CS 677: Parallel Programming for Many-core Processors
Lecture 5

Instructor: Philippos Mordohai

Webpage: www.cs.stevens.edu/~mordohai

E-mail: Philippos.Mordohai@stevens.edu

Logistics

- Midterm: March 22
- Project proposal presentations: March 8
  – Have to be approved by me by March 3

Project Proposal

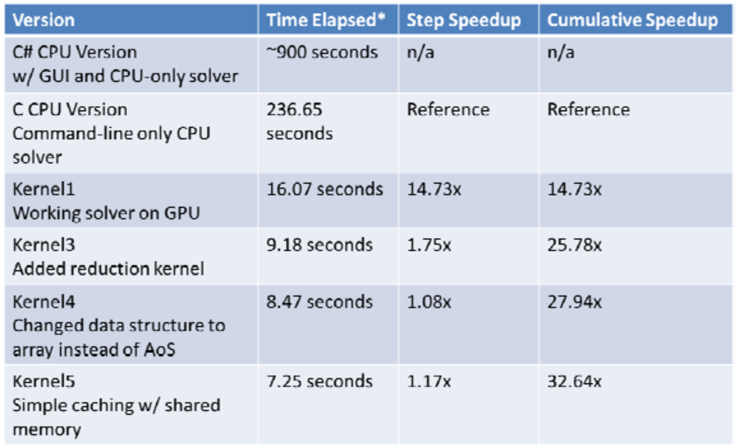
- Problem description
  - What is the computation and why is it important?
  - Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page
- Suitability for GPU acceleration
  - Amdahl's Law: describe the inherent parallelism. Argue that it is close to 100% of computation.
  - Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
  - Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.
- Intellectual Challenges
  - Generally, what makes this computation worthy of a project?
  - Point to any difficulties you anticipate at present in achieving high speedup

Some Ideas

- k-means
- Perceptron
- Boosting
  - General
  - Face detector (group of 2)
- Mean Shift
- Normal estimation for 3D point clouds

More Ideas

- Look for parallelizable problems in:
  - Image processing
  - Cryptanalysis
  - Graphics
    - GPU Gems
  - Nearest neighbor search



Version	Time Elapsed*	Step Speedup	Cumulative Speedup
C# CPU Version w/ GUI and CPU-only solver	~900 seconds	n/a	n/a
C CPU Version Command-line only CPU solver	236.65 seconds	Reference	Reference
Kernel1 Working solver on GPU	16.07 seconds	14.73x	14.73x
Kernel3 Added reduction kernel	9.18 seconds	1.75x	25.78x
Kernel4 Changed data structure to array instead of AoS	8.47 seconds	1.08x	27.94x
Kernel5 Simple caching w/ shared memory	7.25 seconds	1.17x	32.64x

GPU: Shared Memory
512 Zombies
Average FPS: 45.9

Even More…

- Particle simulations
- Financial analysis
- MCMC
- Games/puzzles
  - Mastermind example

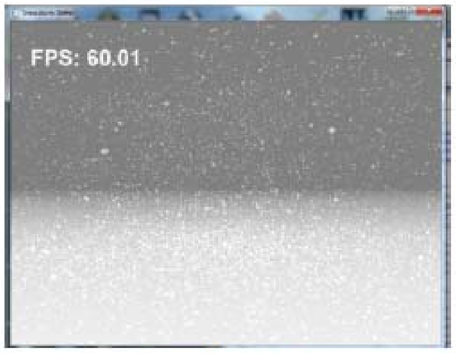
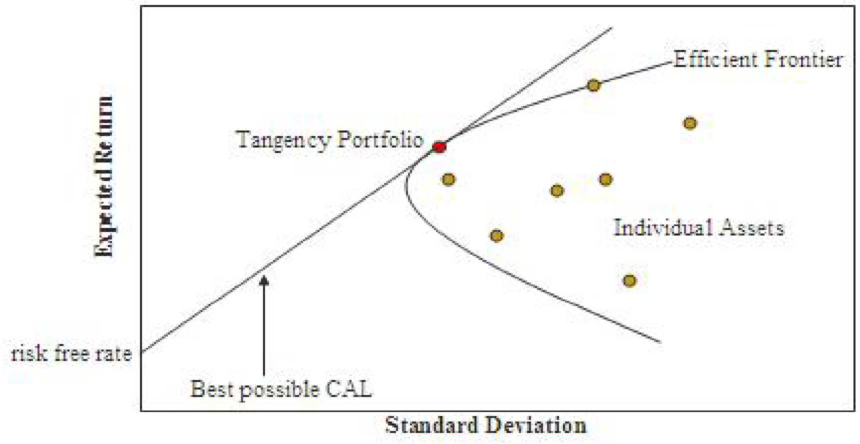
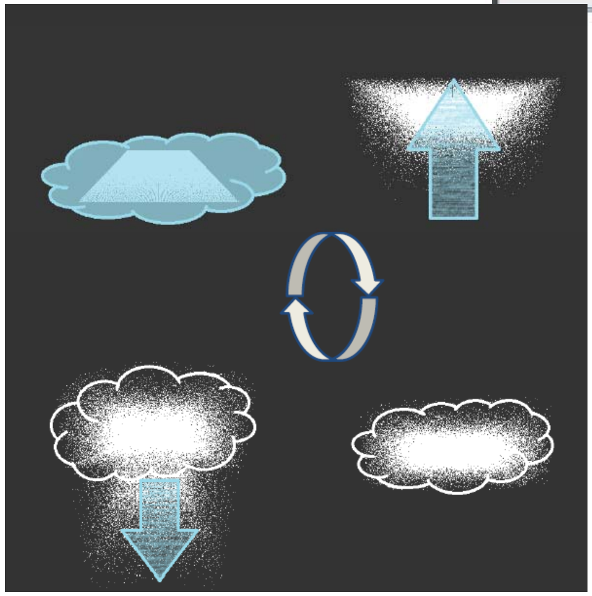


Figure 3: Snowfall

Figure 4: Interactive Snow





k-means

- See also http://www.cs.stevens.edu/~mordohai/classes/cs559_f10.html
  - Notes 13

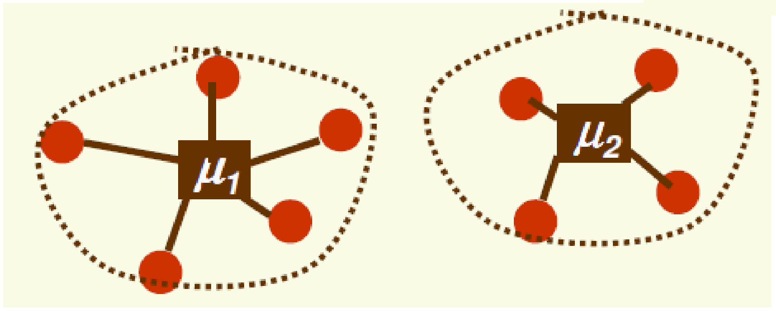
SSE Criterion Function

- Let **n**$_i$ be the number of samples, then the mean is:

$$\mu_i = \frac{1}{n_i}\sum_{x \in D_i} x$$

- The sum-of-squared errors criterion function (to minimize) is:

$$J_{SSE} = \sum_{i=1}^{c}\sum_{x \in D_i} || x - \mu_i ||^2$$



- Note that the number of clusters, **c**, is fixed

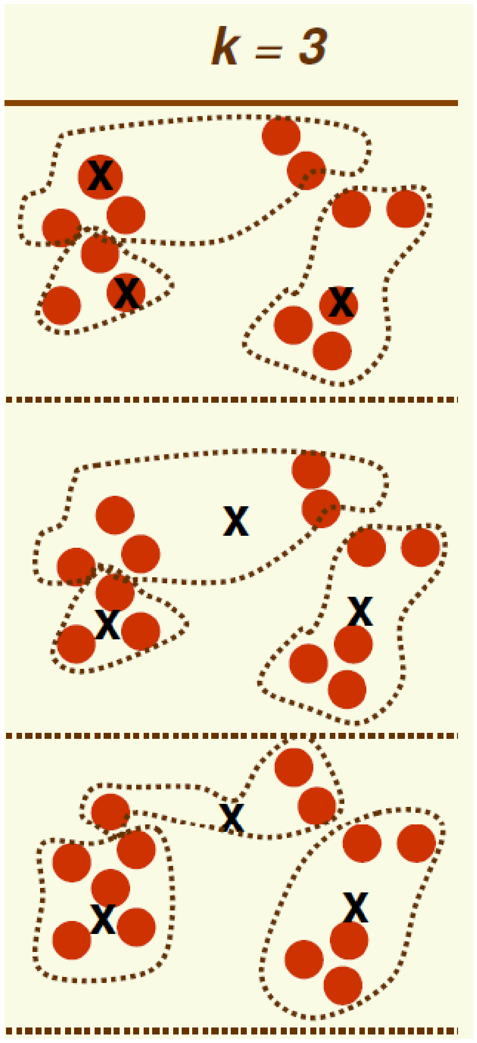
K-means Clustering

1. Initialize
   - Pick *k* cluster centers arbitrarily
   - Assign each example to closest center

2. Compute sample means for each cluster

3. Reassign all samples to the closest mean

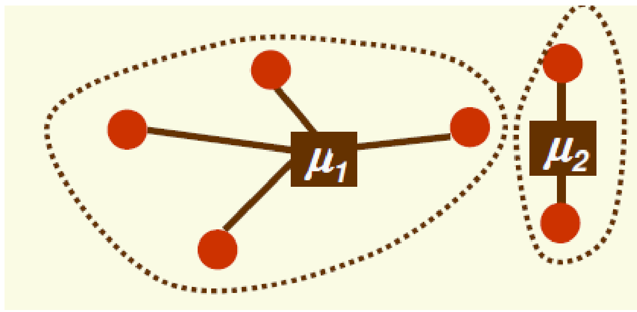
4. If clusters changed at step 3, go to step 2



K-means Clustering

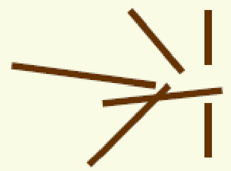
Consider steps 2 and 3 of the algorithm

2. compute sample means for each cluster

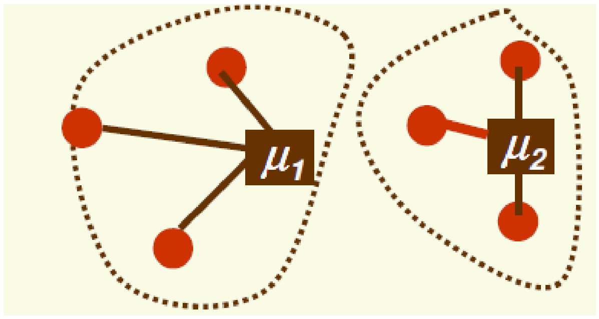


$$J_{SSE} = \sum_{i=1}^{k} \sum_{x \in D_i} \| \mathbf{x} - \mu_i \|^2$$

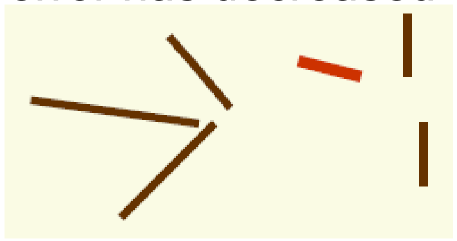
= sum of



3. reassign all samples to the closest mean

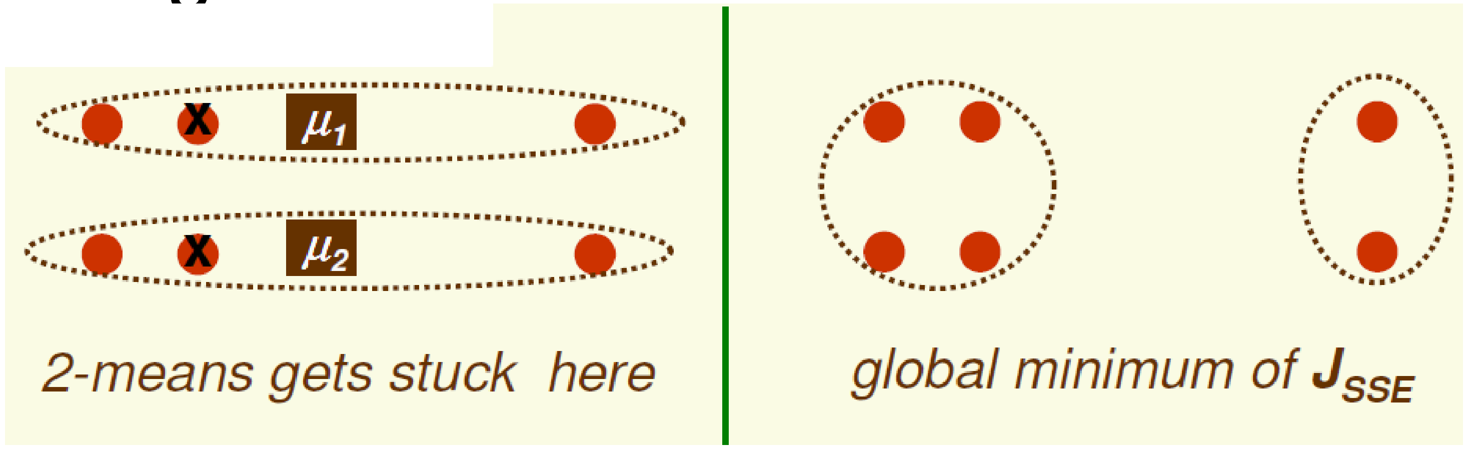


If we represent clusters by their old means, the error has decreased



K-means Clustering

- We can prove that by repeating steps 2 and 3, the objective function is reduced
- Thus k-means converges after a finite number of iterations of steps 2 and 3
- However k-means is not guaranteed to find a global minimum



K-means Clustering

- Finding the optimum of J_{SSE} is NP-hard
- In practice, k-means clustering usually performs well
- To avoid local minima, in practice we randomly re-initialize it several times

Perceptron

- See also http://www.cs.stevens.edu/~mordohai/classes/cs559_f14.html
  - Notes 13

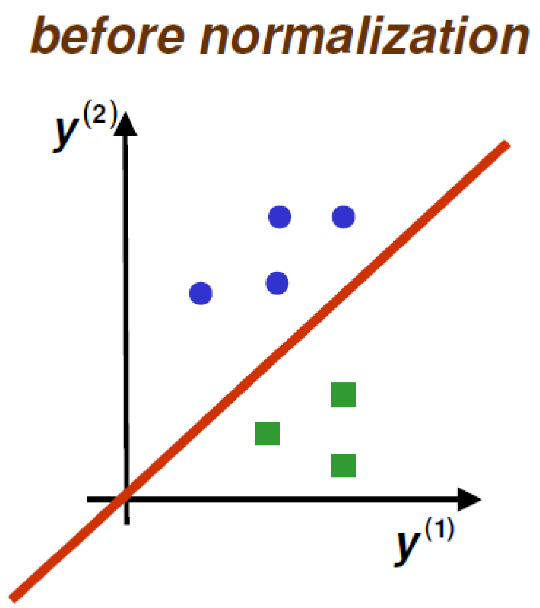
The Problem

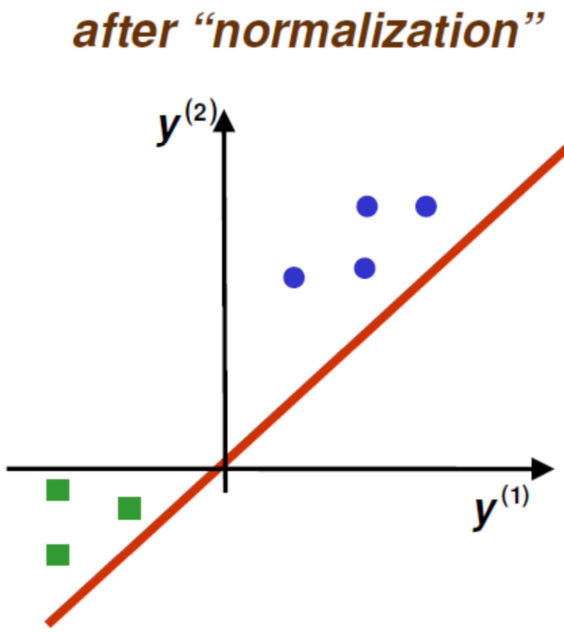
- Assume we have 2 classes
  - Samples: $y_1, \ldots, y_n$, some in class 1, some in class 2
- Use samples to determine weights a in the discriminant function $g(y) = a^t y$
- We want to minimize the training error (the number of misclassified samples $y_1, \ldots, y_n$)
- If: $g(y_i) > 0 \Rightarrow y_i$ ***classified as*** c_1
  $g(y_i) < 0 \Rightarrow y_i$ ***classified as*** c_2
- Thus training error is 0 if $\begin{cases} g(y_i) > 0 & \forall y_i \in c_1 \\ g(y_i) < 0 & \forall y_i \in c_2 \end{cases}$

