CUDA Application Development

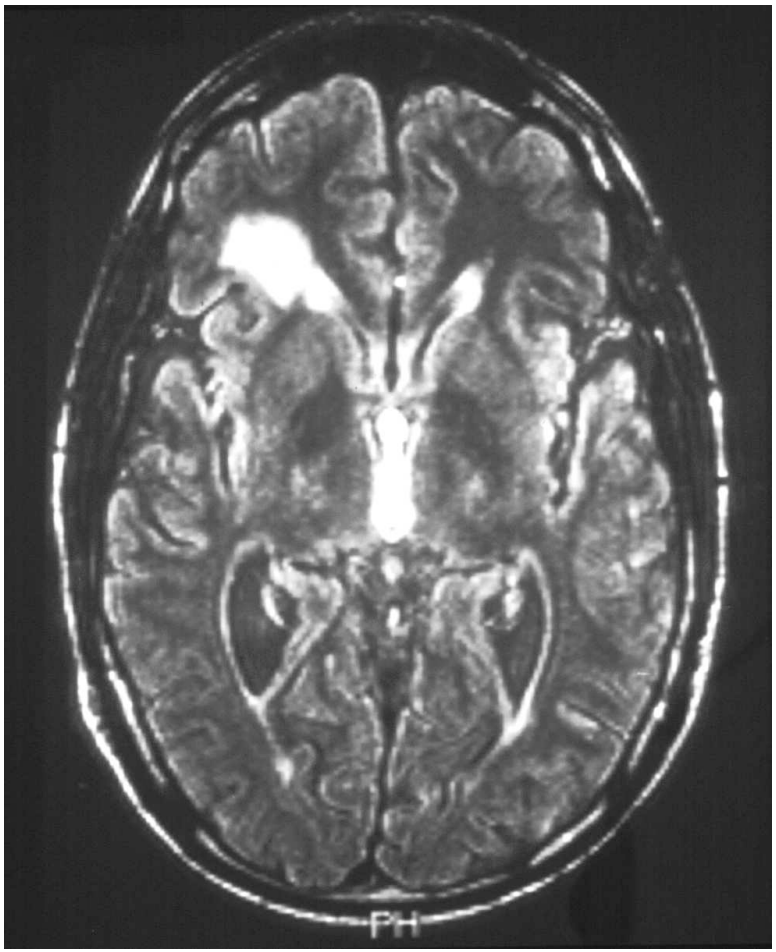
Wen-mei Hwu
University of Illinois, Urbana-Champaign

Science and Engineering Application Speedup

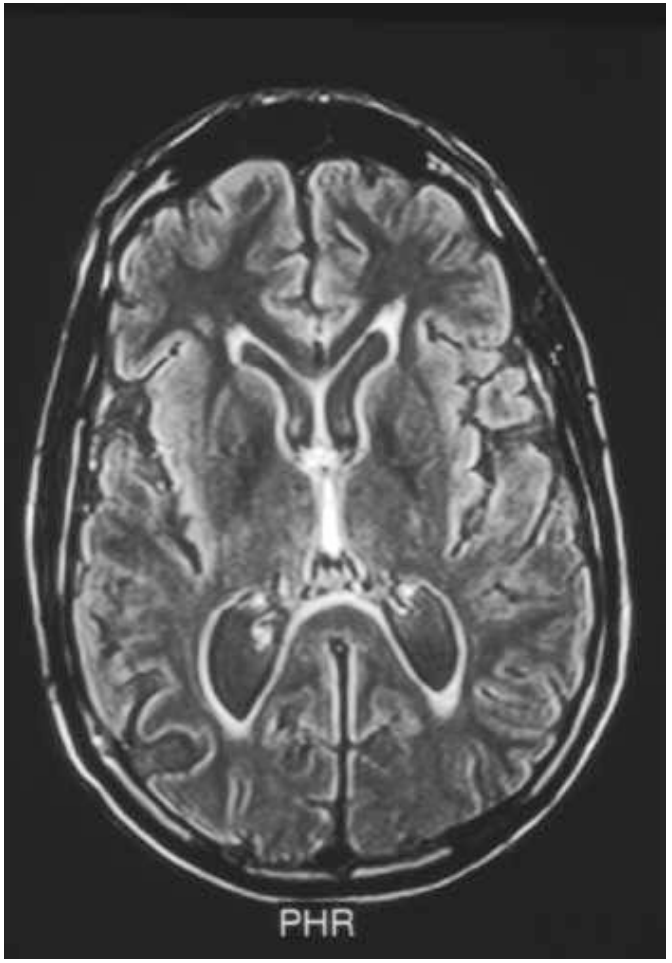
App.	Archit. Bottleneck	Simult. T	Kernel X	App X
H.264	Registers, global memory latency	3,936	20.2	1.5
LBM	Shared memory capacity	3,200	12.5	12.3
RC5-72	Registers	3,072	17.1	11.0
FEM	Global memory bandwidth	4,096	11.0	10.1
RPES	Instruction issue rate	4,096	210.0	79.4
PNS	Global memory capacity	2,048	24.0	23.7
LINPACK	Global memory bandwidth, CPU-GPU data transfer	12,288	19.4	11.8
TRACF	Shared memory capacity	4,096	60.2	21.6
FDTD	Global memory bandwidth	1,365	10.5	1.2
MRI-FHD	Instruction issue rate	8,192	23.0	23.0

[HKR HotChips-2007]

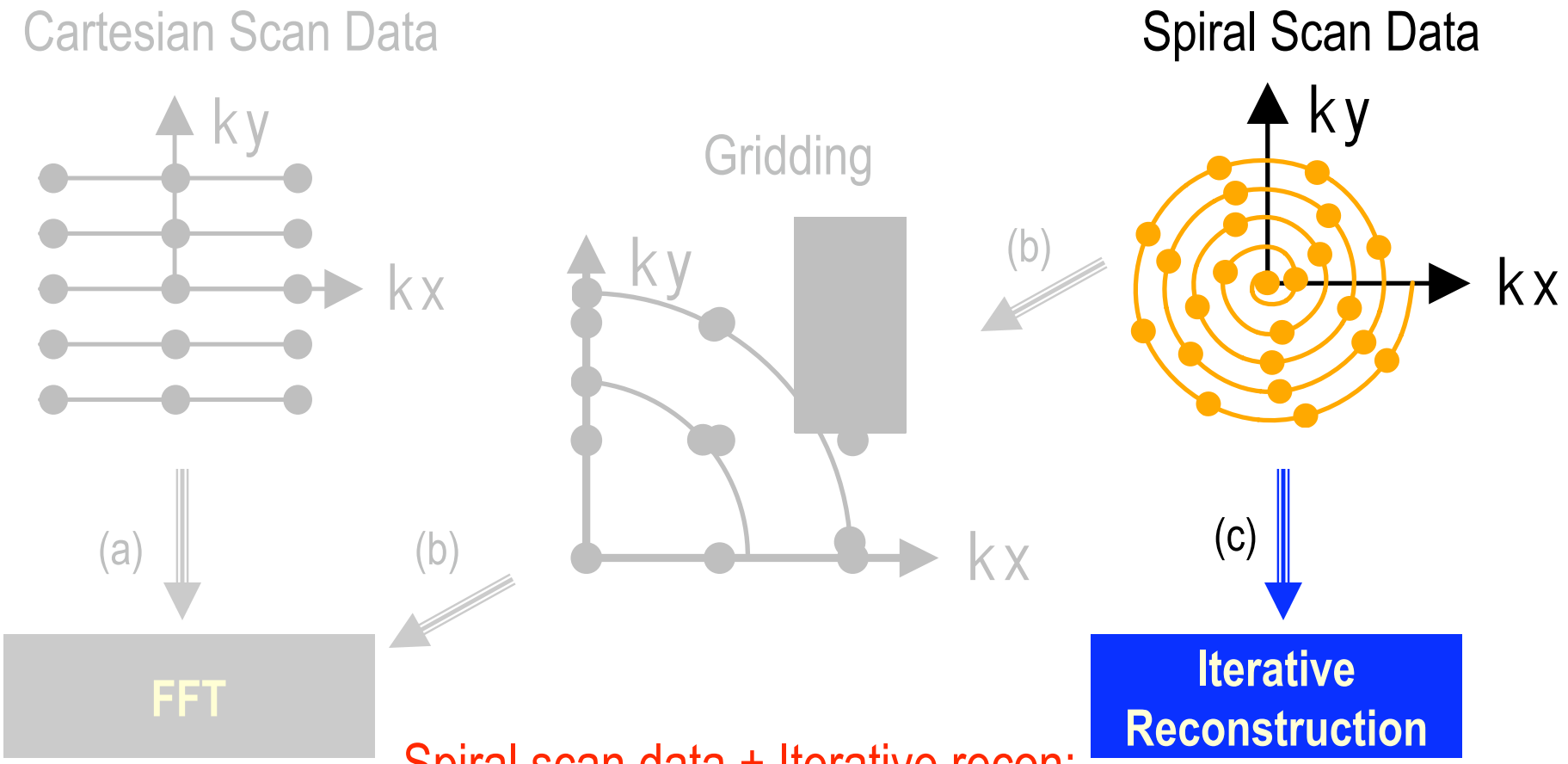
Chemo Therapy Monitoring



6-12 weeks

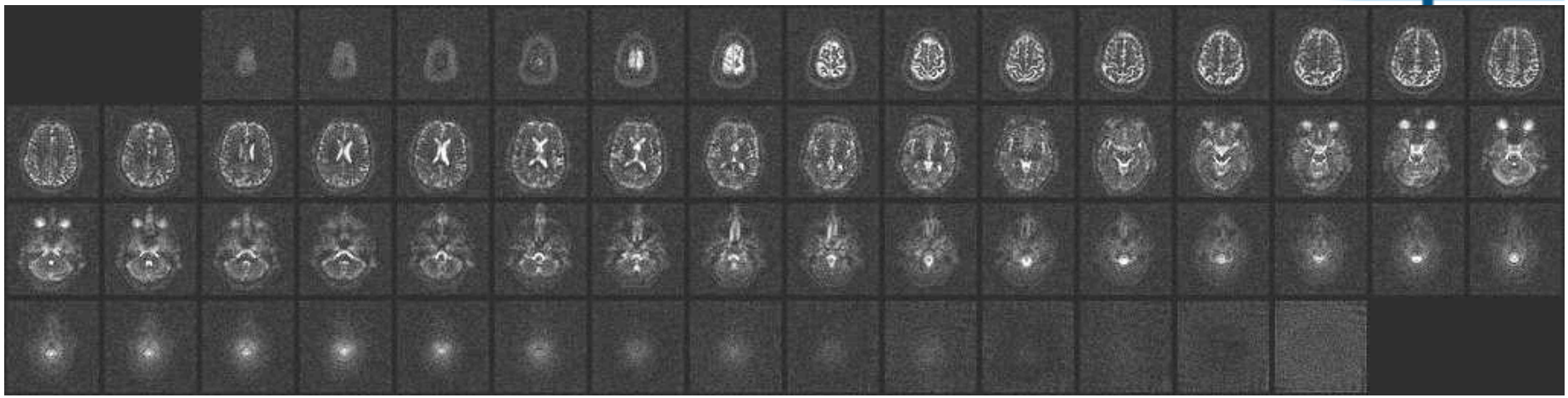


MRI Reconstruction



Spiral scan data + Iterative recon:
Fast scan reduces artifacts, iterative reconstruction increases SNR.
Reconstruction requires a lot of computation.

