上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

CS4302-01
Parallel and Distributed Computing

Lecture 7 CUDA, cont'd

Zhuoran Song

2023/10/20

Register File Limitation

- If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  - Each block requires 10*256 = 2560 registers
  - 8192 = **3** * 2560 + change
  - So, three blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 1?
  - Each Block now requires 11*256 = 2816 registers
  - 8192 < 2816 *3
  - Only two Blocks can run on an SM, **1/3 reduction of parallelism**!!!

Dynamic Partitioning

- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism

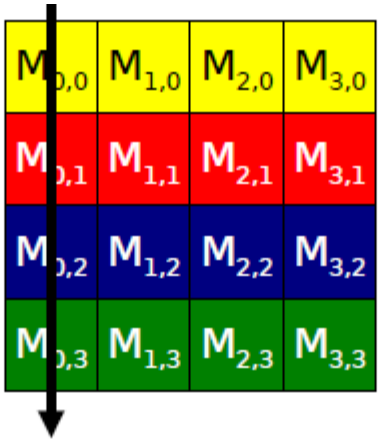
ILP Vs. TLP

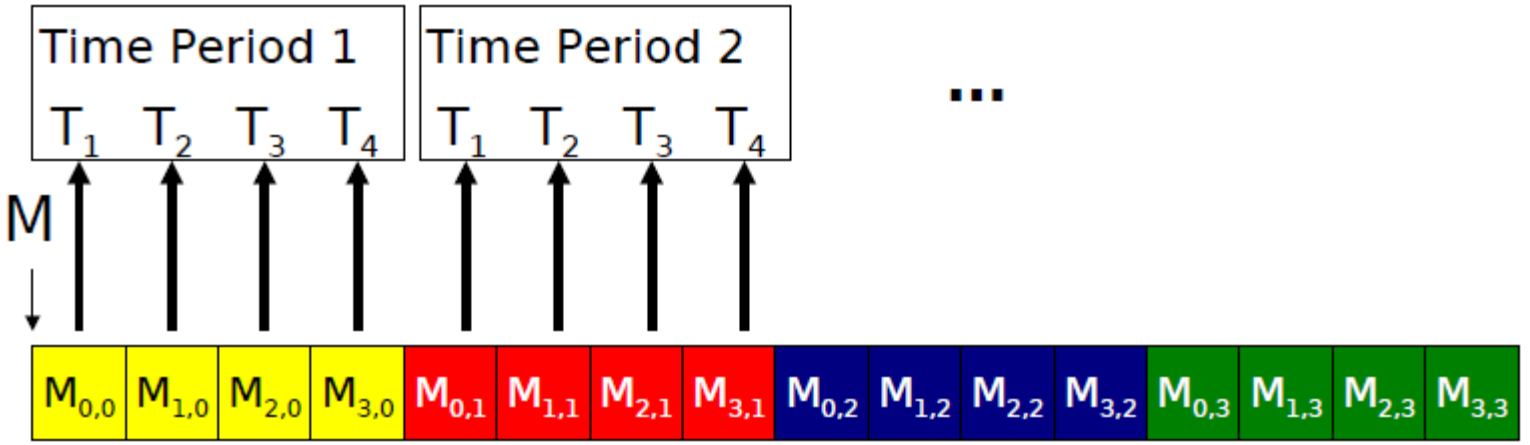
- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 10 registers, can fit 3 blocks global loads have 400 cycles
  - *4 cycles * 4 inst * 24 warps = 384 < 400*

- If a compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load, can only fit 2 blocks
  - *4 cycles * 8 inst * 16 warps = 512 > 400, better hiding memory latency*