“Normalization”

- Thus training error is 0 if: $\begin{cases} a^t y_i > 0 & \forall y_i \in c_1 \\ a^t y_i < 0 & \forall y_i \in c_2 \end{cases}$
- Equivalently, training error is 0 if: $\begin{cases} a^t y_i > 0 & \forall y_i \in c_1 \\ a^t(-y_i) > 0 & \forall y_i \in c_2 \end{cases}$
- This suggests “normalization” (a.k.a. reflection):
  1. Replace all examples from class 2 by:
  $$y_i \to -y_i \qquad \forall y_i \in c_2$$
  2. Seek weight vector ***a*** such that
  $$a^t y_i > 0 \qquad \forall y_i$$
  - If such ***a*** exists, it is called a separating or solution vector
  - Original samples $x_1, \ldots, x_n$ can indeed be separated by a line

Normalization

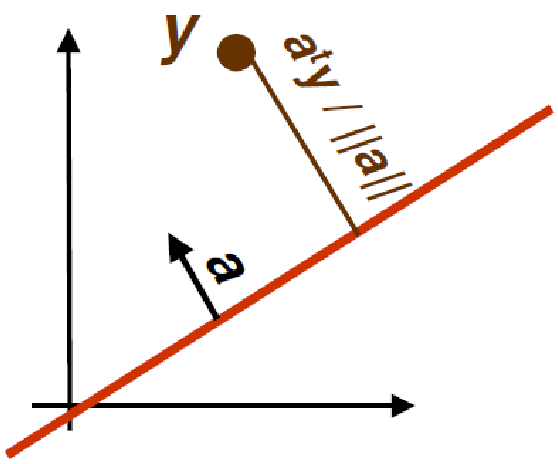




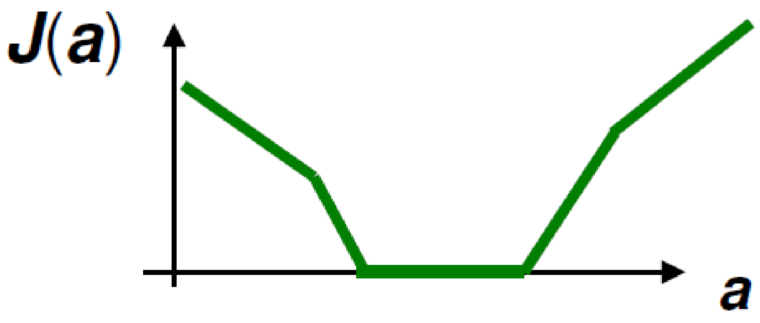
- Seek a hyperplane that separates patterns from different categories
- Seek hyperplane that puts *normalized* patterns on the same(positive) side

Perceptron Criterion Function

$$J_p(a) = \sum_{y \in Y_M} \left(-a^t y\right)$$



- If *y* is misclassified, $a^t y < 0$
- Thus $J_p(a) > 0$
- $J_p(a)$ is $||a||$ times the sum of distances of misclassified examples to decision boundary
- Jp(a) is piecewise linear and thus suitable for gradient descent



Perceptron Batch Rule

$$J_p(a) = \sum_{y \in Y_M} \left(-a^t y\right)$$

- Gradient of $J_p(a)$ is: $\nabla J_p(a) = \sum_{y \in Y_M} (-y)$
  - Y_M are samples misclassified by $a^{(k)}$
  - It is not possible to solve $\nabla J_p(a) = 0$ analytically because of Y_M
- Update rule for gradient descent: $x^{(k+1)} = x^{(k)} - \eta^{(k)} \nabla J(x)$
- Thus the *gradient decent batch update rule for* $J_p(a)$ is: $$a^{(k+1)} = a^{(k)} + \eta^{(k)} \sum_{y \in Y_M} y$$
- It is called batch rule because it is based on all misclassified examples

Boosting

- See also http://www.cs.stevens.edu/~mordohai/classes/cs559_f14.html
  - Notes 11

Boosting

- Idea: given a set of weak learners, run them multiple times on (reweighted) training data, then let learned classifiers vote
- At each iteration *t:*
  - Weight each training example by how incorrectly it was classified
  - Learn a hypothesis – h_t
  - Choose a strength for this hypothesis – α_t
- Final classifier: weighted combination of weak learners

Learning from Weighted Data

- Sometimes not all data points are equal
  - Some data points are more equal than others
- Consider a weighted dataset
  - D(i) – weight of i th training example ($\mathbf{x}_i$,$\mathbf{y}_i$)
  - Interpretations:
    - i th training example counts as D(i) examples
    - If I were to "resample" data, I would get more samples of "heavier" data points
- Now, in all calculations the *i th* training example counts as D(i) "examples"

Definition of Boosting

- Given training set $(x_1,y_1),\ldots,(x_m,y_m)$
- $y_i \in \{-1,+1\}$ correct label of instance $x_i \in X$
- For $t=1,\ldots,T$
  - construct distribution D_t on $\{1,\ldots,m\}$
  - find weak hypothesis
  - $h_t: X \rightarrow \{-1,+1\}$
    with small error ε_t on D_t

$$\epsilon_t = \Pr_{i \sim D_t}\left[h_t(x_i) \neq y_i\right]$$

- Output final hypothesis H_{final}

AdaBoost

- Constructing D_t
  - $D_1 = 1/m$
  - Given D_t and h_t: $$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \cdot \begin{cases} e^{-\alpha_t} & \text{if } y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if } y_i \neq h_t(x_i) \end{cases}$$ $$= \frac{D_t(i)}{Z_t} \cdot \exp(-\alpha_t \, y_i \, h_t(x_i))$$

    where Z_t is a normalization constant $$Z_t = \sum_{i=1}^{m} D_t(i) \exp(-\alpha_t y_i h_t(x_i))$$

- Final hypothesis: $$\alpha_t = \tfrac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right) > 0$$

$$\boxed{H_{\text{final}}(x) = \text{sign}\left(\sum_t \alpha_t h_t(x)\right)}$$

Face Detection

- I see this as a two person project
  - One implements boosting as before
  - One implements the face-specific parts

- See also http://www.cs.stevens.edu/~mordohai/classes/cs559_f14.html
  - Notes 11