An Exciting Revolution - Sodium Map of the Brain

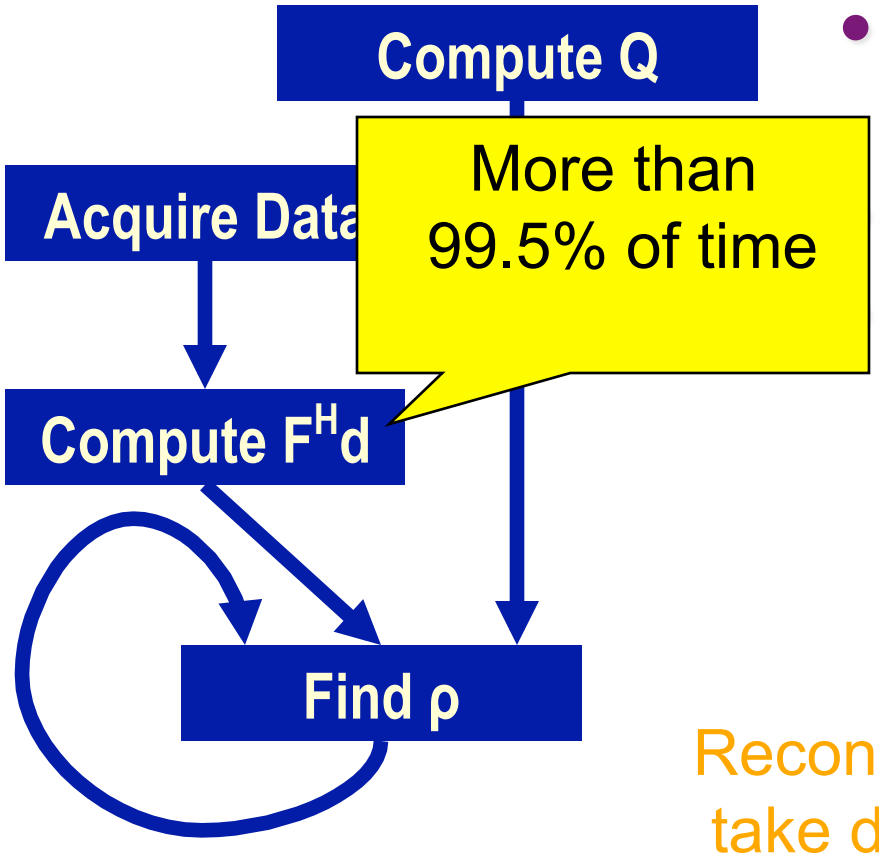


- Images of sodium in the brain
  - Requires powerful scanner (9.4 Tesla)
  - Very large number of samples for increased SNR
  - Requires high-quality reconstruction
- Enables study of brain-cell viability before anatomic changes occur in stroke and cancer treatment – within days!

Courtesy of Keith Thulborn and Ian Atkinson, Center for MR Research, University of Illinois at Chicago

Advanced MRI Reconstruction

$$(F^H F + \lambda W^H W)\rho = F^H d$$

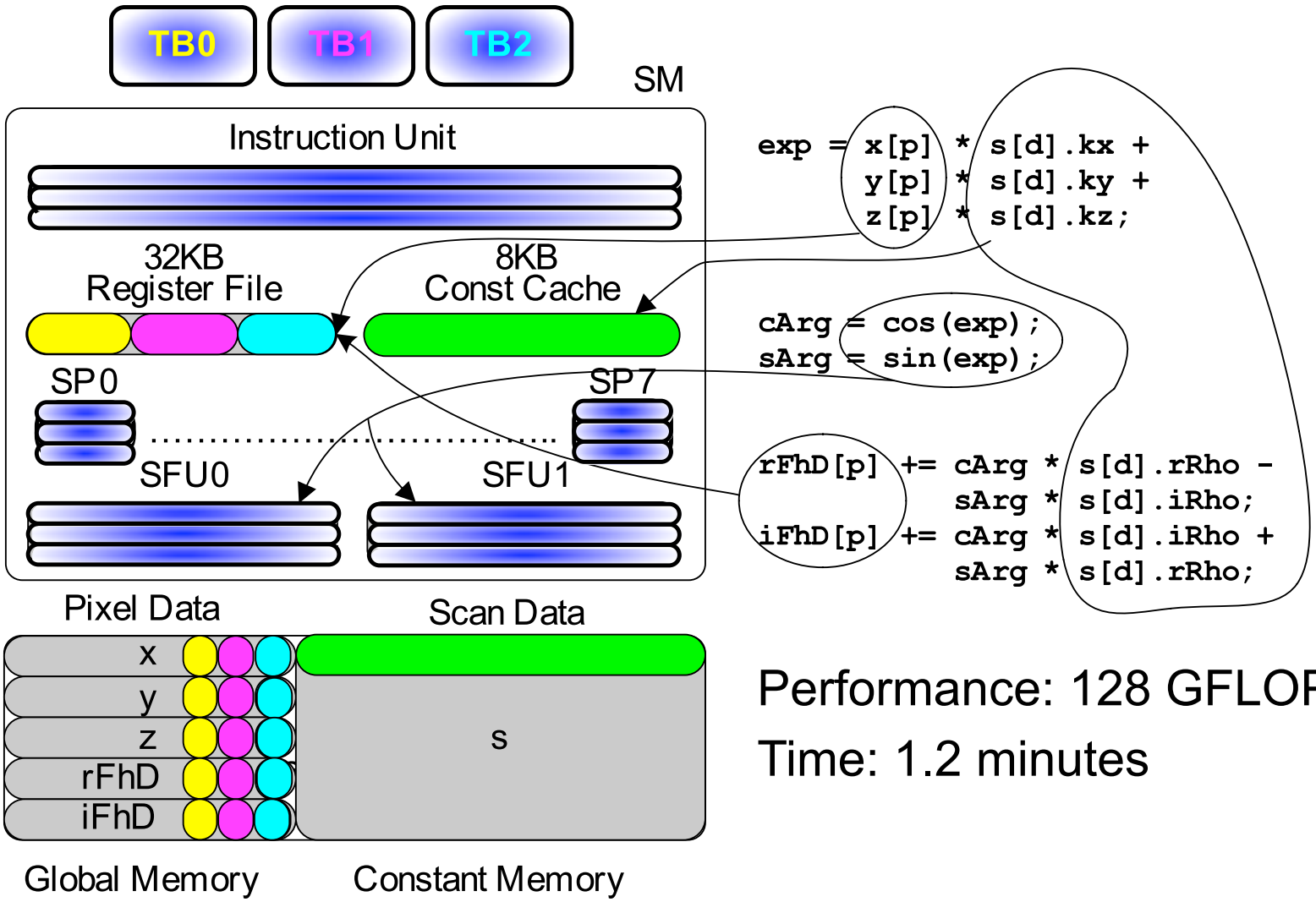


- Q depends only on scanner configuration
  $F^H d$ depends on scan data
  ρ found using linear solver
  - $F^H F$ computed once per iteration; depends on Q, $F^H d$
  - $\lambda W^H W$ incorporates anatomical constraints

Reconstruction of a 64^3 image used to take days!

Haldar, et al, "Anatomically-constrained reconstruction from noisy data," MR in Medicine.

Final Data Arrangement and Fast Math



Performance: 128 GFLOPS

Time: 1.2 minutes

Algorithms to Accelerate

Compute Q

```
for (m = 0; m < M; m++) {

  phi[m] = rPhi[m]*rPhi[m]
         + iPhi[m]*iPhi[m]

  for (n = 0; n < N; n++) {
    exp = 2*PI*(kx[m]*x[n] +
                ky[m]*y[n] +
                kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
  }

}
```

- $F^H d$ is nearly identical
- Scan data
  - M = # scan points
  - kx, ky, kz = 3D scan data
- Pixel data
  - N = # pixels
  - x, y, z = input 3D pixel data
  - Q = output pixel data
- Complexity is O(MN)
- Inner loop
  - 10 FP MUL or ADD ops
  - 2 FP trig ops
  - 10 loads

From C to CUDA: Step 1
What unit of work is assigned to each thread?


```
for (m = 0; m < M; m++) {
  phi[m] = rPhi[m]*rPhi[m] +
           iPhi[m]*iPhi[m]

  for (n = 0; n < N; n++) {
    exp = 2*PI*(kx[m]*x[n] +
                ky[m]*y[n] +
                kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
  }
}
```

From C to CUDA: Step 1
What unit of work is assigned to each thread?


```
for (m = 0; m < M; m++) {
  phi[m] = rPhi[m]*rPhi[m] +
           iPhi[m]*iPhi[m]

  for (n = 0; n < N; n++) {
    exp = 2*PI*(kx[m]*x[n] +
                ky[m]*y[n] +
                kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
  }
}
```

```
for (n = 0; n < N; n++) {
  for (m = 0; m < M; m++) {
    phi[m] = rPhi[m]*rPhi[m]
           + iPhi[m]*iPhi[m]
    exp = 2*PI*(kx[m]*x[n] +
                ky[m]*y[n] +
                kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
  }
}
```

How does loop interchange help?

From C to CUDA: Step 1
What unit of work is assigned to each thread?


```
for (n = 0; n < N; n++) {
  for (m = 0; m < M; m++) {
    phi[m] = rPhi[m]*rPhi[m]
           + iPhi[m]*iPhi[m]
    exp = 2*PI*(kx[m]*x[n] +
                ky[m]*y[n] +
                kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
  }
}
```

```
for (m = 0; m < M; m++) {
  phi[m] = rPhi[m]*rPhi[m]
         + iPhi[m]*iPhi[m]
}
for (n = 0; n < N; n++) {
  for (m = 0; m < M; m++) {
    exp = 2*PI*(kx[m]*x[n] +
                ky[m]*y[n] +
                kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
  }
}
```

How does loop fission help?

From C to CUDA: Step 1
What unit of work is assigned to each thread?

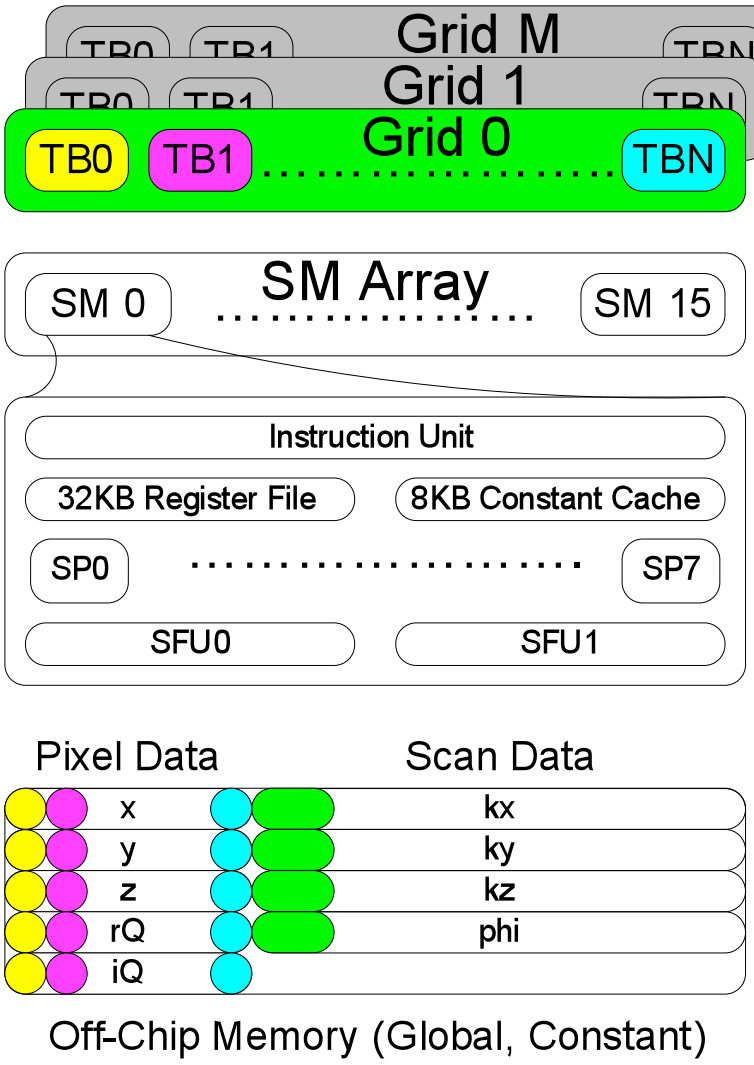

```
for (m = 0; m < M; m++) {
  phi[m] = rPhi[m]*rPhi[m]
         + iPhi[m]*iPhi[m]
}

for (n = 0; n < N; n++) {
  for (m = 0; m < M; m++) {
    exp = 2*PI*(kx[m]*x[n] +
                ky[m]*y[n] +
                kz[m]*z[n])
    rQ[n] += phi[m]*cos(exp)
    iQ[n] += phi[m]*sin(exp)
  }
}
```