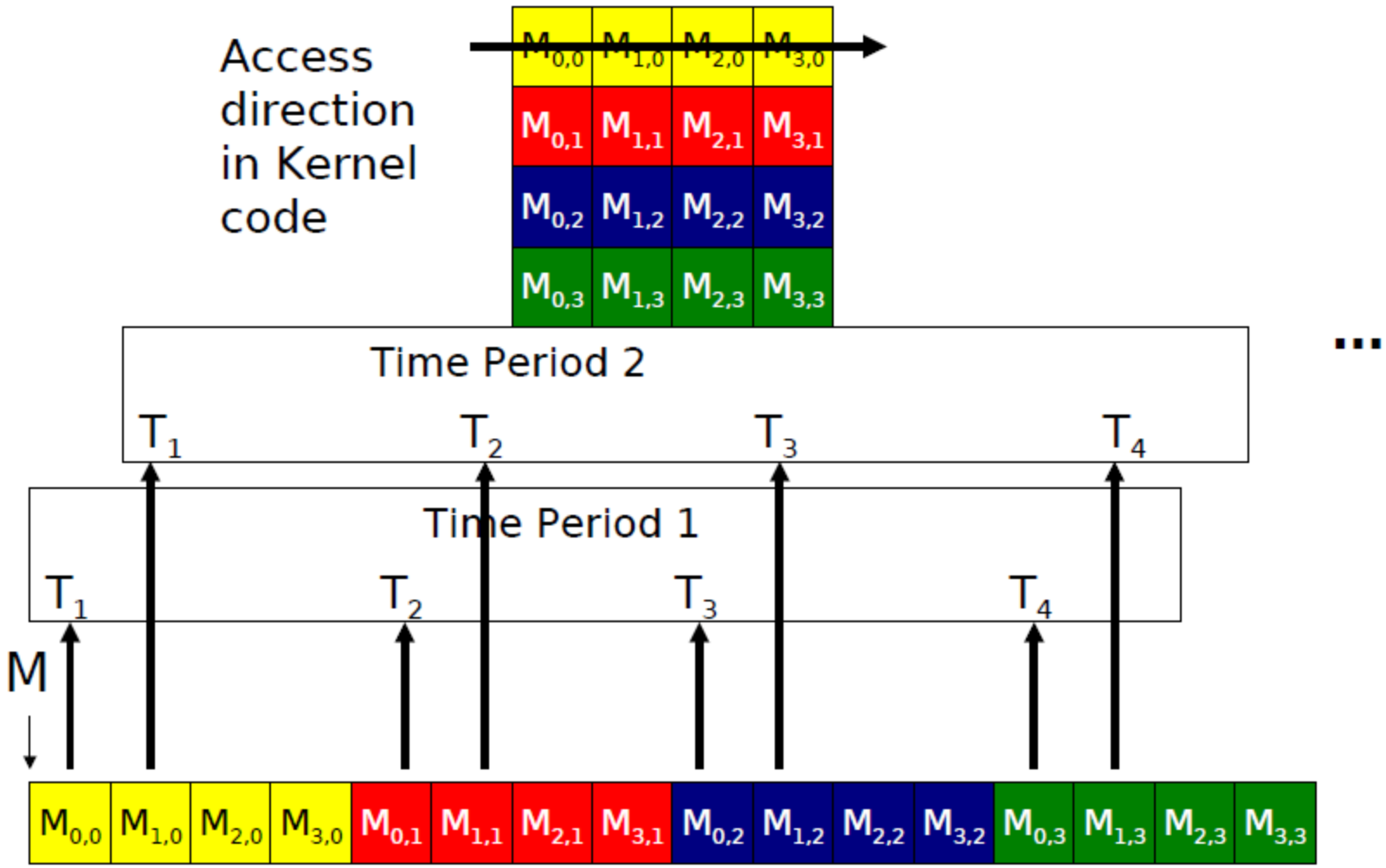
Memory Coalescing

Access direction in Kernel code





Memory Coalescing

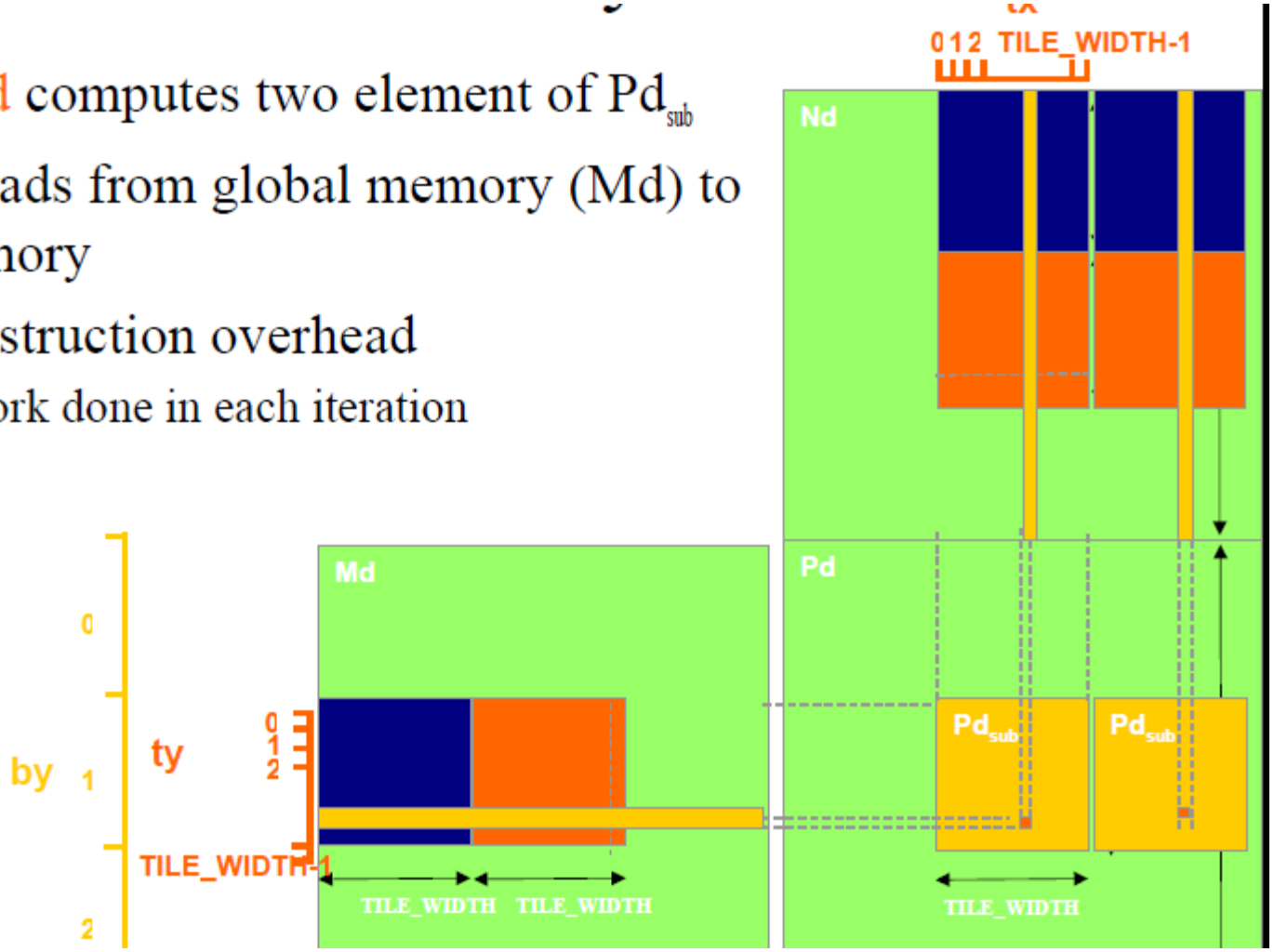


Memory Coalescing

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.   __shared__float Mds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__float Nds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;  int by = blockIdx.y;
4.   int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.   int Row = by * TILE_WIDTH + ty;
6.   int Col = bx * TILE_WIDTH + tx;

7.    float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
// Coolaborative loading of Md and Nd tiles into shared memory
9.     Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.    Nds[ty][tx] = Nd[Col + (m*TILE_WIDTH + ty)*Width];
11.    __syncthreads();