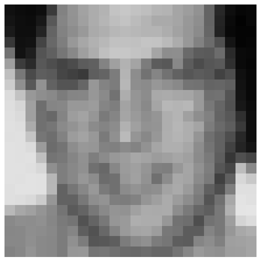
Classifier is Learned from Labeled Data

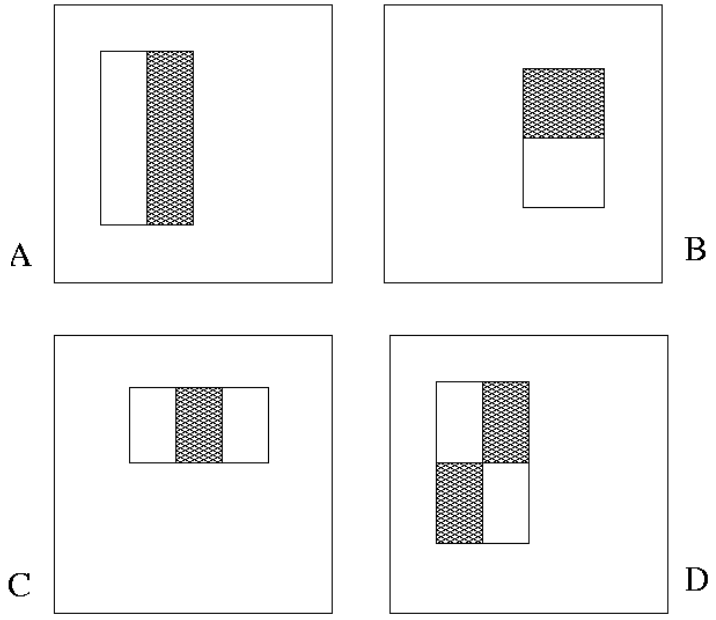
- Training Data
  - 5000 faces
    - All frontal
  - 10^8 non faces
  - Faces are normalized
    - Scale, translation
- Many variations
  - Across individuals
  - Illumination
  - Pose (rotation both in plane and out)

Boosted Face Detection: Image Features

"Rectangle filters"

Similar to Haar wavelets





$$h_t(x_i) = \begin{cases} \alpha_t & \text{if } f_t(x_i) > \theta_t \\ \beta_t & \text{otherwise} \end{cases}$$

$$C(x) = \theta\left(\sum_t h_t(x) + b\right)$$

$$60{,}000 \times 100 = 6{,}000{,}000$$

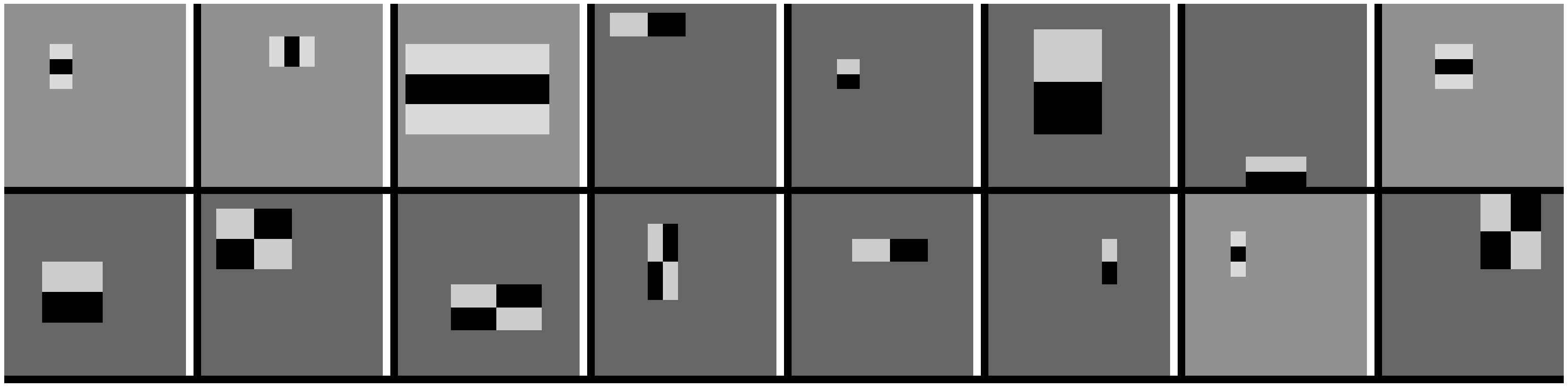
Unique Binary Features

Feature Selection

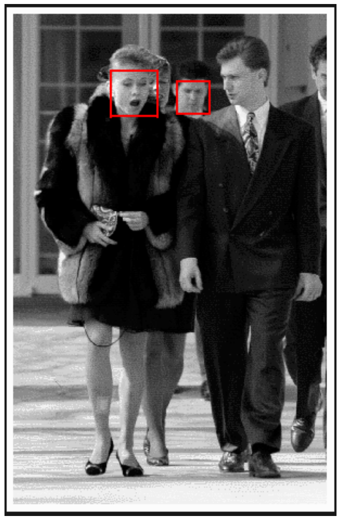
- For each round of boosting:
  - Evaluate each rectangle filter on each example
  - Sort examples by filter values
  - Select best threshold for each filter
  - Select best filter/threshold (= Feature)
  - Reweight examples