- phi kernel
- Each thread computes phi at one scan point (each thread corresponds to one loop iteration)

- Q kernel
- Each thread computes Q at one pixel (each thread corresponds to one outer loop iteration)

Tiling of Scan Data



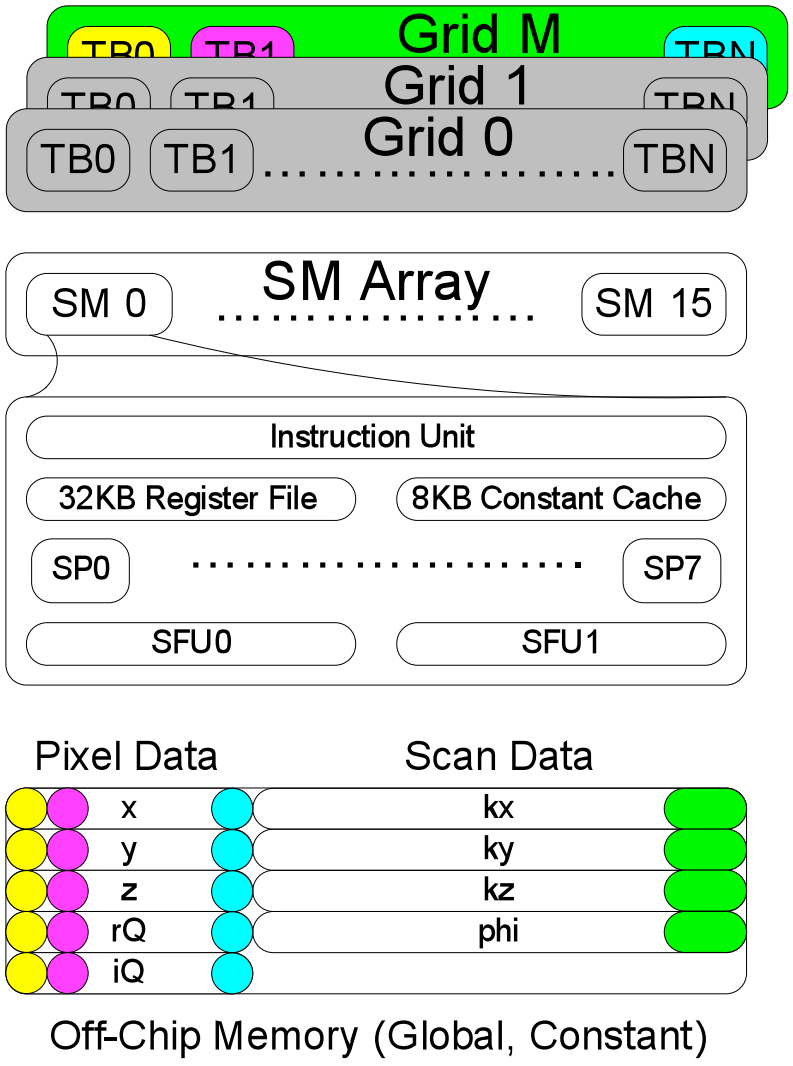
LS recon uses multiple grids

- 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++) {
  exp = 2*PI*(kx[m]*x[n] +
              ky[m]*y[n] +
              kz[m]*z[n])
  rQ[n] += phi[m]*cos(exp)
  iQ[n] += phi[m]*sin(exp)
}
```

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 = 31M/32; m < 32M/32; m++)
{
  exp = 2*PI*(kx[m]*x[n] +
              ky[m]*y[n] +
              kz[m]*z[n])
  rQ[n] += phi[m]*cos(exp)
  iQ[n] += phi[m]*sin(exp)
}
```

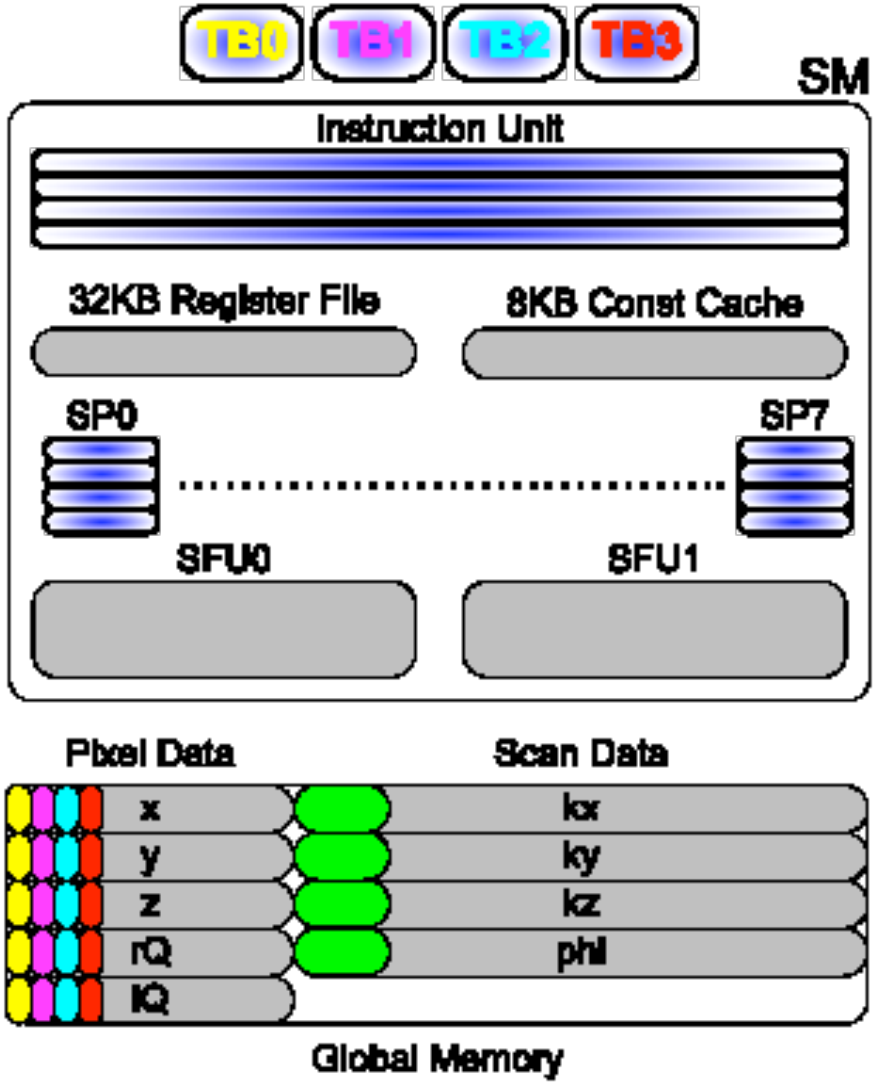
From C to CUDA: Step 2
Where are the potential bottlenecks?


```
Q(float* x,y,z,rQ,iQ,kx,ky,kz,phi,
  int startM,endM)
{
  n = blockIdx.x*TPB + threadIdx.x
  for (m = startM; m < endM; m++) {
    exp = 2*PI*(kx[m]*x[n]
              + ky[m]*y[n]
              + kz[m]*z[n])
    rQ[n] += phi[m] * cos(exp)
    iQ[n] += phi[m] * sin(exp)
  }
}
```

Bottlenecks

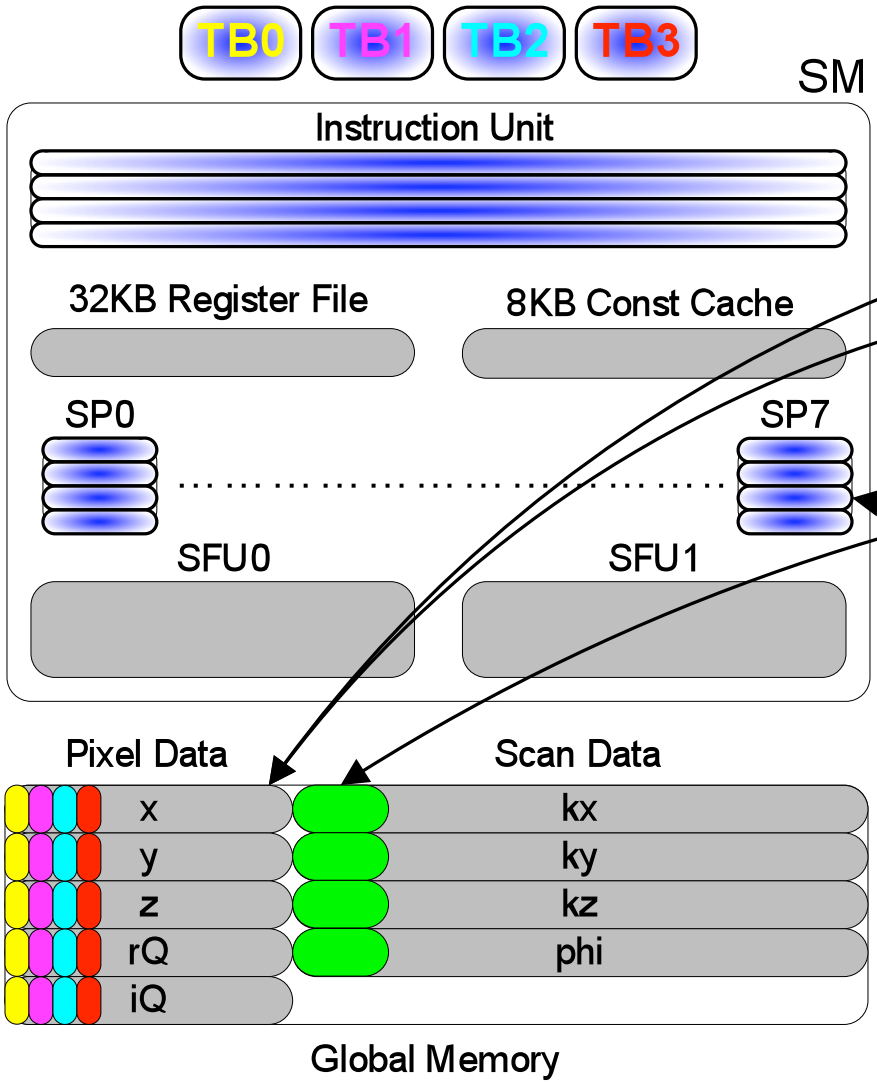
- Memory BW
- Trig ops
- Overheads (branches, addr calcs)

