L14: Application Case Studies II

Outline
- 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

Administrative Issues
- Project proposals
  - Due 5PM, Wednesday, March 17 (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
Reconstructing MR Images

**Cartesian Scan Data**
- Slow scan, fast reconstruction, images may be poor

**Spiral Scan Data**
- 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 G80
  - \( Q \): 1-2 days on CPU
  - \( F^H d \): 6-7 hours on CPU
  - \( \rho \): 1.5 minutes on CPU

---

```c
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

```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;
    }
}
```

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

---

**Q v.s. \( F^H d \)**
Algorithms to Accelerate

- Scan data
  - M = # scan points
  - kx, ky, kz = 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

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;
    }
}
```

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;
    }
}
```

Step 1. Consider Parallelism to Evaluate Partitioning Options

- What about M total threads?
- Note: M is O(millions)

(Step 2) What happens to data accesses with this strategy?
A Separate cmpMu Kernel

Step 2. New Fhd kernel

Figure 7.9 Loop interchange of the FhD computation
Step 3. Using Registers to Reduce Global Memory Traffic

```c
__global__ void cmpFHd(float* rPhi, iPhi, phiMag, 
    ks, 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:
```c
for (m = 0; m < M/32; m++) {
    float expFhD = 2*PI*(kx[m]*x[n] + ky[m]*y[n] + kz[m]*z[n]);
    rQ[n] += phi[m]*cos(expFhD);
    iQ[n] += phi[m]*sin(expFhD);
}
```

Revised Kernel for 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,&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

Scan Data

- x, y, z, 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>>>()
    }
}
```

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
  - FP arithmetic to off-chip loads: 421 to 1
- Performance
  - 22.8 GFLOPS (F^3d)
- New bottleneck: trig operations
Recall from L9: Arithmetic Instruction Throughput

- **int and float add, shift, min, max and float mul, mad**: 4 cycles per warp
  - **int multiply (*)** is by default 32-bit
  - Requires multiple cycles / warp
  - Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply

- **Integer divide and modulo are expensive**
  - Compiler will convert literal power-of-2 divides to shifts
  - Be explicit in cases where compiler can't tell that divisor is a power of 2
  - Useful trick: foo % n == foo & (n-1) if n is a power of 2

- **Reciprocal, reciprocal square root, sin/cos, log, exp**: 16 cycles per warp
  - These are the versions prefixed with “__”
  - Examples: __rcp(), __sin(), __exp() (Other functions are combinations of the above
    - y / x == rcp(x) * y == 20 cycles per warp
    - sqrt(x) == rcp(rsqrt(x)) == 32 cycles per warp)

Using Super Function Units

- **Old bottleneck: trig operations**
  - Solution: SFUs
  - Performance: 92.2 GFLOPS (F16)

- **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

\[
\text{MSE} = \frac{1}{mn} \sum (I_o(i,j) - I_i(i,j))^2
\]

\[
\text{PSNR} = 20 \log_{10} \left( \frac{\max(I_o(i,j))}{\sqrt{\text{MSE}}} \right)
\]
Experimental Tuning: Tradeoffs

- In the Q kernel, three parameters are natural candidates for experimental tuning
  - Loop unrolling factor (1, 2, 4, 8, 16)
  - Number of threads per block (32, 64, 128, 256, 512)
  - Number of scan points per grid (32, 64, 128, 256, 512, 1024, 2048)
- Can't optimize these parameters independently
  - Resource sharing among threads (register file, shared memory)
  - Optimizations that increase a thread's performance often increase the thread's resource consumption, reducing the total number of threads that execute in parallel
- Optimization space is not linear
  - Threads are assigned to SMs in large thread blocks
  - Causes discontinuity and non-linearity in the optimization space

Step 4: Overcoming Bottlenecks (Overheads)

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

Summary of Results

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>FHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recons.</td>
<td>Run Time (m)</td>
<td>GFLOP</td>
</tr>
<tr>
<td>Gridding + FFT (CPU, DP)</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>LS (CPU, DP)</td>
<td>4009.0</td>
<td>0.3</td>
</tr>
<tr>
<td>LS (CPU, SP)</td>
<td>2678.7</td>
<td>0.5</td>
</tr>
<tr>
<td>LS (GPU, naive)</td>
<td>268.2</td>
<td>5.1</td>
</tr>
<tr>
<td>LS (GPU, CMem)</td>
<td>72.0</td>
<td>18.6</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU)</td>
<td>13.6</td>
<td>59.2</td>
</tr>
<tr>
<td>LS (GPU, CMem, SFU, Exp)</td>
<td>7.5</td>
<td>357X</td>
</tr>
</tbody>
</table>

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://epboc.ncsa.illinois.edu/09/papers/Chiang_paper.pdf)
  - Review for Midterm (week after spring break)