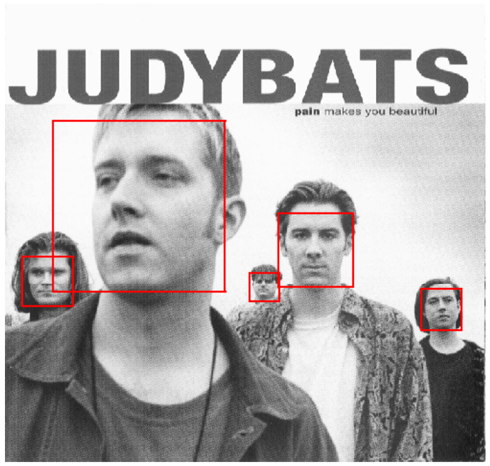
Feature Localization

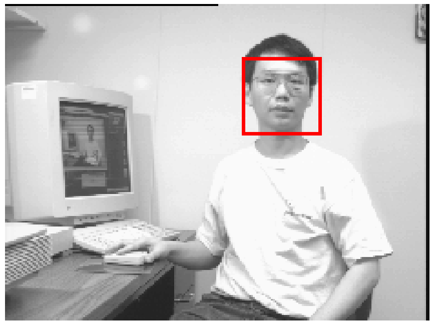
- Learned features reflect the task

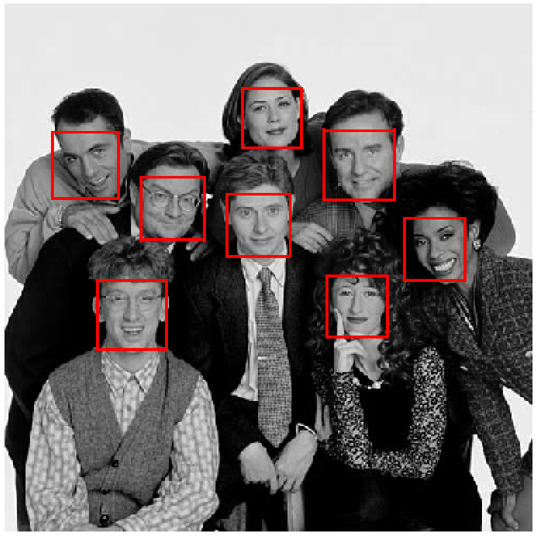


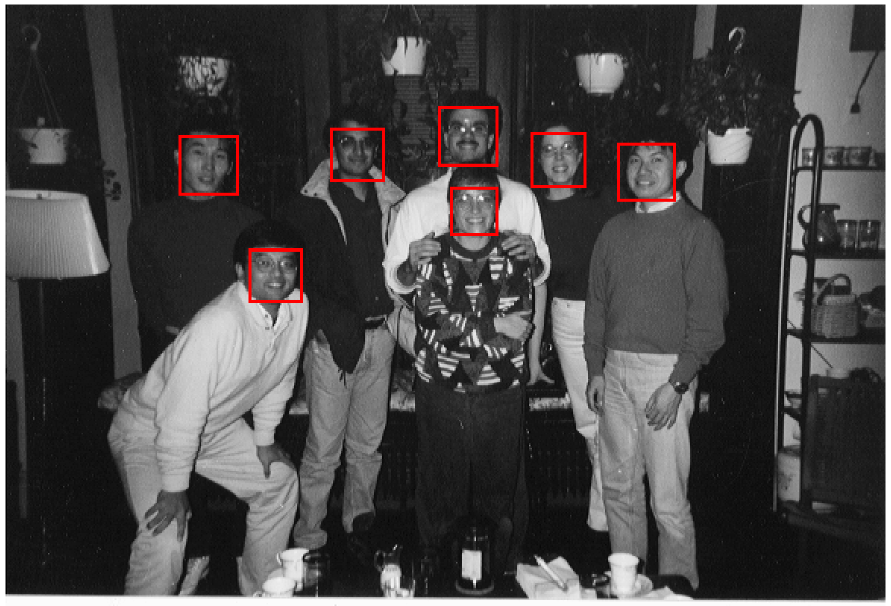
Output of Face Detector on Test Images







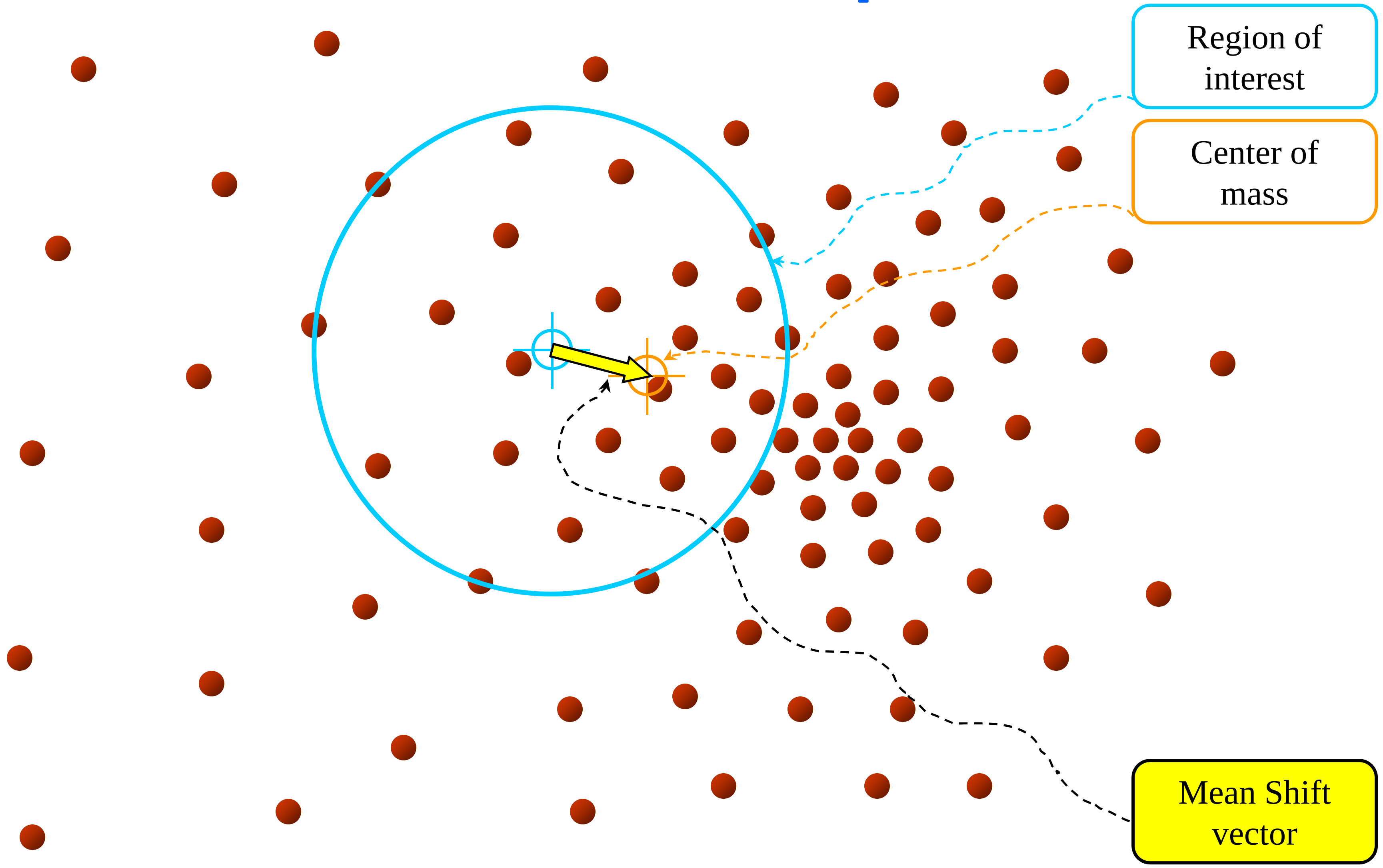




Mean Shift

- See also http://www.cs.stevens.edu/~mordohai/classes/cs559_f10.html
  - Notes 13

Intuitive Description



Objective : Find the densest region

Distribution of identical billiard balls

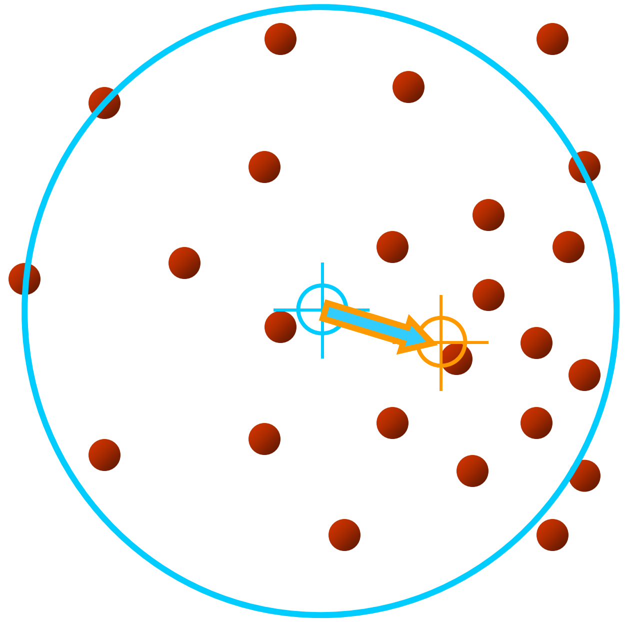
Computing The Mean Shift

Simple Mean Shift procedure:

• Compute mean shift vector

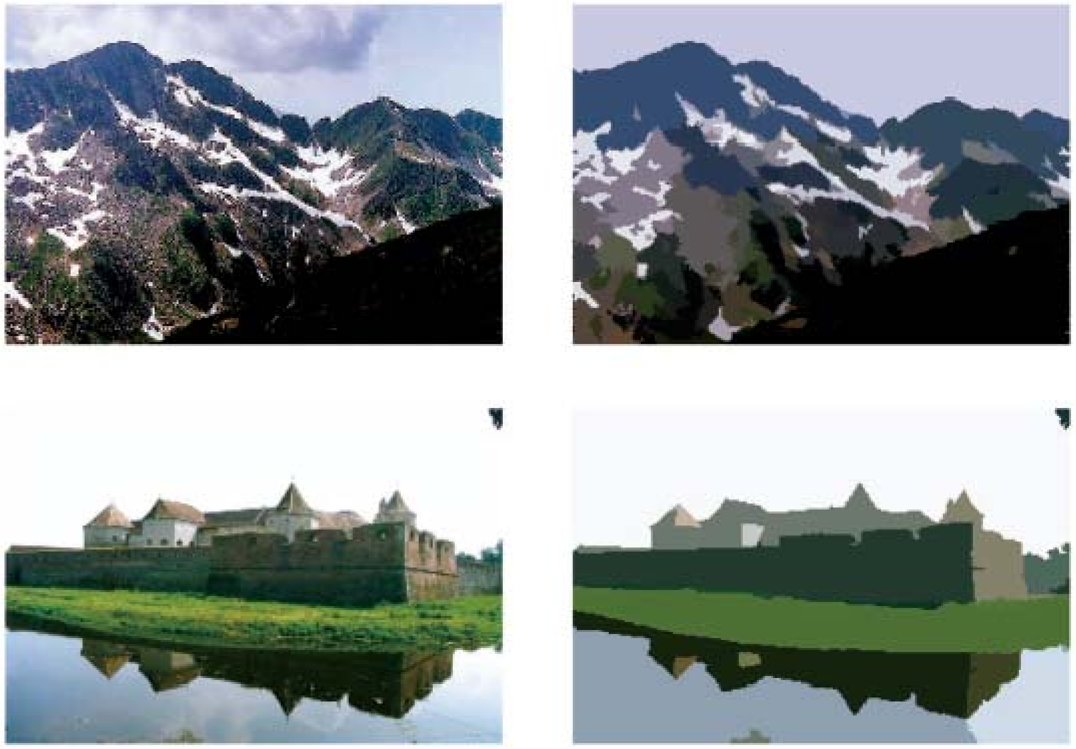
$$\mathbf{m}(\mathbf{x}) = \left[\frac{\sum_{i=1}^{n} \mathbf{x}_i g\left(\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{h} \right)}{\sum_{i=1}^{n} g\left(\frac{\|\mathbf{x} - \mathbf{x}_i\|^2}{h} \right)} - \mathbf{x} \right]$$