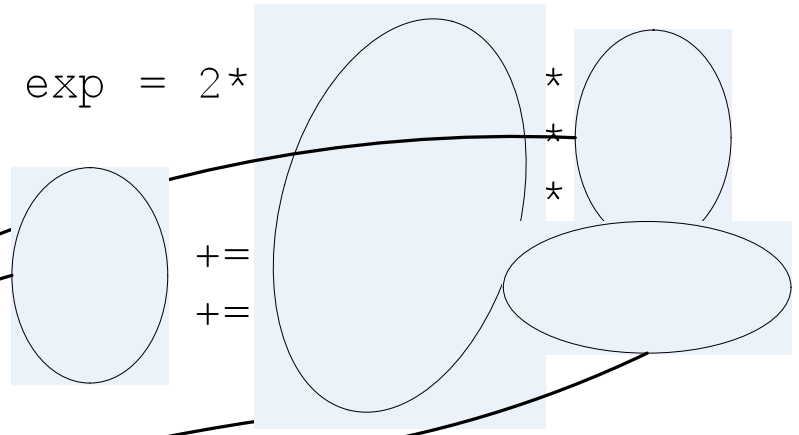
Step 3: Overcoming bottlenecks



- LS recon on CPU (SP)
  - Q: 45 hours, 0.5 GFLOPS
  - $F^H d$: 7 hours, 0.7 GFLOPS
- Counting each trig op as 1 FLOP

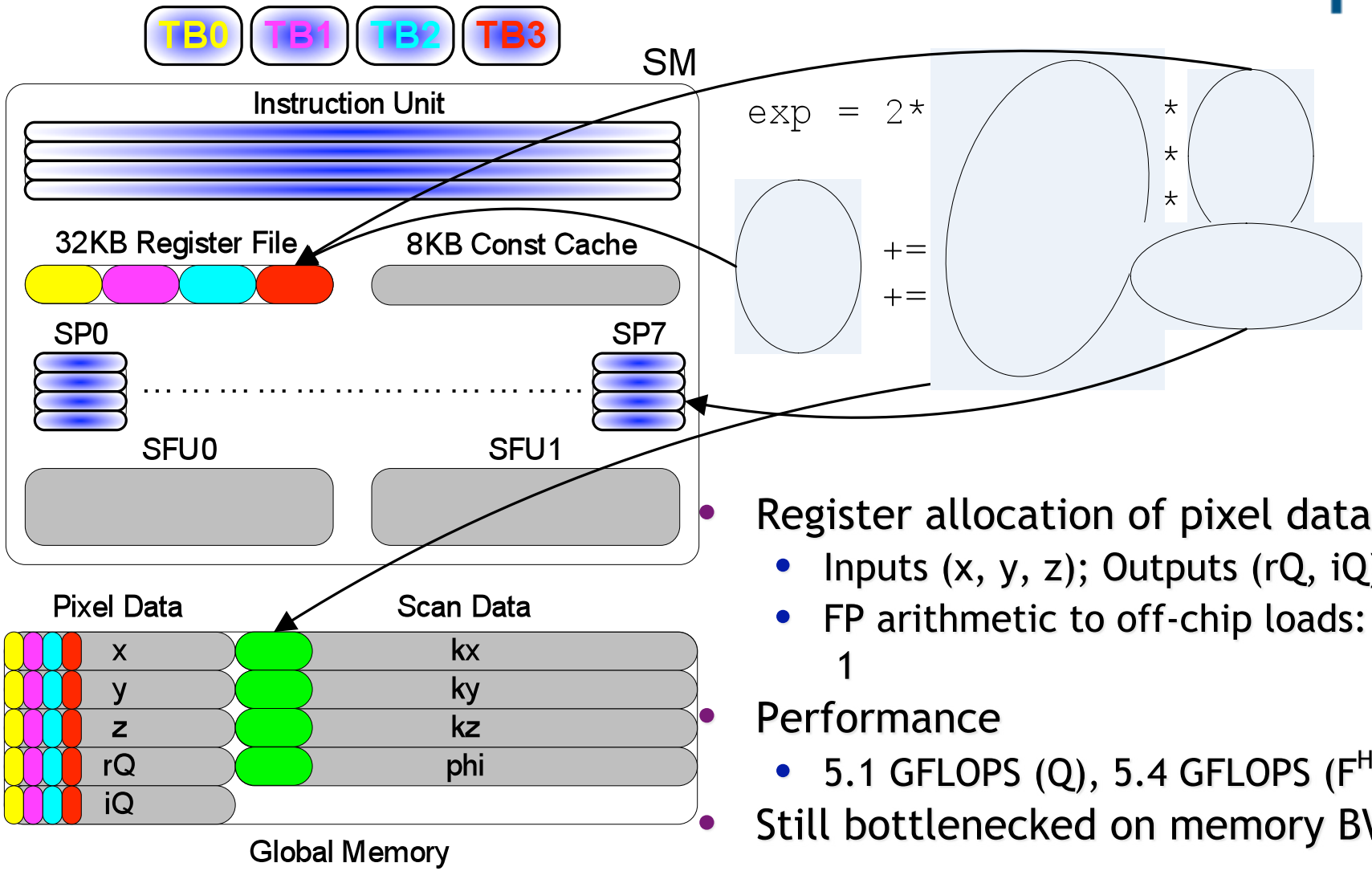
Step 3: Overcoming Bottlenecks (Mem BW)





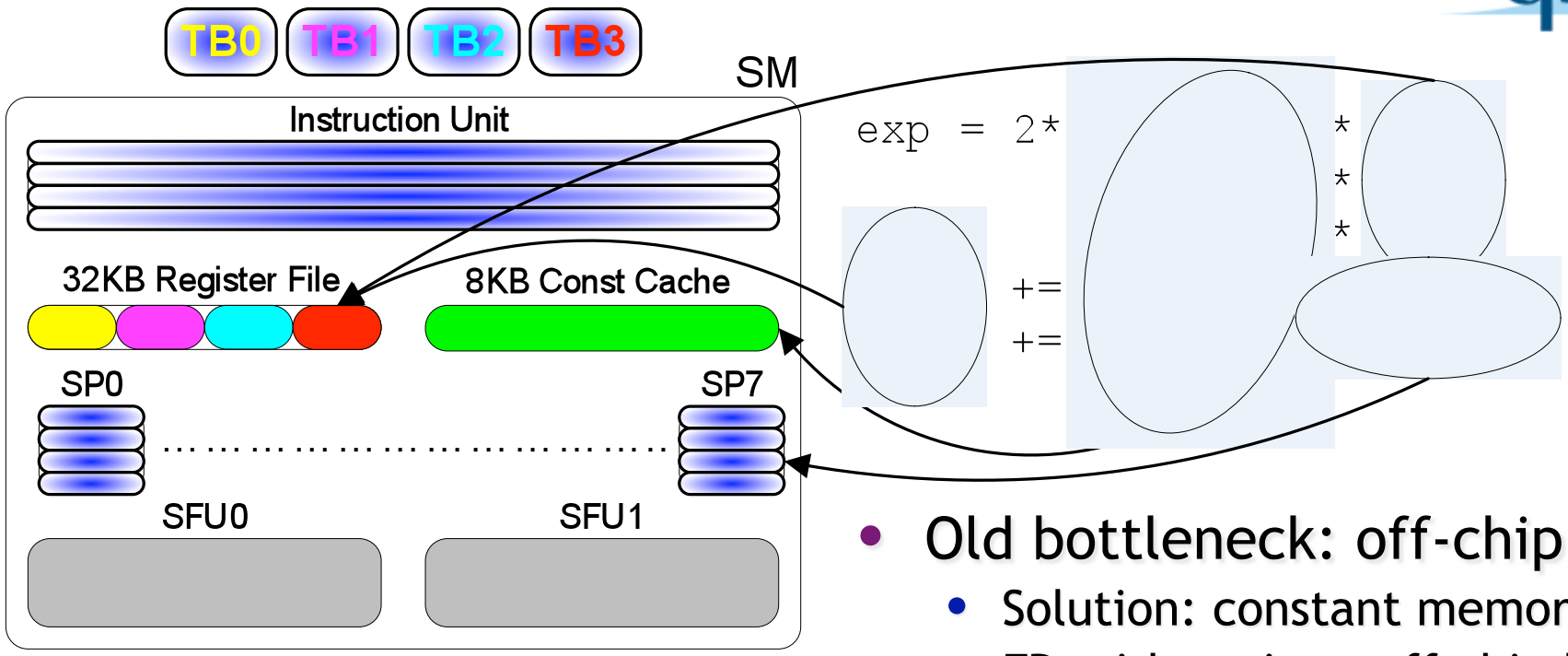
- Register allocate pixel data
  - Inputs (x, y, z); Outputs (rQ, iQ)
- Exploit temporal and spatial locality in access to scan data
  - Constant memory + constant caches
  - Shared memory

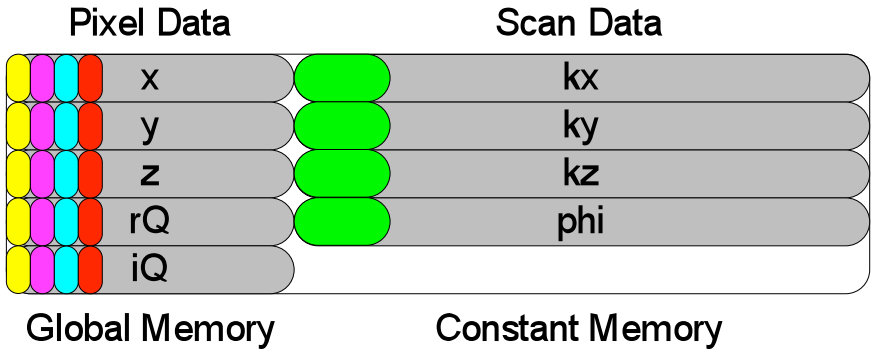
Step 3: Overcoming Bottlenecks (Mem BW)



- Register allocation of pixel data
  - Inputs (x, y, z); Outputs (rQ, iQ)
  - FP arithmetic to off-chip loads: 2 to 1
- Performance
  - 5.1 GFLOPS (Q), 5.4 GFLOPS ($F^H d$)
- Still bottlenecked on memory BW

Step 3: Overcoming Bottlenecks (Mem BW)



