L12: Application Case Studies

Outline
• Discussion of strsm
• How to approach your projects
• Application Case Studies
  – Advanced MRI Reconstruction
    Reading: Kirk and Hwu, Chapter 7, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter7-MRI-Case-Study.pdf

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

Administrative Issues
• Next assignment, triangular solve
  – Due 5PM, Tuesday, March 15
  – handin cs6963 lab 3 <probfile>
• Project proposals
  – Due 5PM, Monday, March 7 (hard deadline)
  – handin cs6963 prop <pdf file>
Approaching Projects/STRSM/Case Studies

1. Parallelism?
   - How do dependences constrain partitioning strategies?
2. Analyze data accesses for different partitioning strategies
   - Start with global memory: coalesced?
   - Consider reuse: within a thread? Within a block? Across blocks?
3. Data Placement (adjust partitioning strategy?)
   - Registers, shared memory, constant memory, texture memory or just leave in global memory
4. Tuning
   - Unrolling, fine-tune partitioning, floating point, control flow

---

Step 1. Simple Partition for STRSM

```c
__global__ void strsm(int n, float *A, float *B) {
    int bx = blockIdx.x;
    int tx = threadIdx.x;
    int j = bx*THREADSPerBlock + tx; // one thread per column, columns work independently
    int JN = j * n;
    int i, k;
    for (k = 0; k < n; ++k) {    // ROW
        for (i = k+1; i < n; ++i) {  // ALSO row
        }
    }
}
```

Slide source: Mark Hall

---

Reconstructing MR Images

<table>
<thead>
<tr>
<th>Cartesian Scan Data</th>
<th>Spiral Scan Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gridding</td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>Least-Squares (LS)</td>
</tr>
</tbody>
</table>

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

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
- \[ F^H d, \] output pixel data
- Complexity is \( O(MN) \)

Inner loop
- 13 FP MUL or ADD ops
- 2 FP trig ops
- 12 loads, 2 stores

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);
    }
}
```

```
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;
    }
}
```

Step I. Consider Parallelism to Evaluate Partitioning Options

For \( M = 0; m < M; m++ \)
- \( rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m] \)
- \( iMu[m] = rPhi[m]*iD[m] - iPhi[m]*rD[m] \)

What about \( M \) total threads?

Note: \( M \) is \( O(millions) \)

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

```c
__global__ void cmpFHd(float* rPhi, iPhi, phiMag, 
xk, ky, kz, n, 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

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;
}
```

A Separate cmpMu Kernel

```c
__global__ void cmpMu(float* rPhi, iPhi, rD, iD, rMu, iMu) |
int m = blockIdx.x * 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];
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 Separate cmpMu Kernel

(a) FhD computation
(b) after loop fission
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) 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

__global__ void cmpFHd(float* kx, ky, kz, x, y, z, rMu, iMu, int M) {
    int n = blockIdx.x * FHD_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 2. New FHD kernel

__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;
}

Step 3. Using Registers to Reduce Global Memory Traffic

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

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:
Tiling k-space data to fit into constant memory

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

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,FHD_THREADS_PER_BLOCK>>
        ();
}
```

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
__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);
    rFhDn_r += rMu[m]*cArg – iMu[m]*sArg;
    iFhDn_r += iMu[m]*cArg + rMu[m]*sArg;
  }

  rFhD[n] = rFhD_r; iFhD[n] = iFhD_r;
}

Figure 7.16 Adjusting the k-space data memory layout in the F\text{Hd} kernel

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\text{Hd})

Using Super Function Units

- Old bottleneck: trig operations
  - Solution: SFUs
  - Performance
    - 92.2 GFLOPS (F\text{Hd})
  - New bottleneck: overhead of branches and address calculations

Sidebar: Effects of Approximations

- Avoid temptation to measure only absolute error (I_0 – 1)
  - 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

\[
\text{MSE} = \frac{1}{mn} \sum_{i,j} (i(f_i) – I_i(f_j))^2
\]

\[
\text{PSNR} = 20 \log_{10} \left( \frac{\text{max}(i(f_i))}{\sqrt{\text{MSE}}} \right)
\]

Step 4: Overcoming Bottlenecks (Overheads)

- Old bottleneck: Overhead of branches and address calculations
  - Solution: Loop unrolling and experimental tuning
  - Performance: 145 GFLOPS ($P^3_d$)

### Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>$Q$</th>
<th>$P^3_d$</th>
<th>Linear Solver (m)</th>
<th>Recon. Time (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gridding + FFT (CPU, DP)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>0.39</td>
</tr>
<tr>
<td>LS (CPU, DP)</td>
<td>4009.0</td>
<td>0.3</td>
<td>518.0</td>
<td>0.4</td>
</tr>
<tr>
<td>LS (CPU, SP)</td>
<td>3678.7</td>
<td>0.5</td>
<td>342.3</td>
<td>0.7</td>
</tr>
<tr>
<td>LS (GPU, Naive)</td>
<td>260.2</td>
<td>5.1</td>
<td>41.0</td>
<td>3.4</td>
</tr>
<tr>
<td>LS (GPU, CMem)</td>
<td>72.0</td>
<td>18.6</td>
<td>9.8</td>
<td>22.8</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU)</td>
<td>13.6</td>
<td>98.2</td>
<td>2.4</td>
<td>92.2</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU, Exp)</td>
<td>7.5</td>
<td>378.9</td>
<td>1.5</td>
<td>144.5</td>
</tr>
</tbody>
</table>

What’s Coming

- **Next Time**
  - Two applications from last year’s class
- **Near Term**
  - More application patterns (tree/recursive, data reorganization)
  - Review for Midterm (week after spring break)