•Translate the Kernel window by **m(x)**



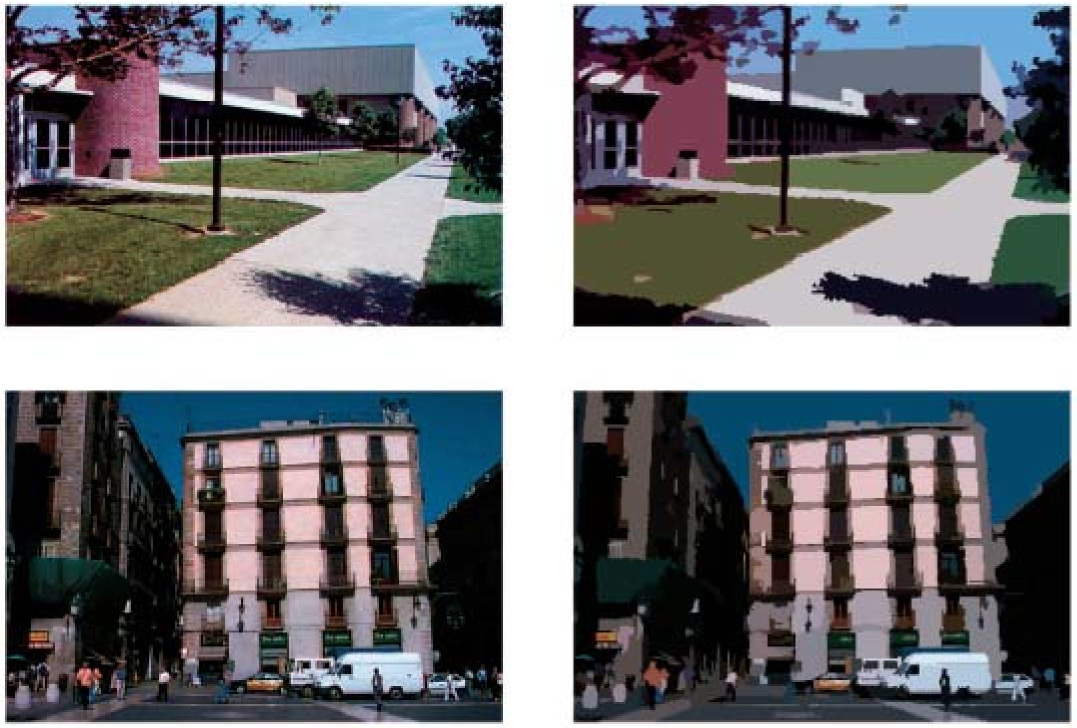
Segmentation

Example

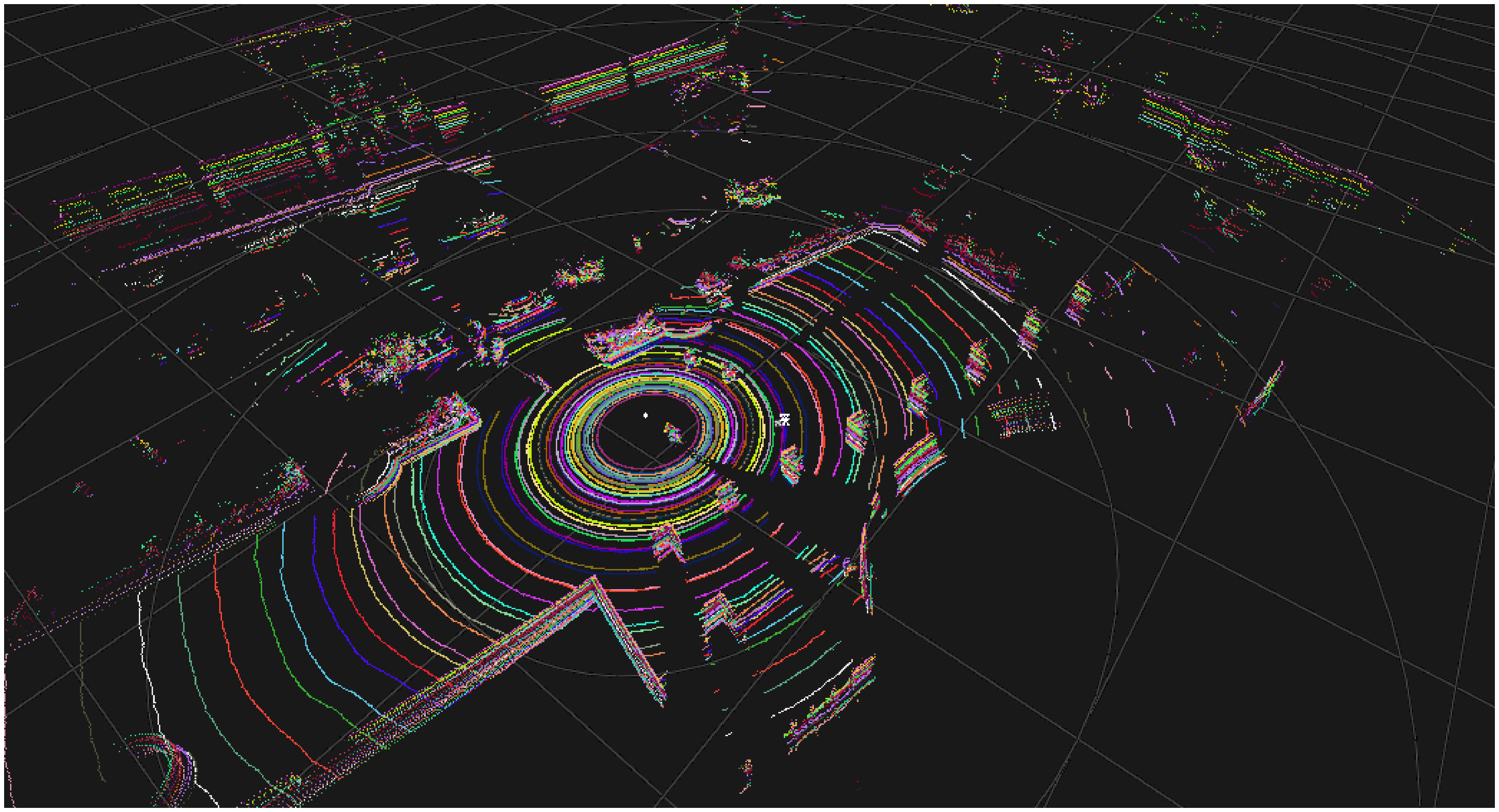


Segmentation

Example



Normal Estimation for 3D Point Clouds



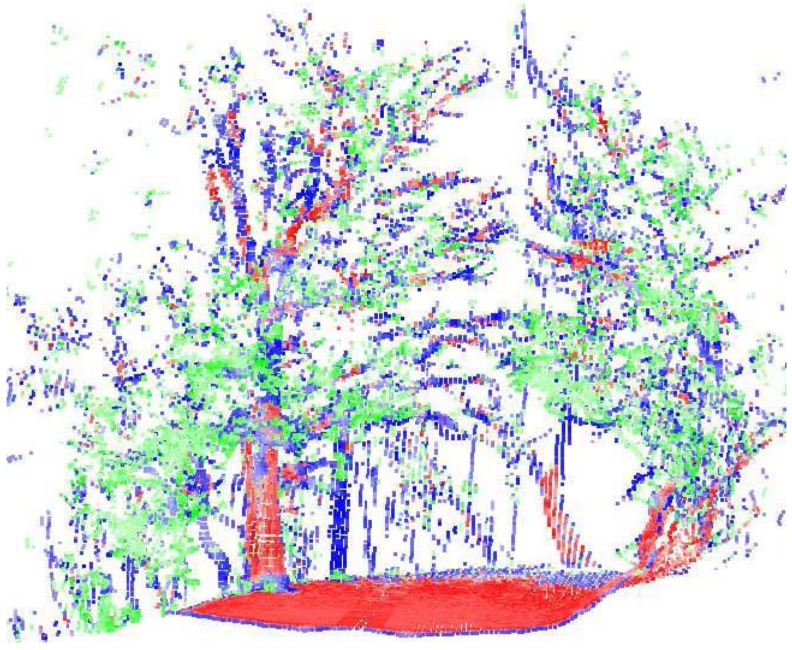
Scatter Matrix

- Compute the symmetric positive definite covariance matrix from N neighbors of a 3-D point
  - $\{X_i\} = \{(x_i, y_i, z_i)^T\}$

