# L14: Application Case Studies II



## Administrative Issues

- Project proposals
  - Due 5PM, Wednesday, March 17 (hard deadline)
  - handin cs6963 prop <pdffile>



## Outline

- How to approach your projects
- Application Case Studies
  - Advanced MRI Reconstruction

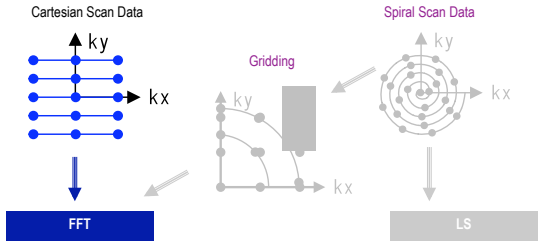
    Reading: Kirk and Hwu, Chapter 7, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter7-MRI-Case-Study.pdf



## 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, …

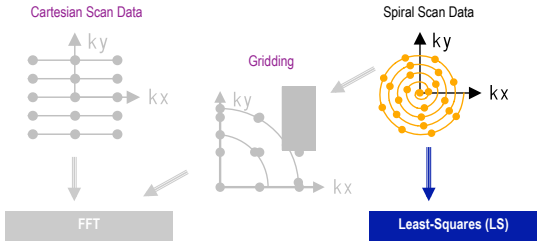
# Reconstructing MR Images




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




# Reconstructing MR Images

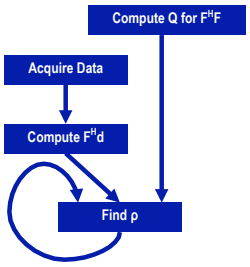



Spiral scan data + LS
Superior images at expense of significantly more computation




# Least-Squares Reconstruction

$$F^H F\rho = F^H d$$




- Q depends only on scanner configuration
- $F^H d$ depends on scan data
- ρ 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
  - ρ: 1.5 minutes on CPU




```
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

Q v.s. $F^H D$

```
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

```
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;
  }
}
```

- 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






# Step 1. 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];

  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(millions)

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



# One Possibility

```
__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?

```
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;
  }
}
```

```
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) $F^{H}d$ 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 (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

```
__global__ void cmpMu(float* rPhi, iPhi, rD, iD, rMu, iMu)
{
  int m = blockIdx.x * MU_THREAEDS_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) 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 $F^{H}D$ computation





# Step 2. New FHd kernel

```
__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 3. Using Registers to Reduce Global Memory Traffic

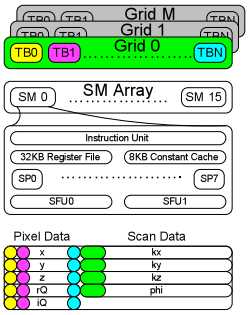
```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
    kx, ky, kz, x, y, z, rMu, iMu, int M) {

  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;
}
```

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:

```
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)
}
```





## Tiling k-space data to fit into constant memory

```
__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 */
}
```

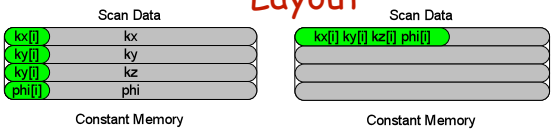




## Revised Kernel for Constant Memory

```
__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

```
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>>>
      ();

 }
```

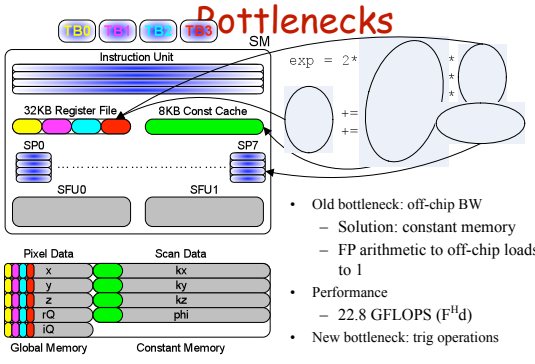



```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
   x, y, z, rMu, iMu, int M) {

  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 $F^Hd$ 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 ($F^Hd$)
- 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

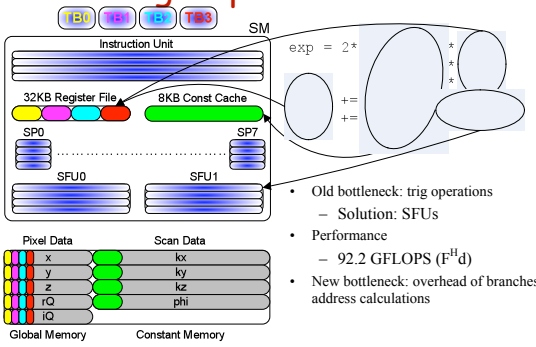



## Recall from L9: Arithmetic Instruction Throughput

- 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 ($F^H d$)
- New bottleneck: overhead of branches and address calculations




## 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

$$MSE = \frac{1}{mn}\sum_i\sum_j (I(i,j) - I_0(i,j))^2 \qquad A_s = \frac{1}{mn}\sum_i\sum_j I_0(i,j)^2$$

$$PSNR = 20\log_{10}\left(\frac{\max(I_0(i,j))}{\sqrt{MSE}}\right) \qquad SNR = 20\log_{10}\left(\frac{\sqrt{A_s}}{\sqrt{MSE}}\right)$$

A.N. Netravali and B.G. Haskell, Digital Pictures: Representation, Compression, and Standards (2nd Ed), Plenum Press, New York, NY (1995).

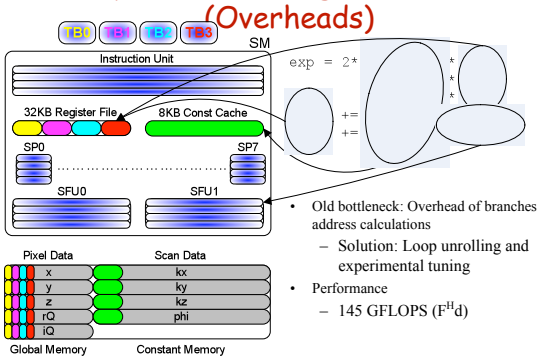
# Experimental Tuning: Tradeoffs

- 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 ($F^H d$)





# Summary of Results

| | Q | | $F^H d$ | | | |
|---|---|---|---|---|---|---|
| Reconstruction | Run Time (m) | GFLOP | Run Time (m) | GFLOP | Linear Solver (m) | Recon. Time (m) |
| Gridding + FFT (CPU, DP) | N/A | N/A | N/A | N/A | N/A | 0.39 |
| LS (CPU, DP) | 4009.0 | 0.3 | 518.0 | 0.4 | 1.59 | 519.59 |
| LS (CPU, SP) | 2678.7 | 0.5 | 342.3 | 0.7 | 1.61 | 343.91 |
| LS (GPU, Naïve) | 260.2 | 5.1 | 41.0 | 5.4 | 1.65 | 42.65 |
| LS (GPU, CMem) | 72.0 | 18.6 | 9.8 | 22.8 | 1.57 | 11.37 |
| LS (GPU, CMem, SFU) | 13.6 | 98.2 | 2.4 | 92.2 | 1.60 | 4.00 |
| LS (GPU, CMem, SFU, Exp) | 7.5 | 178.9 | 1.5 | 144.5 | 1.69 | 3.19 |

357X 228X 108X




# 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)