11.   for (int k = 0; k < TILE_WIDTH; ++k)
12.   Pvalue += Mds[ty][k] * Nds[k][tx];
13.   Synchthreads();
14.  }
13.   Pd[Row*Width+Col] = Pvalue;
}
```

Increasing Per Thread Work

- Each thread computes two element of Pd_{sub}
- Reduced loads from global memory (Md) to shared memory
- Reduced instruction overhead
  - More work done in each iteration



Divergence

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in G80
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

Reduction

- Given an array of values, "reduce" them to a single value in parallel
- Examples
  - sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array

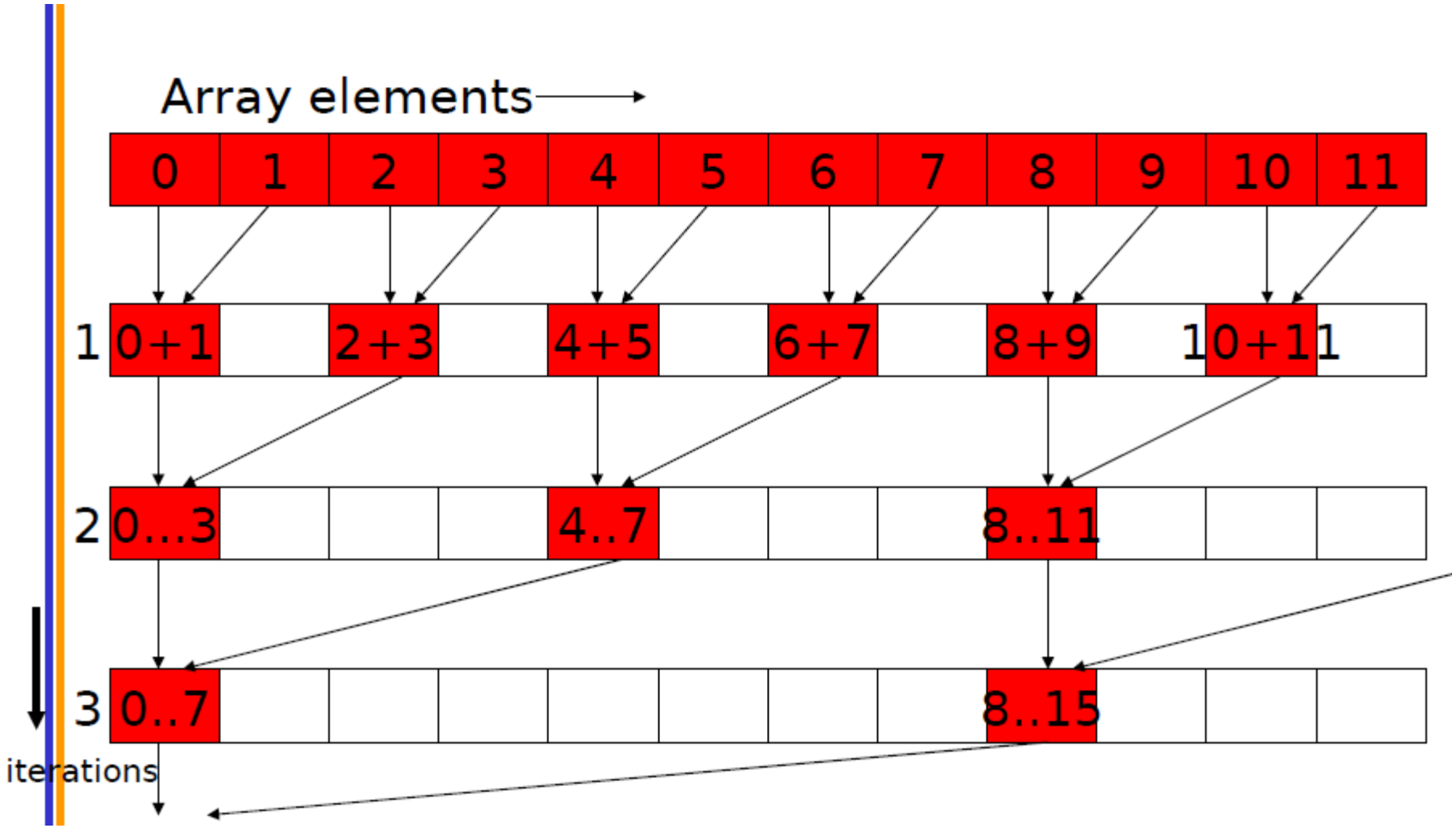
- Typically parallel implementation:
  - Recursively halve # threads, add two values per thread
  - Takes log(n) steps for n elements, requires n/2 threads

Reduction Program

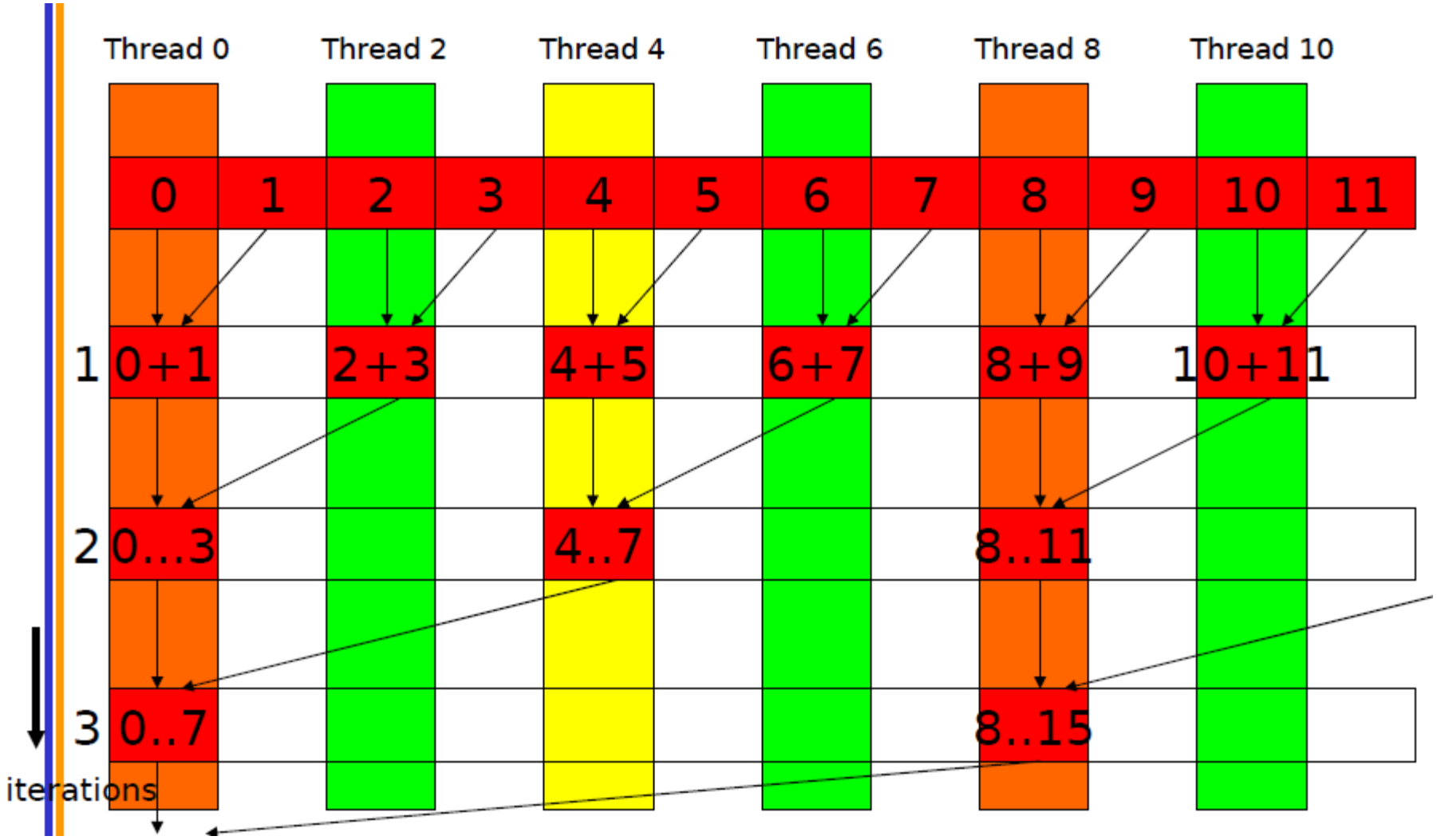
- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
```

Reduction Program



Divergence in Reduction



Problem in the Program

- In each iterations, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

- No more than half of threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time.
  - After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    - This can go on for a while, up to 4 more iterations (512/32=16= 2^4), where each iteration only has one thread activated until all warps retire