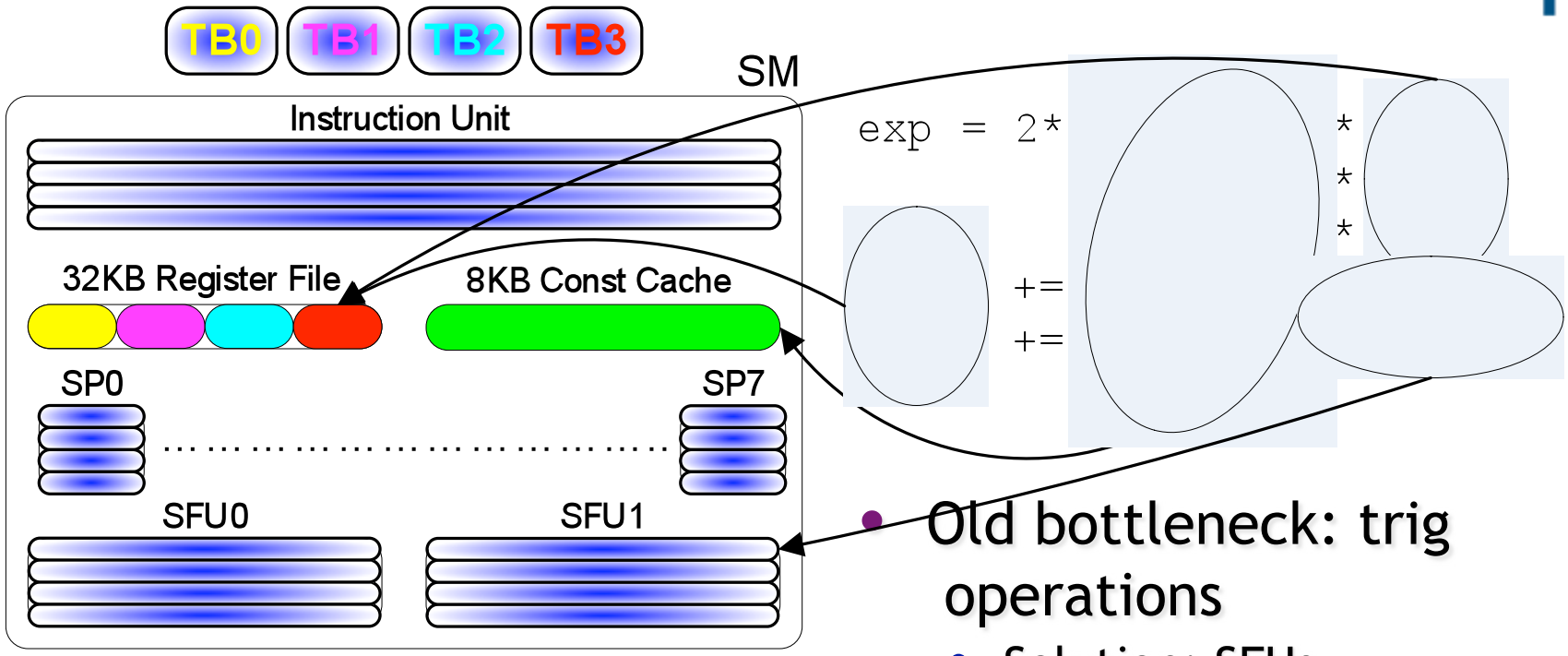


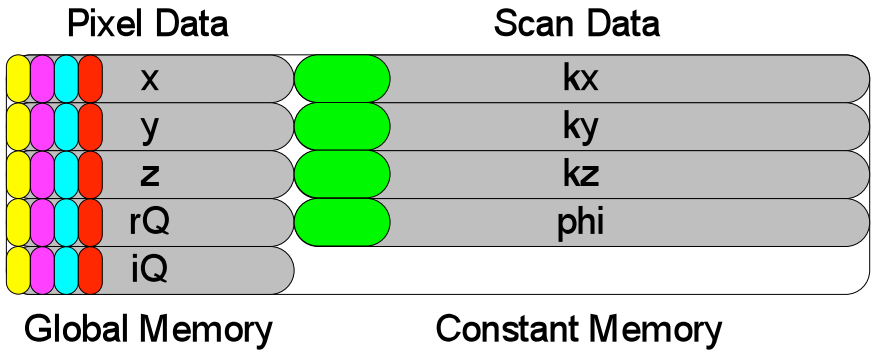
- Old bottleneck: off-chip BW
  - Solution: constant memory
  - FP arithmetic to off-chip loads: 284 to 1
- Performance
  - 18.6 GFLOPS (Q), 22.8 GFLOPS ($F^H d$)
- New bottleneck: trig operations

Sidebar: Estimating Off-Chip Loads with Const Cache

- How can we approximate the number of off-chip loads when using the constant caches?
- Given: 128 tpb, 4 blocks per SM, 256 scan points per grid
- **Assume no evictions due to cache conflicts**
- 7 accesses to global memory per thread (x, y, z, rQ x 2, iQ x 2)
  - 4 blocks/SM * 128 threads/block * 7 accesses/thread = 3,584 global mem accesses
- 4 accesses to constant memory per scan point (kx, ky, kz, phi)
  - 256 scan points * 4 loads/point = 1,024 constant mem accesses
- Total off-chip memory accesses = 3,584 + 1,024 = 4,608
- Total FP arithmetic ops = 4 blocks/SM * 128 threads/block * 256 iters /thread * 10 ops/iter = 1,310,720
- FP arithmetic to off-chip loads: 284 to 1

Step 3: Overcoming Bottlenecks (Trig)





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

Sidebar: Effects of Approximations

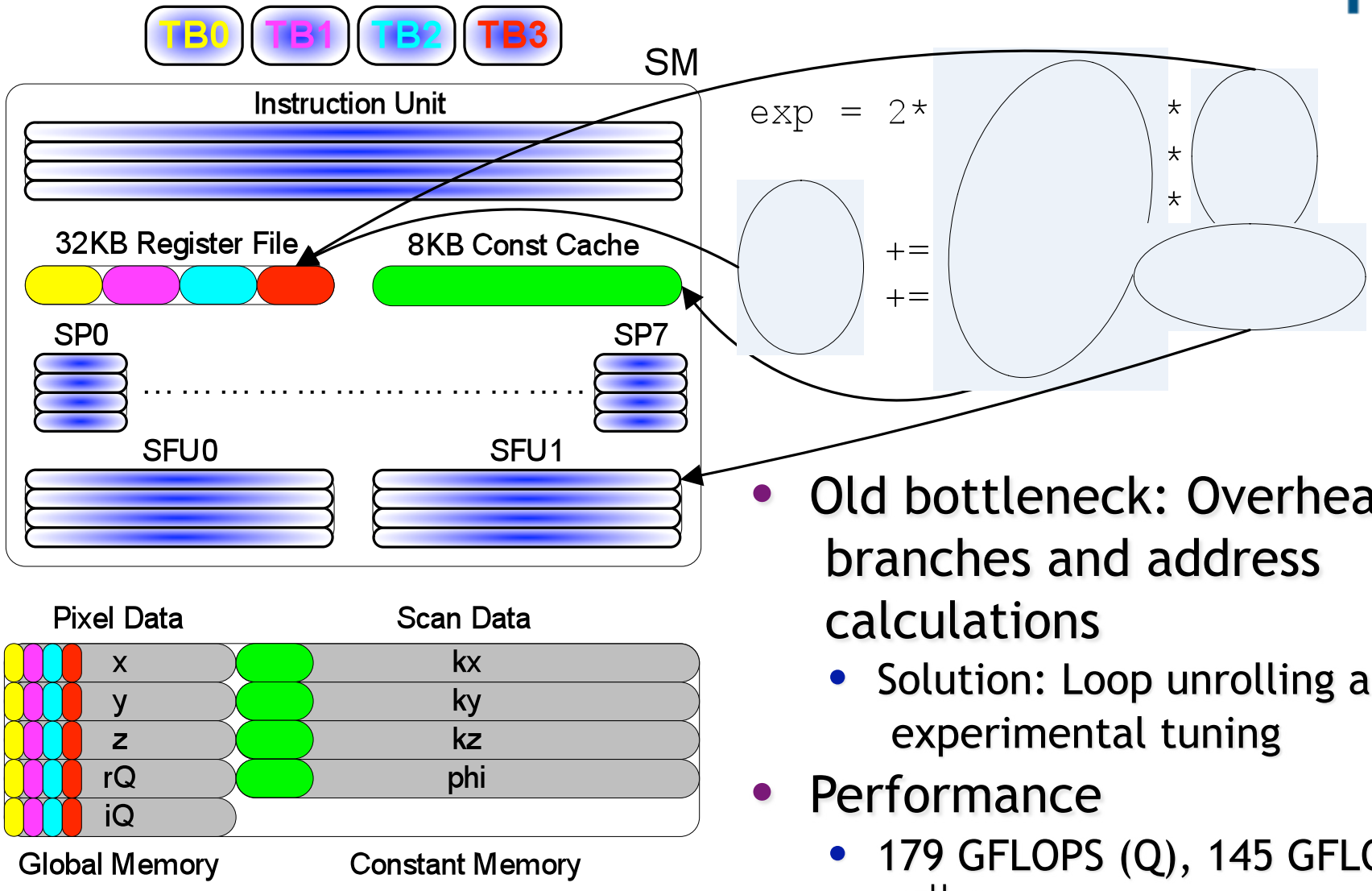
- Avoid temptation to measure only absolute error ($I_0 - I$)
  - Can be deceptively large or small
- Metrics
  - PSNR: Peak signal-to-noise ratio
  - SNR: Signal-to-noise ratio
- Avoid temptation to consider only the error in the computed value
  - Some apps are resistant to approximations; others are 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).

Step 3: Overcoming Bottlenecks (Overheads)

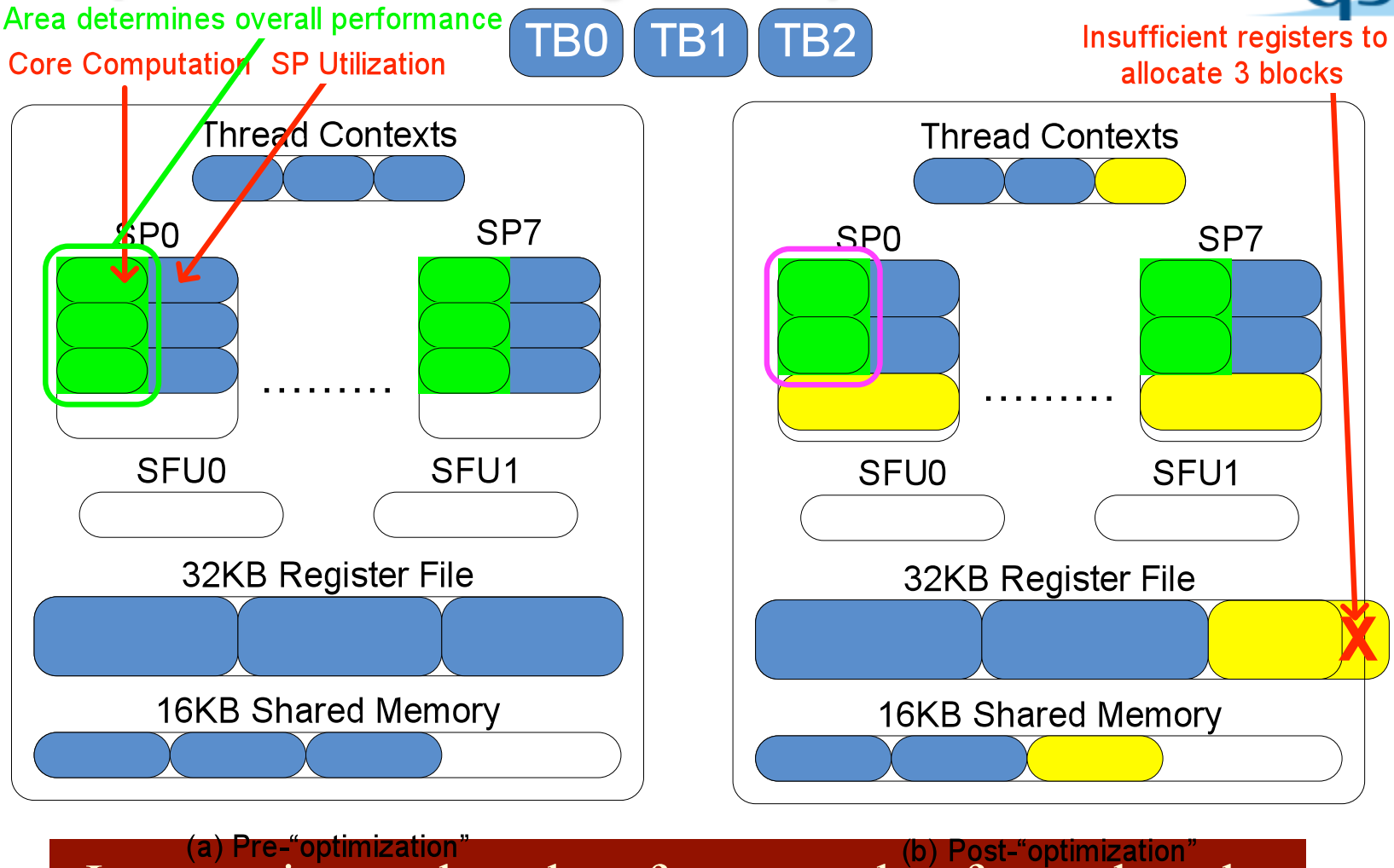


- Old bottleneck: Overhead of branches and address calculations
  - Solution: Loop unrolling and experimental tuning
- Performance
  - 179 GFLOPS (Q), 145 GFLOPS ($F^H d$)