$$\frac{1}{N}\sum_{i=i}^{N}(X_i - \overline{X})(X_i - \overline{X})^T$$

- Then, the eigenvector that corresponds to the smallest eigenvalue is the normal to the surface at each point
  - If each point belonged to a smooth surface

Classification

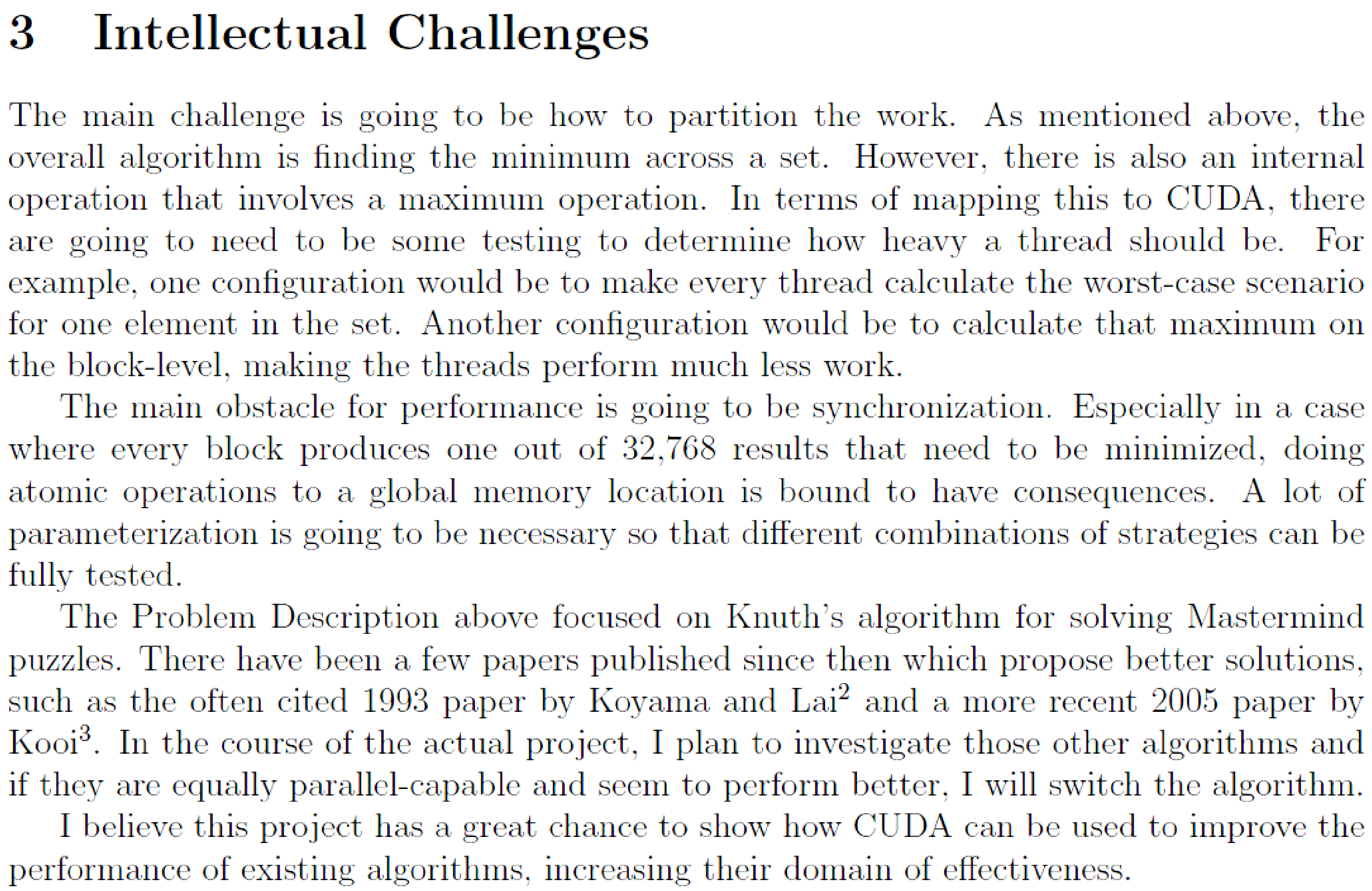


- Points can be classified according to eigenvalues into surfaces, foliage, ground plane etc.
  - Images from Lalonde et al. 2006

Markov Chain Monte Carlo

- Randomized algorithms based on sampling from probability distributions to generate sequences of observations
- Applications
  - Approximate integration
  - Optimization of energy/cost functions in very large search spaces
  - Risk assessment in finance

Sample Proposal



3 Intellectual Challenges

The main challenge is going to be how to partition the work. As mentioned above, the overall algorithm is finding the minimum across a set. However, there is also an internal operation that involves a maximum operation. In terms of mapping this to CUDA, there are going to need to be some testing to determine how heavy a thread should be. For example, one configuration would be to make every thread calculate the worst-case scenario for one element in the set. Another configuration would be to calculate that maximum on the block-level, making the threads perform much less work.

The main obstacle for performance is going to be synchronization. Especially in a case where every block produces one out of 32,768 results that need to be minimized, doing atomic operations to a global memory location is bound to have consequences. A lot of parameterization is going to be necessary so that different combinations of strategies can be fully tested.

The Problem Description above focused on Knuth's algorithm for solving Mastermind puzzles. There have been a few papers published since then which propose better solutions, such as the often cited 1993 paper by Koyama and Lai[2] and a more recent 2005 paper by Kooi[3]. In the course of the actual project, I plan to investigate those other algorithms and if they are equally parallel-capable and seem to perform better, I will switch the algorithm.

I believe this project has a great chance to show how CUDA can be used to improve the performance of existing algorithms, increasing their domain of effectiveness.