Problem in the Program

- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
unsigned int t = threadIdx.x;
for (unsigned int stride = 1;
     stride < blockDim.x;  stride *= 2)
{
  __syncthreads();
  if (t % (2*stride) == 0)
    partialSum[t] += partialSum[t+stride];
}
```

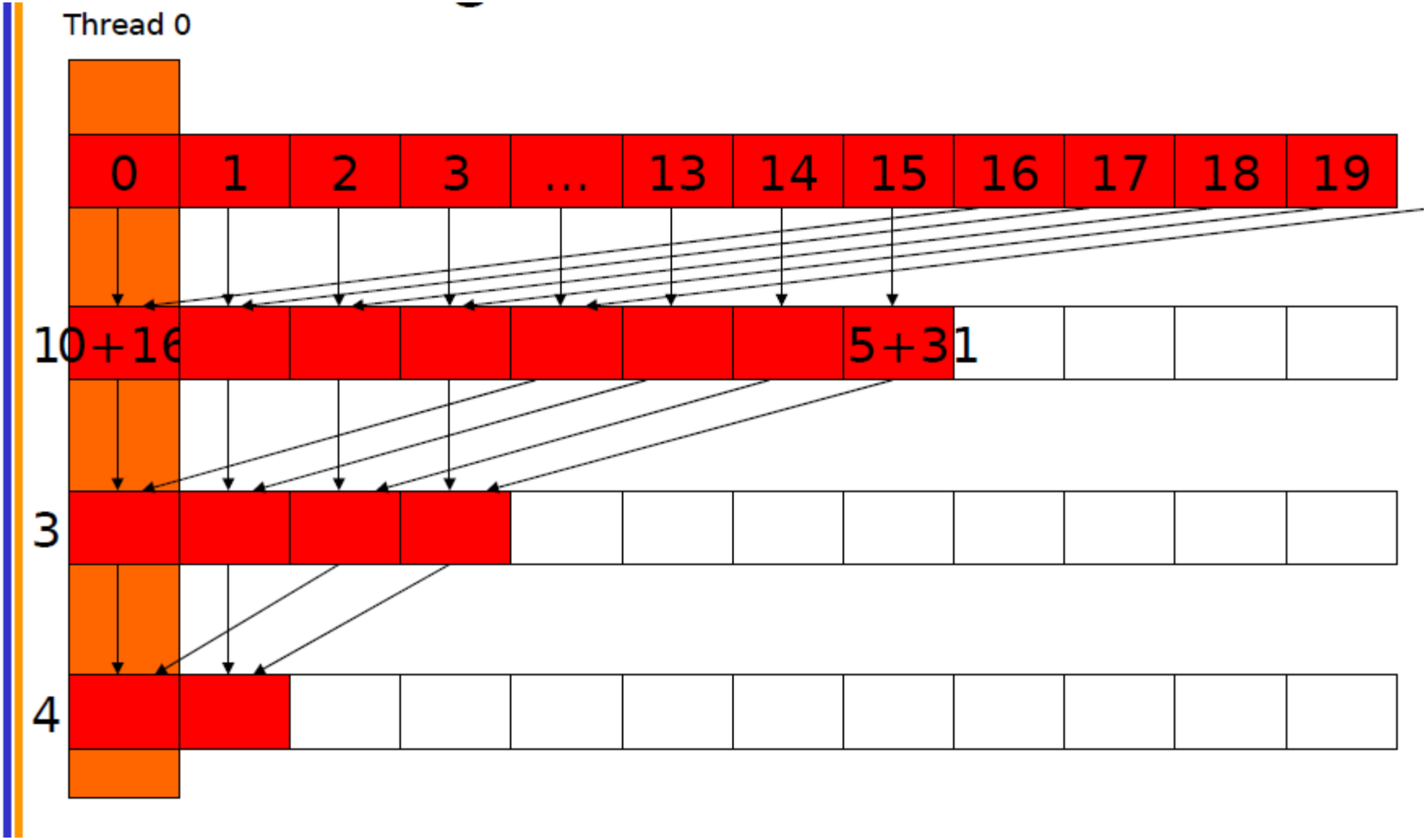
BAD: Divergence due to interleaved branch decisions

Better Implementation

- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x;
     stride > 1;  stride >> 1)
{
  __syncthreads();
  if (t < stride)
    partialSum[t] += partialSum[t+stride];
}
```

Better Implementation



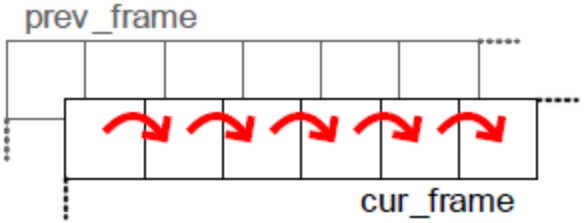
Better Implementation

- Only the last 5 iterations will have divergence
- Entire warps will be shut down as iterations progress
  - For a 512-thread block, 4 iterations to shut down all but one warps in each block
  - Better resource utilization, will likely retire warps and thus blocks faster
- Recall, no bank conflicts either

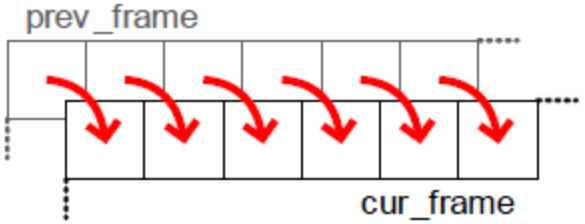
Algorithm Decision

(H.263 motion estimation example)

- Different algorithms may expose different levels of parallelism while achieving desired result
- In motion estimation, can use previous vectors (either from space or time) as guess vectors



(a) Guess vectors are obtained from the previous macroblock .