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

Experimental Tuning: Example

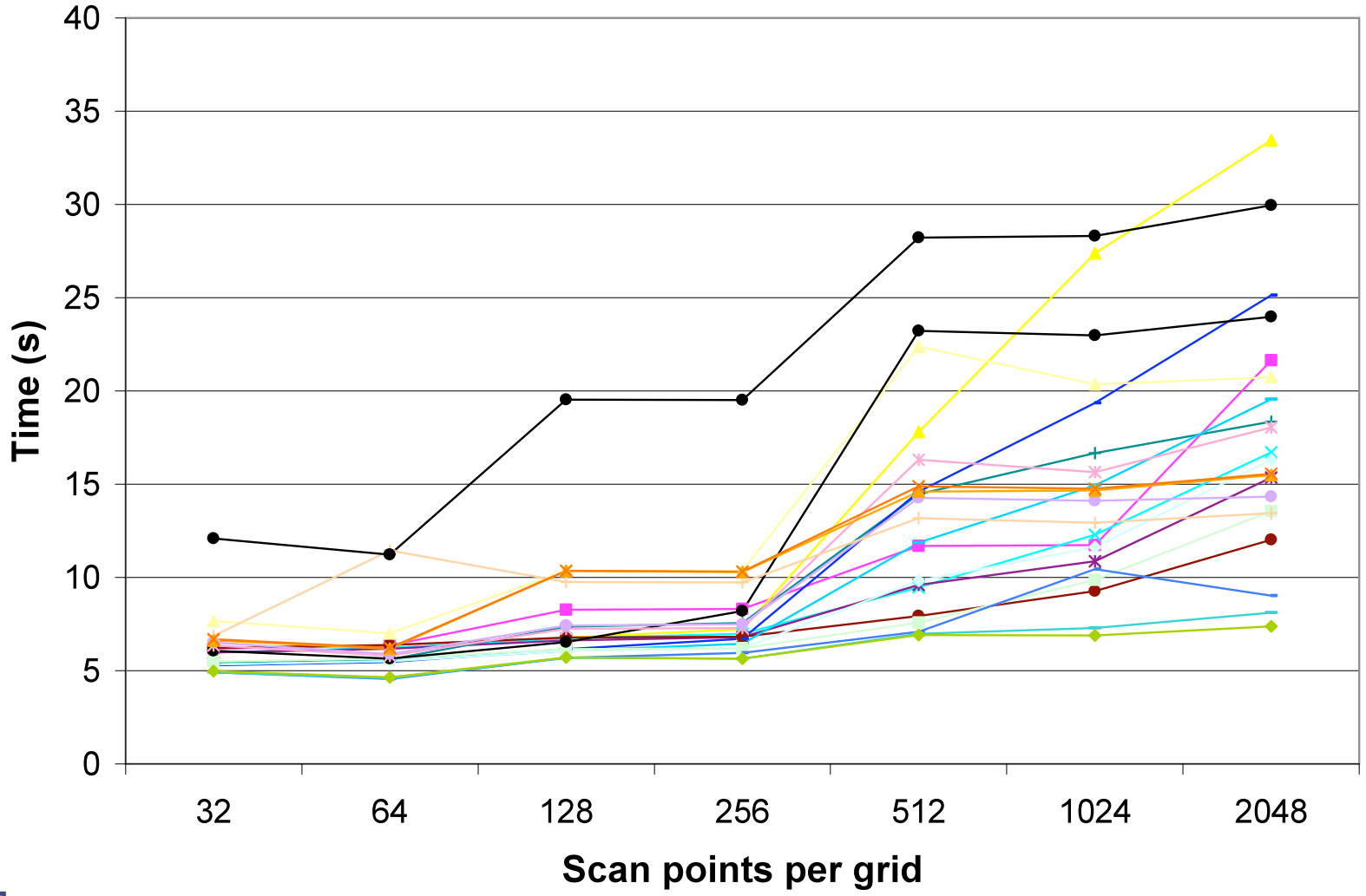


(a) Pre-"optimization"

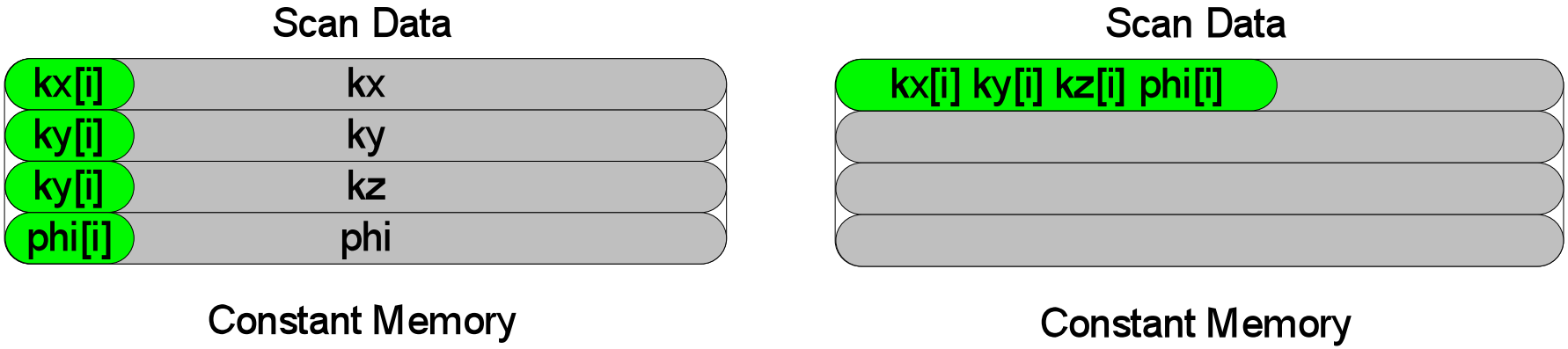
(b) Post-"optimization"

