L12: Application Case Studies

Outline

• Discussion of strsm
• Project questions (time at end, too)
• Application Case Studies
  – Advanced MRI Reconstruction
    – Reading: Kirk and Hwu, Chapter 7
  – Material Point Method (time permitting)

Administrative Issues

• Next assignment, triangular solve
  – Due 5PM, Tuesday, March 5
  – handin cs6235 lab 3 <probfile>*
• Project proposals
  –Due 5PM, Friday, March 8
  – handin cs6235 prop <pdffile>

Triangular Solve (STRSM)

for (j = 0; j < n; j++)
for (k = 0; k < n; k++)
  if (B[j*n+k] != 0.0f) {
    for (i = k+1; i < n; i++)
      B[j*n+i] -= A[k*n+i] * B[j*n+k];
  }

Equivalent to:
cublasStrsm('l' /* left operator */, 'l' /* lower triangular */,
                   'N' /* not transposed */, 'u' /* unit triangular */,
                   N, N, alpha, d_A, N, d_B, N);

See: http://www.netlib.org/blas/strsm.f
Reconstructing MR Images

Cartesian Scan Data
Spiral Scan Data

Gridding
FFT
LS

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

Spiral scan data + LS Superior images at expense of significantly more computation

Least-Squares Reconstruction

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

- \( Q \) depends only on scanner configuration
- \( F^H d \) depends on scan data
- \( \rho \) found 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

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

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) \( F^H d \) computation
### 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
  - \( r_{FhD}, i_{FhD} \): output pixel data
- **Complexity is \( O(MN) \)**
- **Inner loop**
  - 13 FP MUL or ADD ops
  - 2 FP trig ops
  - 12 loads, 2 stores

### 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;
    }
}
```

**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;
    }
}
```

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

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

(Step 2) What happens to data accesses with this strategy?
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) before loop interchange

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

Figure 7.9 Loop interchange of the FHD computation
Step 3. 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;
}

Tiling of Scan Data

LS recon uses multiple grids
- Each grid operates on all pixels
- Each grid operates on a distinct subset of scan data
- Each thread in the same grid operates on a distinct pixel

Thread n operates on pixel 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)
}

Revised Kernel for Constant Memory

__global__ void cmpFHd(float* 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;
}

Tiling k-space data to fit into constant memory

__constant__ float kx_c[CHUNK_SIZE], ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];
__global__ void main() {
    int i;
    for (i = 0; i < N/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, &kz[i*CHUNK_SIZE], 4*CHUNK_SIZE);
        cmpFHd<<<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
    }
}
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

```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>>>()
  }
}
```

Adjusting K-space Data Layout

```c
__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);
    rFhD[n_] += rMu[m]*cArg - iMu[m]*sArg;
    iFhD[n_] += iMu[m]*cArg + rMu[m]*sArg;
  }
  rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
}
```

Overcoming Mem BW Bottlenecks

- Old bottleneck: off-chip BW
  - Solution: constant memory
  - FP arithmetic to off-chip loads: 421 to 1
- Performance
  - 22.8 GFLOPS (F^2d)
- New bottleneck: trig operations
Using Super Function Units

- Old bottleneck: trig operations
  - Solution: SFUs
- Performance:
  - 92.2 GFLOPS (F^2d)
- New bottleneck: overhead of branches and address calculations

Sidebar: Effects of Approximations

- Avoid temptation to measure only absolute error (I_o – 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

\[
MSE = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (I_i - I_j)^2 \\
SNR = 20 \log_{10} \left( \frac{\text{max} I_i}{\sqrt{MSE}} \right)
\]

Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>GFLOP</th>
<th>GFLOP</th>
<th>Linear Solver (m)</th>
<th>Reconstruction Time (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Griding + FFT (CPU, DP)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LS (CPU, DP)</td>
<td>4009.0</td>
<td>0.3</td>
<td>518.0</td>
<td>0.4</td>
<td>1.59</td>
</tr>
<tr>
<td>LS (CPU, SP)</td>
<td>2678.7</td>
<td>0.5</td>
<td>342.1</td>
<td>0.7</td>
<td>1.81</td>
</tr>
<tr>
<td>LS (GPU, Native)</td>
<td>266.2</td>
<td>5.6</td>
<td>41.0</td>
<td>5.4</td>
<td>1.85</td>
</tr>
<tr>
<td>LS (GPU, CMem)</td>
<td>72.0</td>
<td>18.6</td>
<td>9.4</td>
<td>22.8</td>
<td>1.57</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU)</td>
<td>13.6</td>
<td>98.2</td>
<td>2.4</td>
<td>93.2</td>
<td>1.66</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU, Exp)</td>
<td>7.5</td>
<td>378.0</td>
<td>1.5</td>
<td>144.5</td>
<td>1.69</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^2d)