU, SP)</td>
<td>2678</td>
<td>0.5</td>
</tr>
<tr>
<td></td>
<td>342.1</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>343.99</td>
<td>1.81</td>
</tr>
<tr>
<td>LS (GPU, Naïve)</td>
<td>269</td>
<td>5.3</td>
</tr>
<tr>
<td></td>
<td>41.0</td>
<td>5.4</td>
</tr>
<tr>
<td></td>
<td>42.85</td>
<td>1.83</td>
</tr>
<tr>
<td>LS (GPU, CMem)</td>
<td>720</td>
<td>18.6</td>
</tr>
<tr>
<td></td>
<td>9.9</td>
<td>22.8</td>
</tr>
<tr>
<td></td>
<td>11.37</td>
<td>1.57</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU)</td>
<td>13.6</td>
<td>98.2</td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>92.2</td>
</tr>
<tr>
<td></td>
<td>4.06</td>
<td>1.66</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU, Exp)</td>
<td>7.5</td>
<td>178.9</td>
</tr>
<tr>
<td></td>
<td>144.5</td>
<td>1.69</td>
</tr>
<tr>
<td></td>
<td>375</td>
<td>7.70</td>
</tr>
</tbody>
</table>

Step 4: Overcoming Bottlenecks (Overheads)

- Old bottleneck: Overhead of branches and address calculations
  - Solution: Loop unrolling and experimental tuning
- Performance
  - 145 GFLOPS (F\textsuperscript{H}d)