Increase in per-thread performance, but fewer threads:
Lower overall performance

Experimental Tuning: Scan Points Per Grid

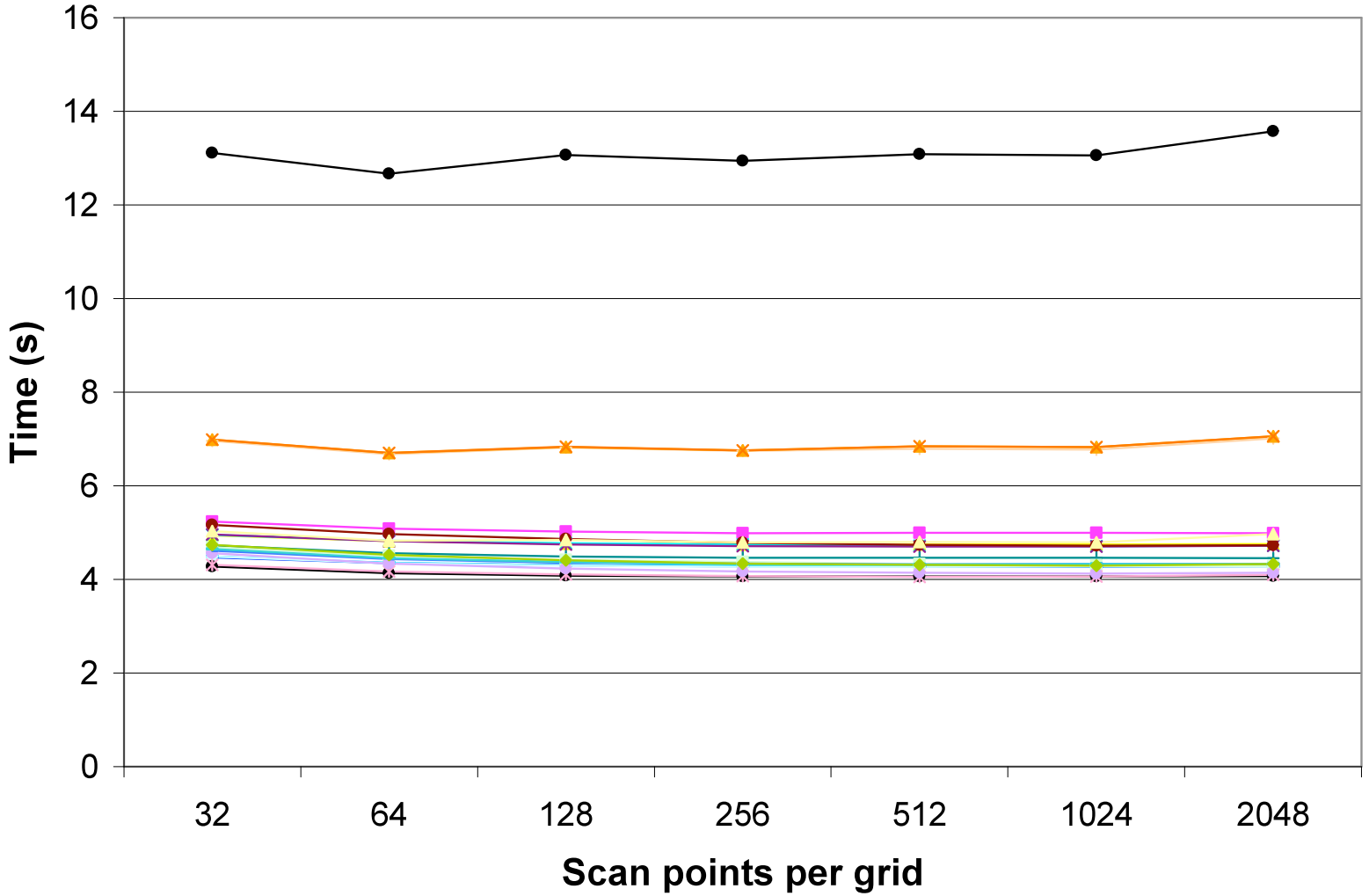


Sidebar: Cache-Conscious Data Layout

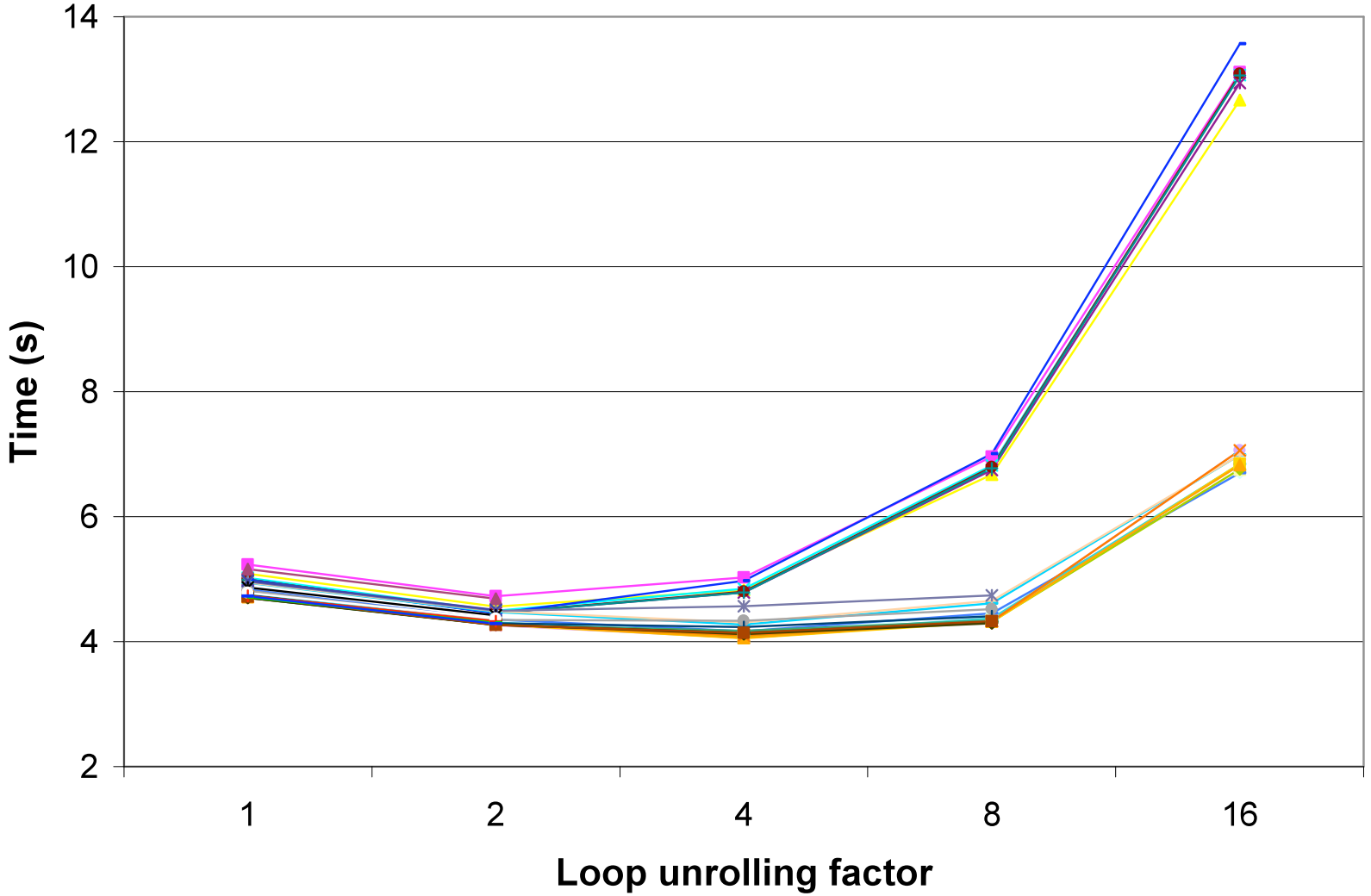


- kx, ky, kz, and phi components of same scan point have spatial and temporal locality
  - Prefetching
  - Caching
- Old layout does not fully leverage that locality
- New layout does fully leverage that locality

Experimental Tuning: Scan Points Per Grid (Improved Data Layout)



Experimental Tuning: Loop Unrolling Factor



Sidebar: Optimizing the CPU Implementation

- Optimizing the CPU implementation of your application is very important
  - Often, the transformations that increase performance on CPU also increase performance on GPU (and vice-versa)
  - The research community won't take your results seriously if your baseline is crippled
- Useful optimizations
  - Data tiling
  - SIMD vectorization (SSE)
  - Fast math libraries (AMD, Intel)
  - Classical optimizations (loop unrolling, etc)
- Intel compiler (icc, icpc)

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

8X

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

Results must be validated by domain experts.

True

Gridded

CPU.DP

CPU.SP

GPU.Tune

ILLINOIS

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

GSRC

Gigascale Systems Research Center

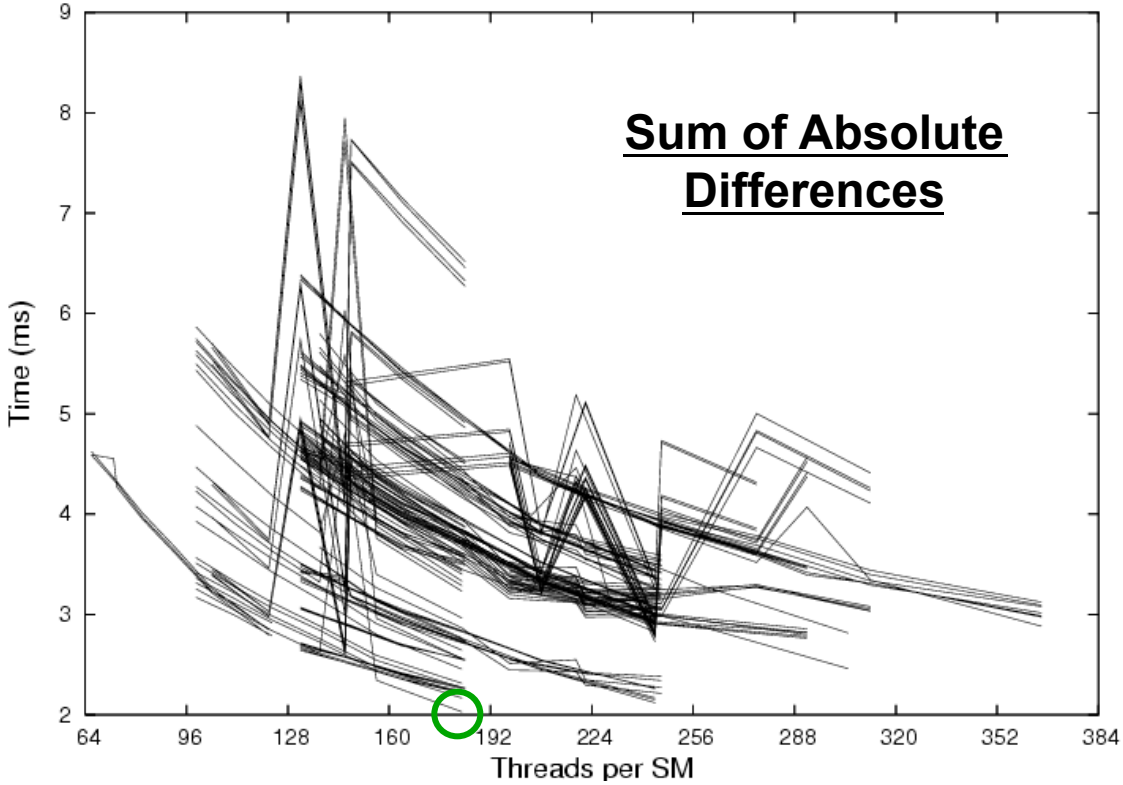
Now, the coming battles.

"There is always hope."

– In the eve of the Battle of the Helms Deep

Battle #1 - Tools need to do more heavy lifting!

- Programmers are doing too much heavy lifting
- Too many memory organizational details are exposed to the programmers



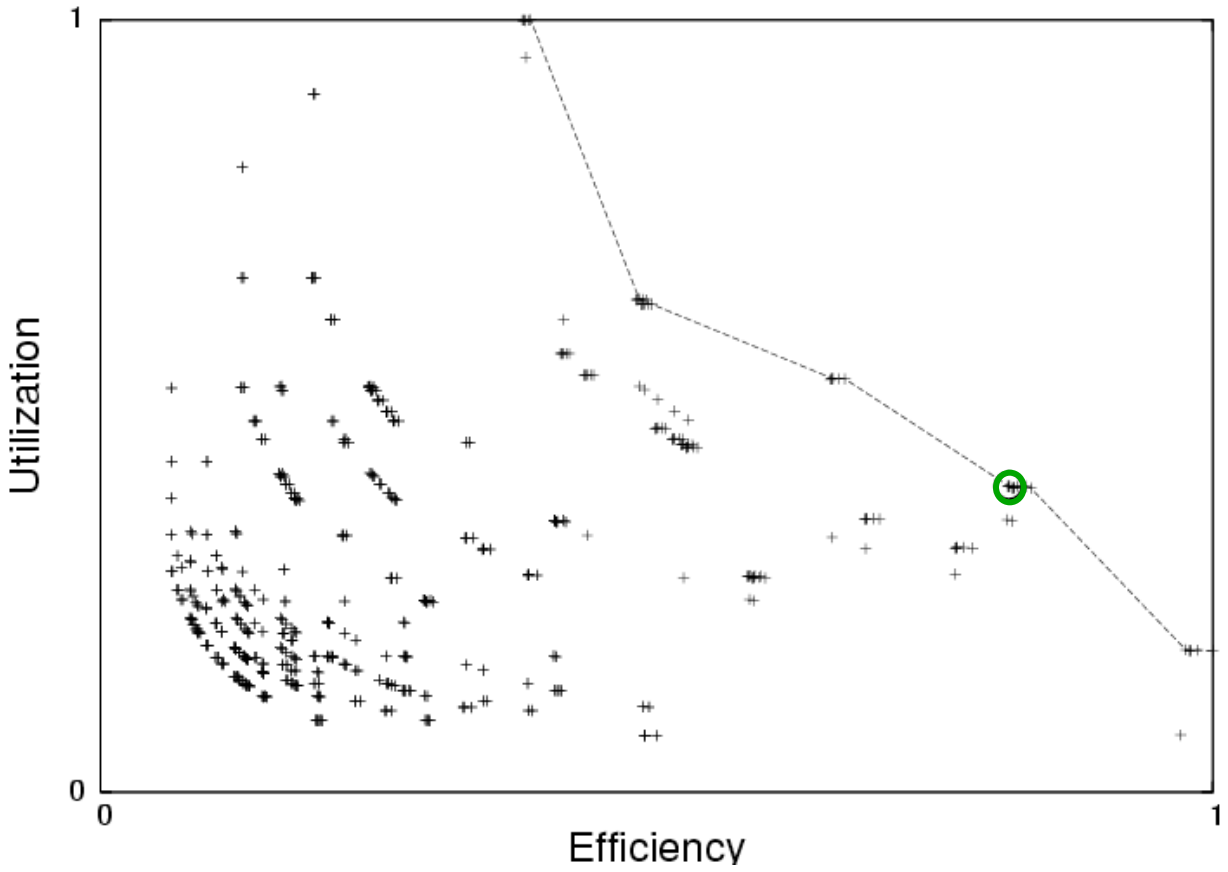
- Search spaces can be huge! Exhaustive search with smaller-than usual data sets still takes significant time.

S. Ryoo, et al, "Program Optimization Space Pruning for a Multithreaded GPU, ACM /IEEE CGO, April 2008.

Analytical Models Help Reduce Search Space.

