L13: Application Case Studies

Outline

- Discussion of strsm (more on Monday)
- 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)

```c
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, Monday, March 8
  - handin cs6963 lab 3 <proffield>
- Project proposals
  - Due 5PM, Wednesday, March 17 (hard deadline)
  - handin cs6963 prop <pdffile>
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 strsm1( int n, float *A, float *B )
{
    int bx = blockIdx.x;
    int tx = threadIdx.x;
    int j = bx*THREADS_PER_BLOCK + tx; // one thread per column, columns work independently
    int 2n + j * n;
    int i, k;

    for (k = 0; k < n; ++k) { // ROW
    }
}
```

Reconstructing MR Images

1. Cartesian Scan Data + FFT:
   - Slow scan, fast reconstruction, images may be poor
2. Spiral Scan Data + LS:
   - Superior images at expense of significantly more computation
Least-Squares Reconstruction

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

- Q depends only on scanner configuration
- \( F^H d \) depends on scan data
- Accelerate Q and \( F^H d \) on G80
  - Q: 1-2 days on CPU
  - \( F^H d \): 6-7 hours on CPU
  - \( p \): 1.5 minutes on CPU

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

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
- 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

- What about \( M \) total threads?
  - Note: \( M \) is \( \Omega(\text{millions}) \)
  - (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;
  }
}
```

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

Step 2. New FHd kernel

Tiling of Scan Data

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

Step 3. Using Registers to Reduce Global Memory Traffic

Thread n operates on pixel n:

for (m = 0; m < N/32; 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] = rFhD_r; iFhD[n] = iFhD_r;
}
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;
  }
}
```

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>>>(
      rPhi, iPhi, phiMag, x, y, z, rMu, iMu, int M);
  }
}
```
```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);
    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^3d kernel

---

### Overcoming Mem BW Bottlenecks

- **Old bottleneck:** off-chip BW
- **Solution:** constant memory
- **Performance:** 22.8 GFLOPS (F^3d)

### New bottleneck: trig operations

- **Solution:** SFUs
- **Performance:** 92.2 GFLOPS (F^3d)

### Sidebar: Effects of Approximations

- Avoid temptation to measure only absolute error (I_0 – 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

\[
MSE = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} (f(i,j) - I(i,j))^2
\]

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

Step 4: Overcoming Bottlenecks (Overheads)

- Old bottleneck: Overhead of branches and address calculations
  - Solution: Loop unrolling and experimental tuning
  - Performance: 145 GFLOPS (Fig. 1)

What's Coming

- Before Spring Break
  - Complete MRI application study + more STRSM
  - MPM/GIMP application study from last year (read
    GPU Acceleration of the Generalized Interpolation Material Point Method
    Wei-Fan Chiang, Michael DeLisi, Todd Hummel, Tyler Prete, Kevin Tew, Mary Hall, Phil Wallstedt, and James Guilkey
    (http://saahpc.ncsa.illinois.edu/09/papers/Chiang_paper.pdf)
  - Review for Midterm (week after spring break)