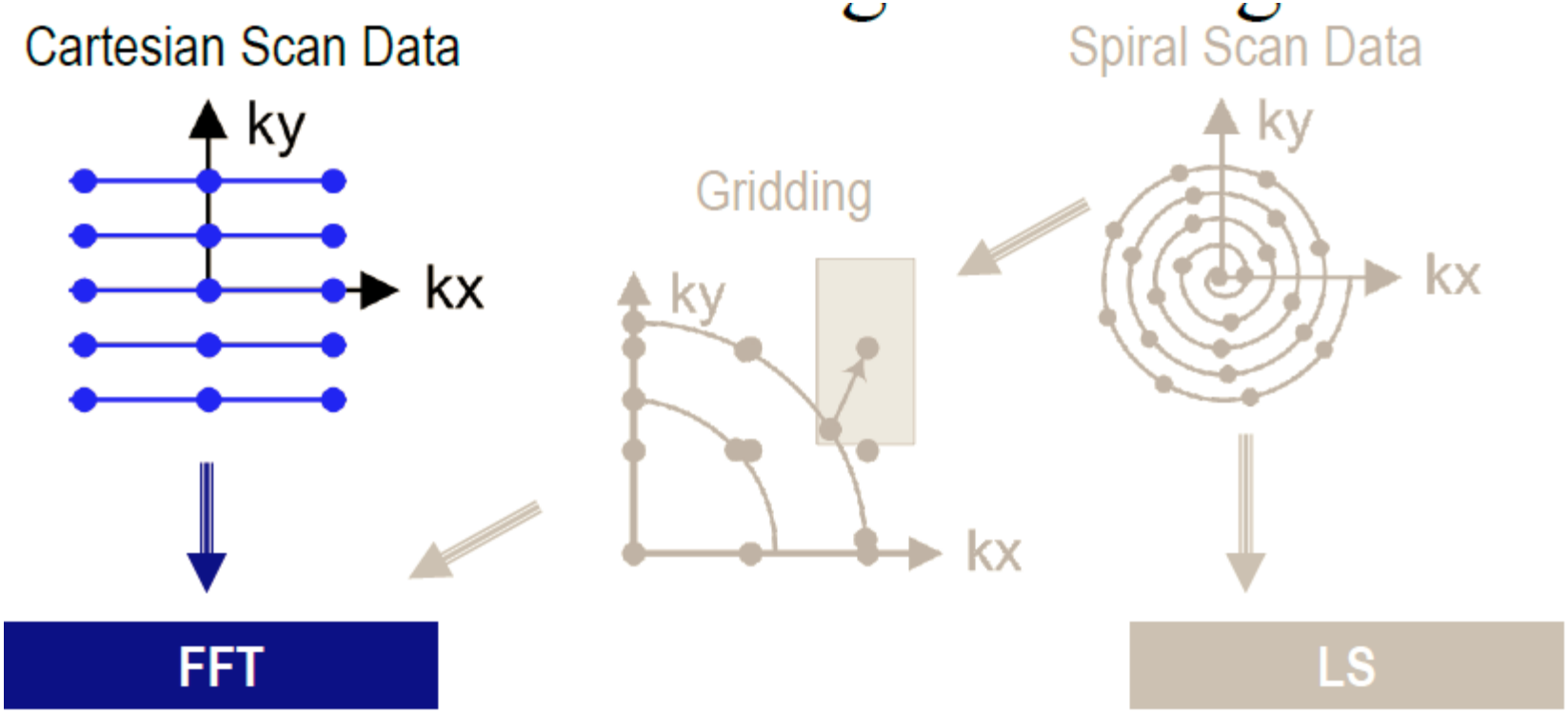


(b) Guess vectors are obtained from the corresponding macroblock in the previous frame .

Global Synchronization

- Remember, there is no barrier synchronization between CUDA thread blocks
  - You can have some limited communication through atomic integer operations on global memory on newer devices
  - Doesn't conveniently address the global reduction problem, or lots of others
- To synchronize, you need to end the kernel (have all thread blocks complete)
  - Then launch a new one

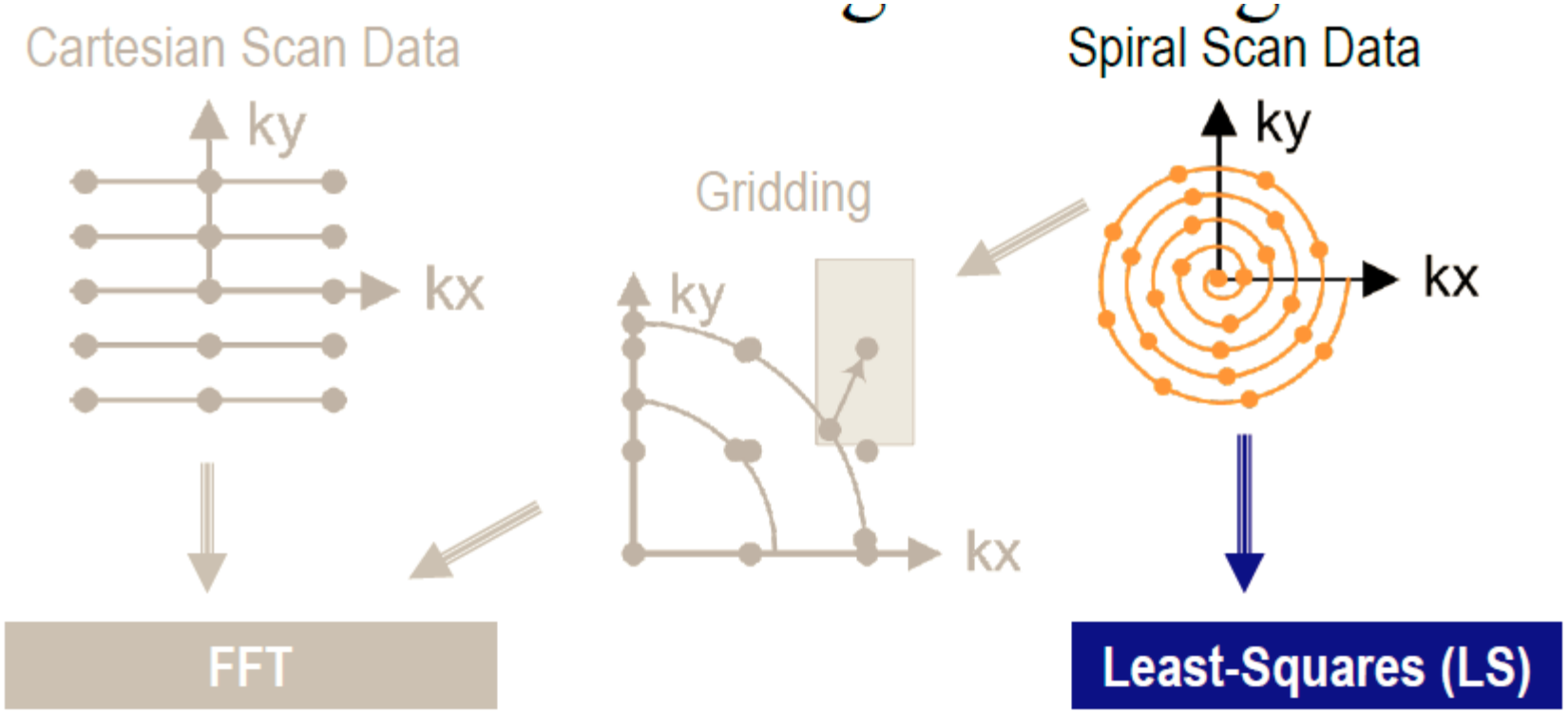
Case Study (MRI)



Cartesian scan data + FFT:

Slow scan, fast reconstruction, images may be poor

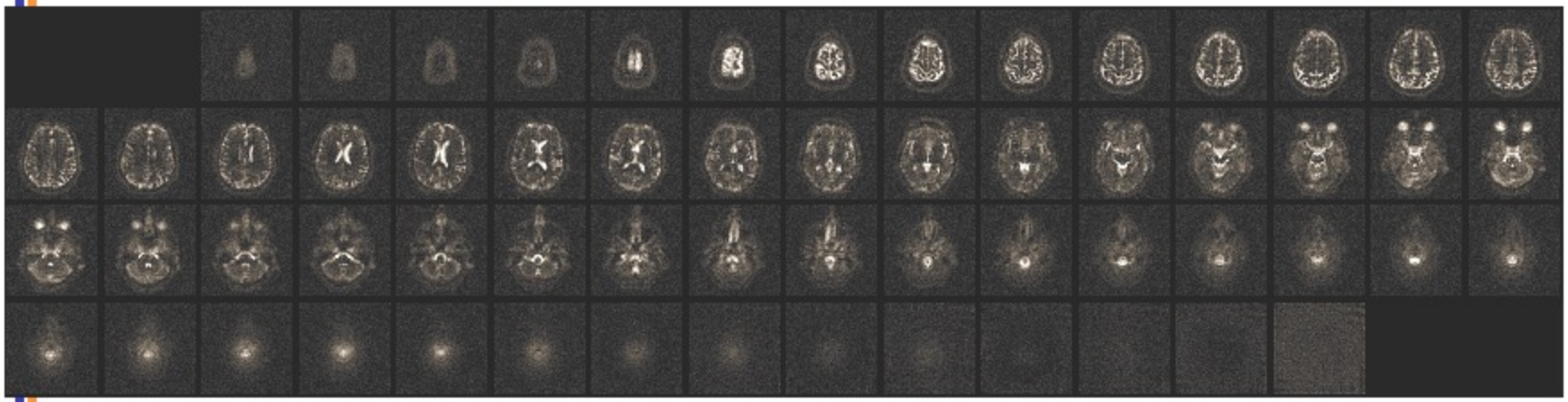
Case Study



Spiral scan data + LS

Superior images at expense of significantly more computation

Case Study

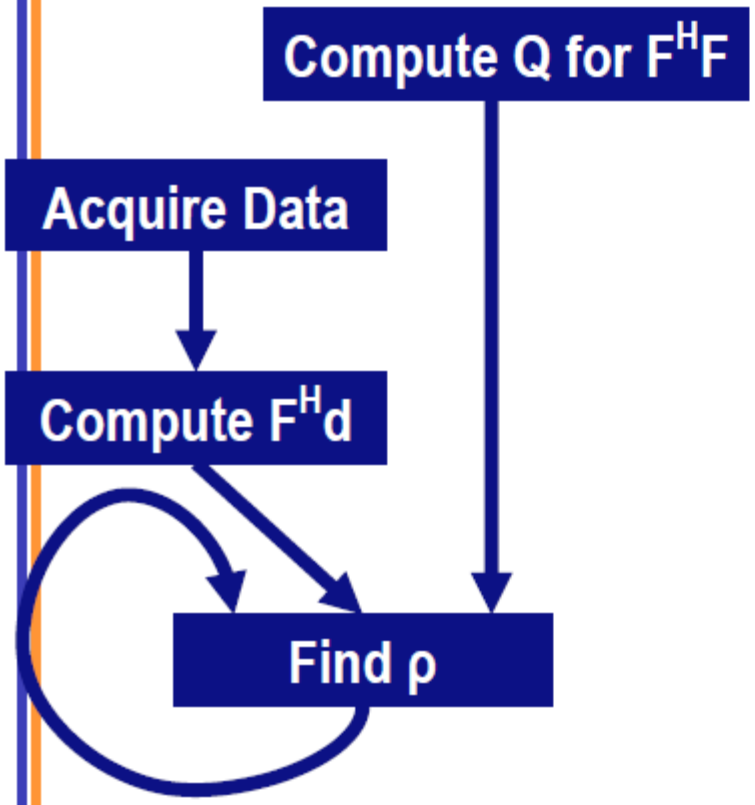


- Images of sodium in the brain
  - Very large number of samples for increased SNR
  - Requires high-quality reconstruction
- Enables study of brain-cell viability before anatomic changes occur in stroke and cancer treatment – within days!

Case Study

Least-Squares Reconstruction

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



- Q depends only on scanner configuration
- F^Hd depends on scan data
- ρ found using linear solver
- Accelerate Q and F^Hd on G80
  - Q: 1-2 days on CPU
  - F^Hd: 6-7 hours on CPU
  - ρ: 1.5 minutes on CPU

Case Study

```
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

Case Study

```
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

Case Study

From C to CUDA: Step 1

What unit of work is assigned to each thread?

```
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;
  }
}
```

Case Study

```
__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

Case Study

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;
  }
}
```

Case Study

```
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++) {
 for (n = 0; n < N; n++) {

   rMu[m] = rPhi[m]*rD[m] +
            iPhi[m]*iD[m];
   iMu[m] = rPhi[m]*iD[m] -
            iPhi[m]*rD[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 code motion

Case Study

```
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

Case Study

```
__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];
}
```

Case Study

```
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

Case Study

```
__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;
  }
}
```

Case Study

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;
}
```

Case Study

Chunking 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++);
    cudaMemcpy(kx_c,&kx[i*CHUNK_SIZE],4*CHUNK_SIZE,
               cudaMemCpyHostToDevice);
    cudaMemcpy(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
               cudaMemCpyHostToDevice);
    cudaMemcpy(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
               cudaMemCpyHostToDevice);
    …
    cmpFHD<<<FHD_THREADS_PER_BLOCK, N/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 */
}
```

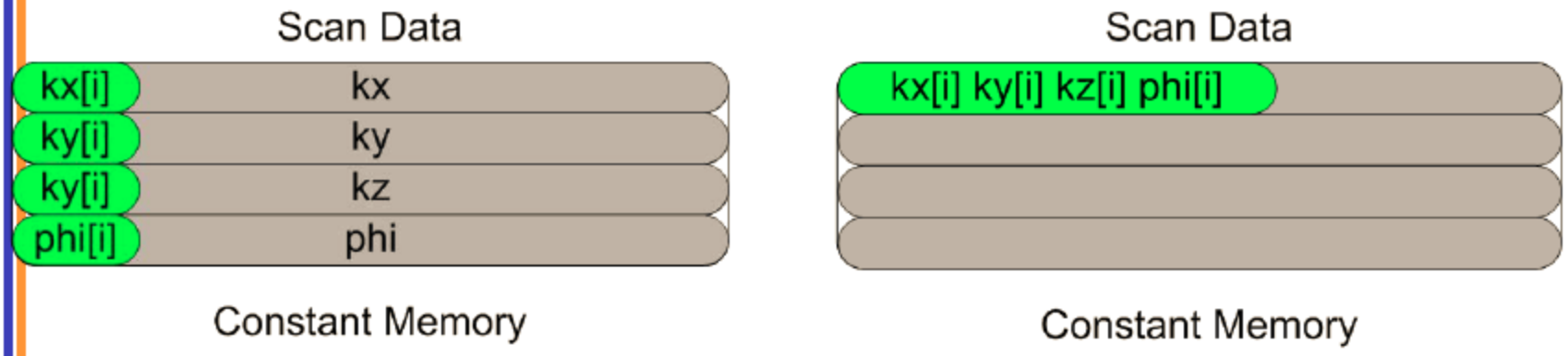
Case Study

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[m]*xn_r+ky[m]*yn_r+kz[m]*zn_r);
                         Kx_c       Ky_c       Kz_c
    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;
}
```