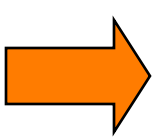
Sum of Absolute Differences

By selecting only Pareto-optimal points, we pruned the search space by 98% and still found the optimal configuration



S. Ryoo, et al, "Program Optimization Space Pruning for a Multithreaded GPU, ACM /IEEE CGO, April 2008.

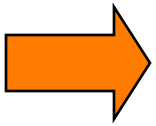
Implicitly parallel programming with data structure and function property annotations to enable auto parallelization



CUDA-auto

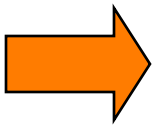
GSRC

Locality annotation programming to eliminate need for explicit management of memory types and data transfers



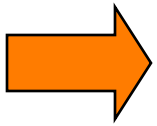
CUDA-lite

Parameterized CUDA programming using auto-tuning and optimization space pruning



CUDA-tune

1st generation CUDA programming with explicit, hardwired thread organizations and explicit management of memory types and data transfers



MCUDA/ OpenMP

NVIDIA SDK 1.1

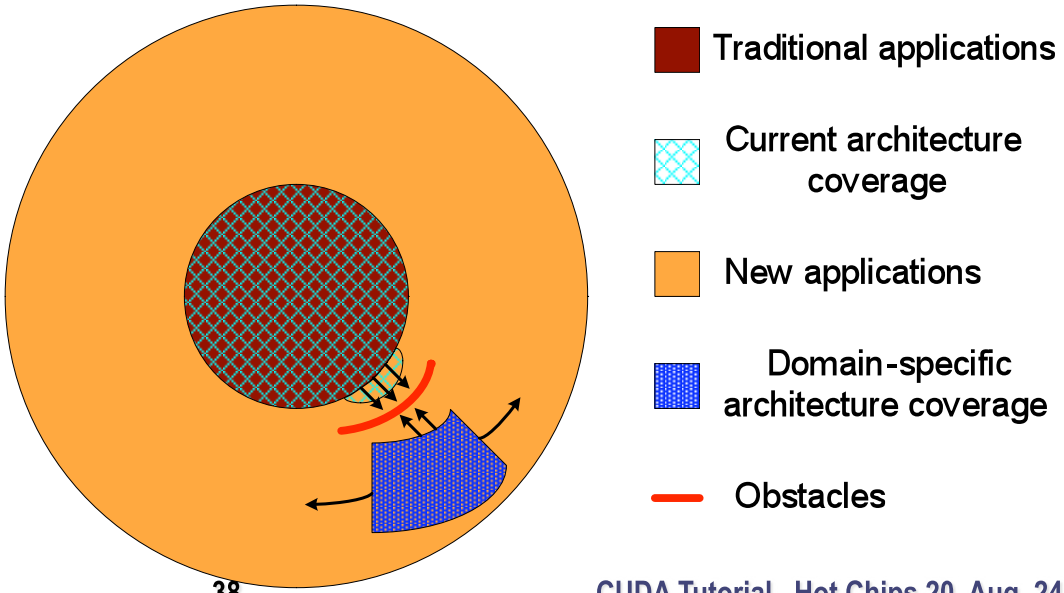
IA multi-core & Larrabe

NVIDIA GPU

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

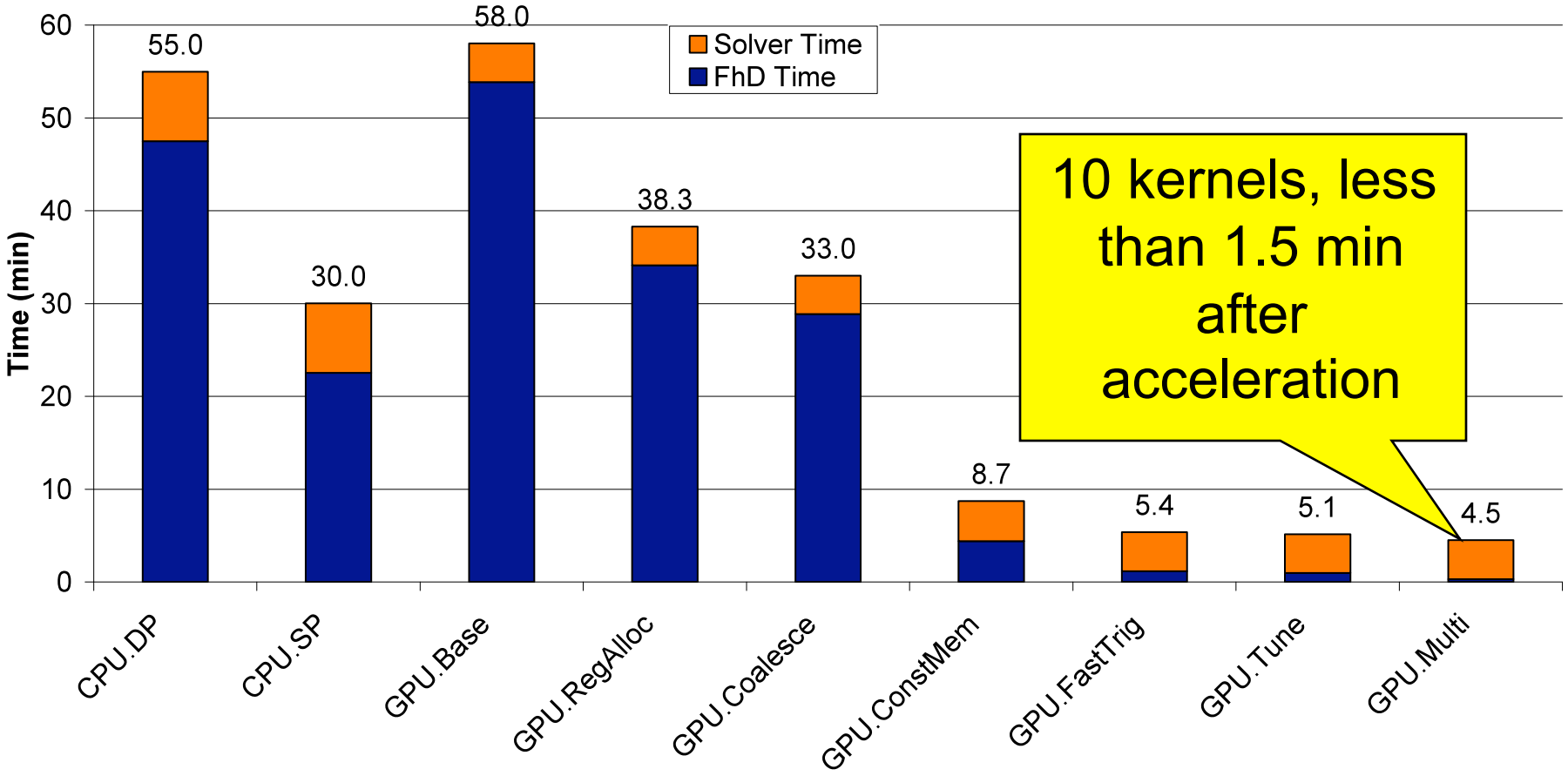
Battle #2 – Reaching deeper into apps.

- Many data parallel apps have a small number of simple, dominating components
  - Low hanging fruit for parallel computing (meat)
- Small computation components often dominate after the low hanging fruits are picked
  - Usually much more difficult to parallelize (pit)



ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Performance of Advanced MRI Reconstruction



S.S. Stone, et al, "Accelerating Advanced MRI Reconstruction using GPUs," ACM Computing Frontier Conference 2008, Italy, May 2008.

What does it take to reach deeper?

- MRI: Launching multiple kernels
  - 3D FFT formulated as multiple 2D FFTs.
  - Multiple kernel types beneficial in some apps
- Global Synchronization
  - Some apps require global synch at time step boundaries
- Atomic memory operations
  - Shared memory, device global memory
- Less tedious tuning process for kernels
  - Developers run out of time!

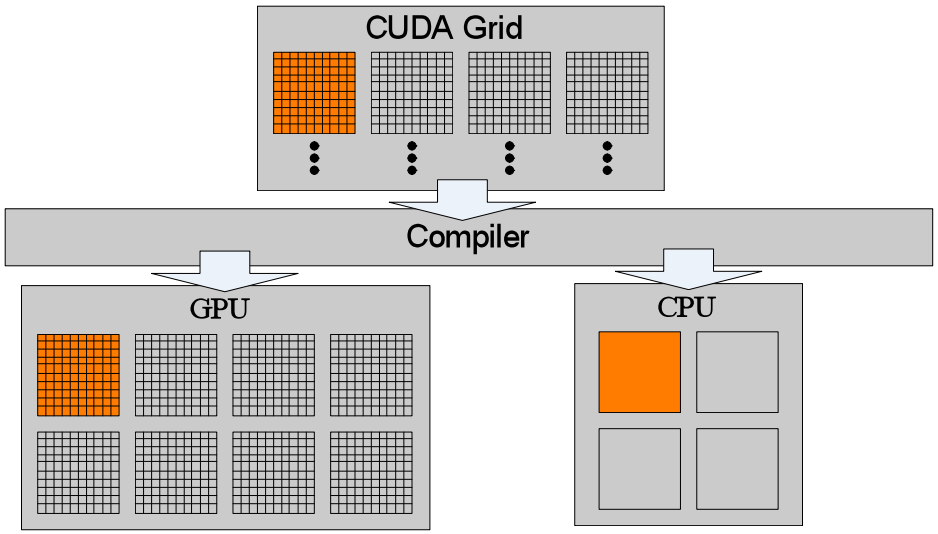
Battle #3: There is room for only one model.

- App developers cannot afford to write their apps in multiple programming models
  - It is all about programmer acceptance!

- Code based on the winning model will need to work well on both GPUs and Multi-cores
  - CUDA kernels currently only work on NVIDIA GPUs
  - How about ATI GPUs, Intel Multi-cores, and AMD Multi-cores?

MCUDA: Thread Blocks to CPU Threads

- A single GPU thread is too small for a CPU Thread
  - CUDA emulation does this and performs poorly
- CPU cores designed for ILP, SIMD
  - Optimizing compilers work well with iterative loops
- Turn GPU thread blocks from CUDA into iterative CPU loops



Key Issue: Synchronization

- Suspend and Wakeup
  - Move on to other threads
  - Begin again after all hit barrier

```
Matrixmul(A[ ], B[ ], C[ ])
{
   __shared__ Asub[ ][ ], Bsub[ ][ ];
   int a,b,c;
   float Csub;
   int k;
   …
   for(…)
    {

        Asub[tx][ty] = A[a];
         Bsub[tx][ty] = B[b];

        __syncthreads();


        for( k = 0; k < blockDim.x; k++ )
             Csub += Asub[ty][k] + Bsub[k][tx];

        __syncthreads();

    }
   …
}
```

Synchronization solution

- Loop fission
  - Break the loop into multiple smaller loops according to syncthreads()
  - Not always possible with synchthreads() in control flow

```
Matrixmul(A[ ], B[ ], C[ ])
{
   __shared__ Asub[ ][ ], Bsub[ ][ ];
   int a,b,c;
   float Csub;
   int k;
   …
   for(…)
    {
        for(ty=0; ty < blockDim.y; ty++)
          for(tx=0; tx < blockDim.x; tx++)
        {
          Asub[tx][ty] = A[a];
          Bsub[tx][ty] = B[b];
        }

        for(ty=0; ty < blockDim.y; ty++)
          for(tx=0; tx < blockDim.x; tx++)
        {
          for( k = 0; k < blockDim.x; k++ )
             Csub += Asub[ty][k] + Bsub[k][tx];
        }

    }
   …
}
```

Bigger Picture Performance Results
(reduced Sample volume)

- Consistent speed-up over hand-tuned single-thread code
- Best optimizations for GPU and CPU not always the same

Application	C on CPU Time	CUDA on CPU Time	Speedup*	CUDA on G80 Time
MRI-FHD	**~1000s**	**230s**	**~4x**	**8.5s**
CP	180s	45s	4x	.28s
SAD	42.5ms	25.6ms	1.66x	4.75ms
MM (4Kx4K)	7.84s**	10.1s	3.69x	1.12s

*Over hand-optimized CPU
**Intel MKL, multi-core execution

To Learn More

- UIUC ECE498AL – Programming Massively Parallel Processors (http://courses.ece.uiuc.edu/ece498/al/)
  - David Kirk (NVIDIA) and Wen-mei Hwu (UIUC) co-instructors
  - CUDA programming, GPU computing, lab exercises, and projects
  - Lecture slides and voice recordings
- More than 500 students worldwide follow the course each semester.

Thank you! Any questions?