Case Study

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

Case Study

Adjusting K-space Data Layout

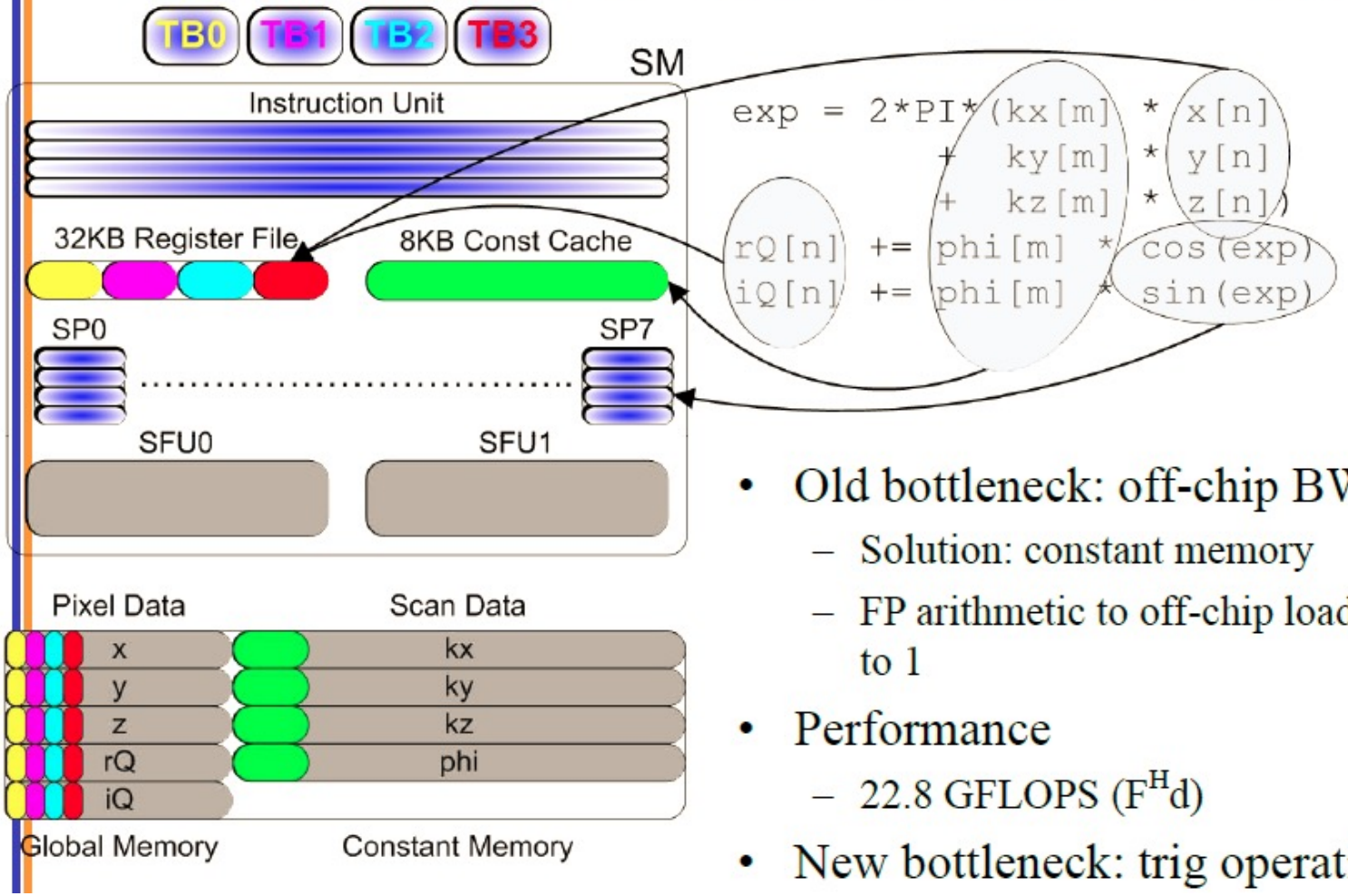
```
struct kdata {
   float x, float y, float z;
} k;

__constant__ struct kdata k_c[CHUNK_SIZE];
…

__ void main() {

 int i;

 for (i = 0; i < M/CHUNK_SIZE; i++);
   cudaMemcpy(k_c,k,12*CHUNK_SIZE, cudaMemCpyHostToDevice);

   cmpFHD<<<FHD_THREADS_PER_BLOCK,N/FHD_THREADS_PER_BLOCK>>>
              ();

  }
```

Case Study

```
__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;
}
```

Case Study

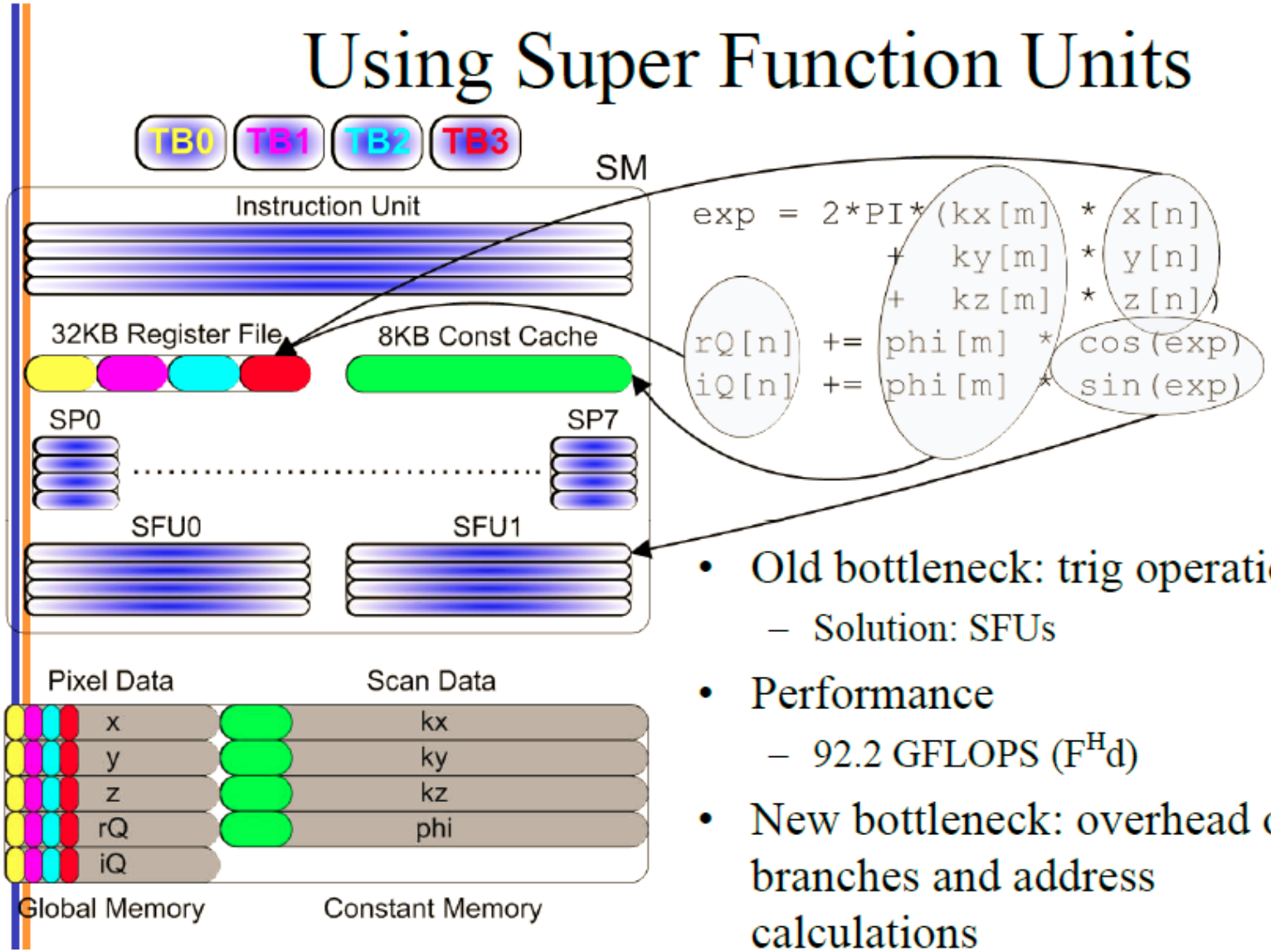
Overcoming Mem BW Bottlenecks



- Old bottleneck: off-chip BW
  - Solution: constant memory
  - FP arithmetic to off-chip loads: 421 to 1
- Performance
  - 22.8 GFLOPS ($F^H d$)
- New bottleneck: trig operations

Case Study

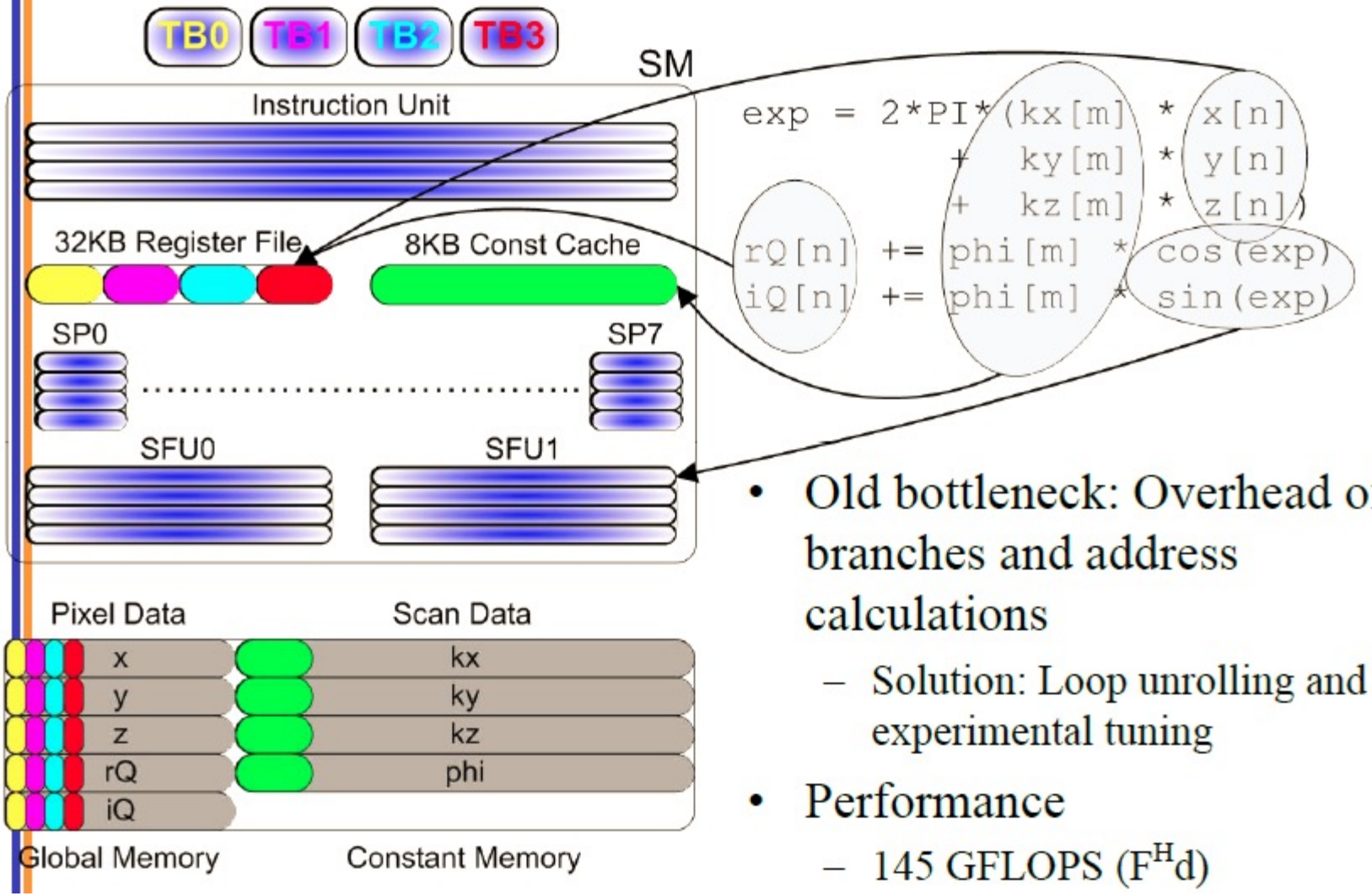
Using Super Function Units



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

Case Study

Step 3: Overcoming Bottlenecks (Overheads)



- Old bottleneck: Overhead of branches and address calculations
  - Solution: Loop unrolling and experimental tuning
- Performance
  - 145 GFLOPS ($F^H d$)

Case Study

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

Case Study

	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

Case Study

+

=