Overview

- Timers
- Case Study – Advanced MRI Reconstruction
  - A class project at UIUC resulting in a publication

Timers

- Any timer can be used
  - Check resolution
- Important: many CUDA API functions are asynchronous
  - They return control back to the calling CPU thread prior to completing their work
  - All kernel launches are asynchronous
  - So are all memory copy functions with the Async suffix on the name

Synchronization

- Synchronize the CPU thread with the GPU by calling `cudaThreadSynchronize()` immediately before starting and stopping the CPU timer
- `cudaThreadSynchronize()`blocks the calling CPU thread until all CUDA calls previously issued by the thread are completed

Synchronization

- `cudaEventSynchronize()` blocks until a given event in a particular stream has been recorded by the GPU
  - Safe only in the default (0) stream
  - Fine for our purposes

CUDA Timer

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

kernel<<<grid,threads>>> ( d_odata, d_idata,
  size_x, size_y, NUM_REPS);

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop ); // after cudaEventRecord
cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Output

- `time` is in milliseconds
- Its resolution of approximately half a microsecond
- The timings are measured on the GPU clock
  - Operating system-independent

Application Case Study – Advanced MRI Reconstruction

Objective

- To learn about computational thinking skills through a concrete example
  - Problem formulation
  - Designing implementations to steer around limitations
  - Validating results
  - Understanding the impact of your improvements

Acknowledgements

Sam S. Stone[§], Haoran Yi[§], Justin P. Haldar[†], Deepthi Nandakumar, Bradley P. Sutton[†],
Zhi-Pei Liang[†], Keith Thulburin[*]

[§]Center for Reliable and High-Performance Computing

[†] Beckman Institute for Advanced Science and Technology

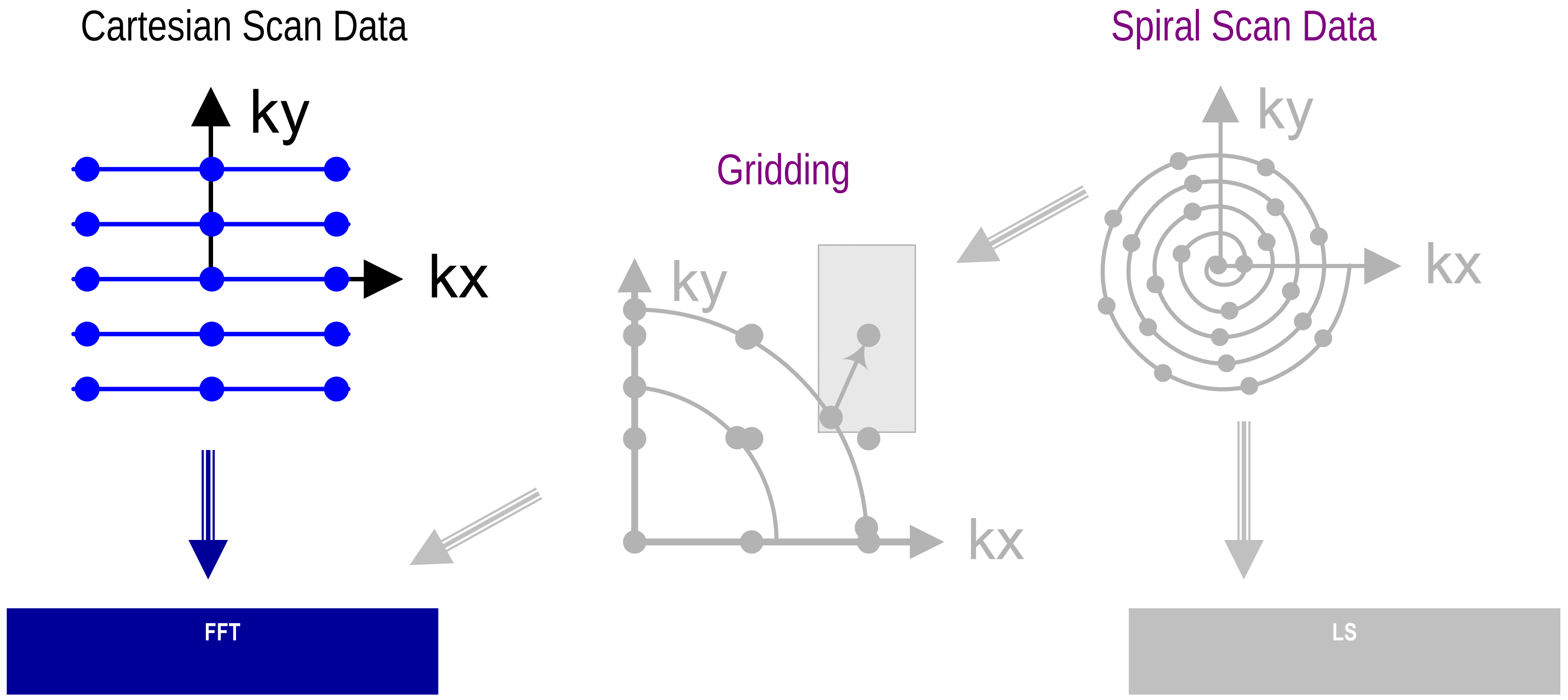
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

* *University of Illinois, Chicago Medical Center*

Overview

- Magnetic resonance imaging
- Non-Cartesian Scanner Trajectory
- Least-squares (LS) reconstruction algorithm
- Optimizing the LS reconstruction on the G80
    - Overcoming bottlenecks
    - Performance tuning
- Summary

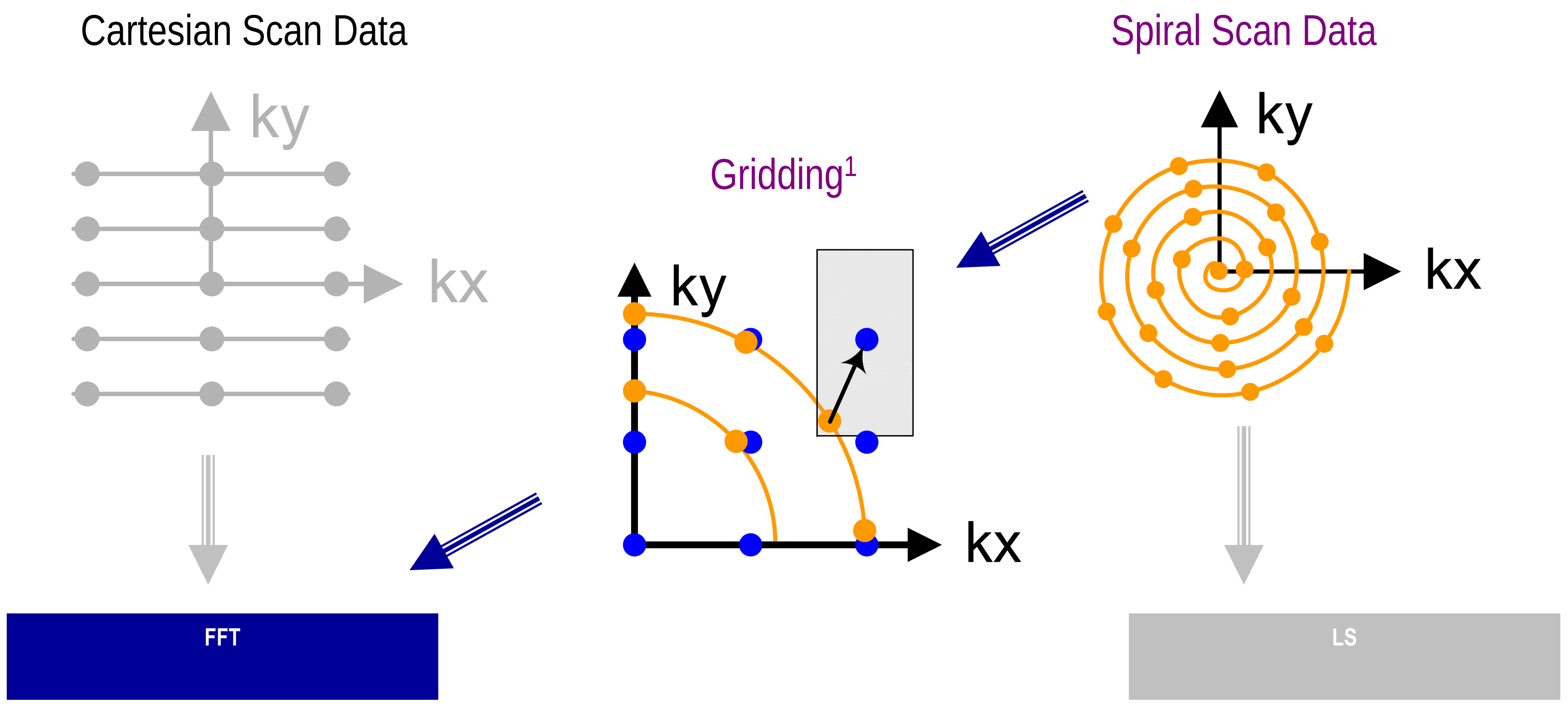
Reconstructing MR Images



Cartesian scan data + FFT:

Slow scan, fast reconstruction, images may be poor

Reconstructing MR Images

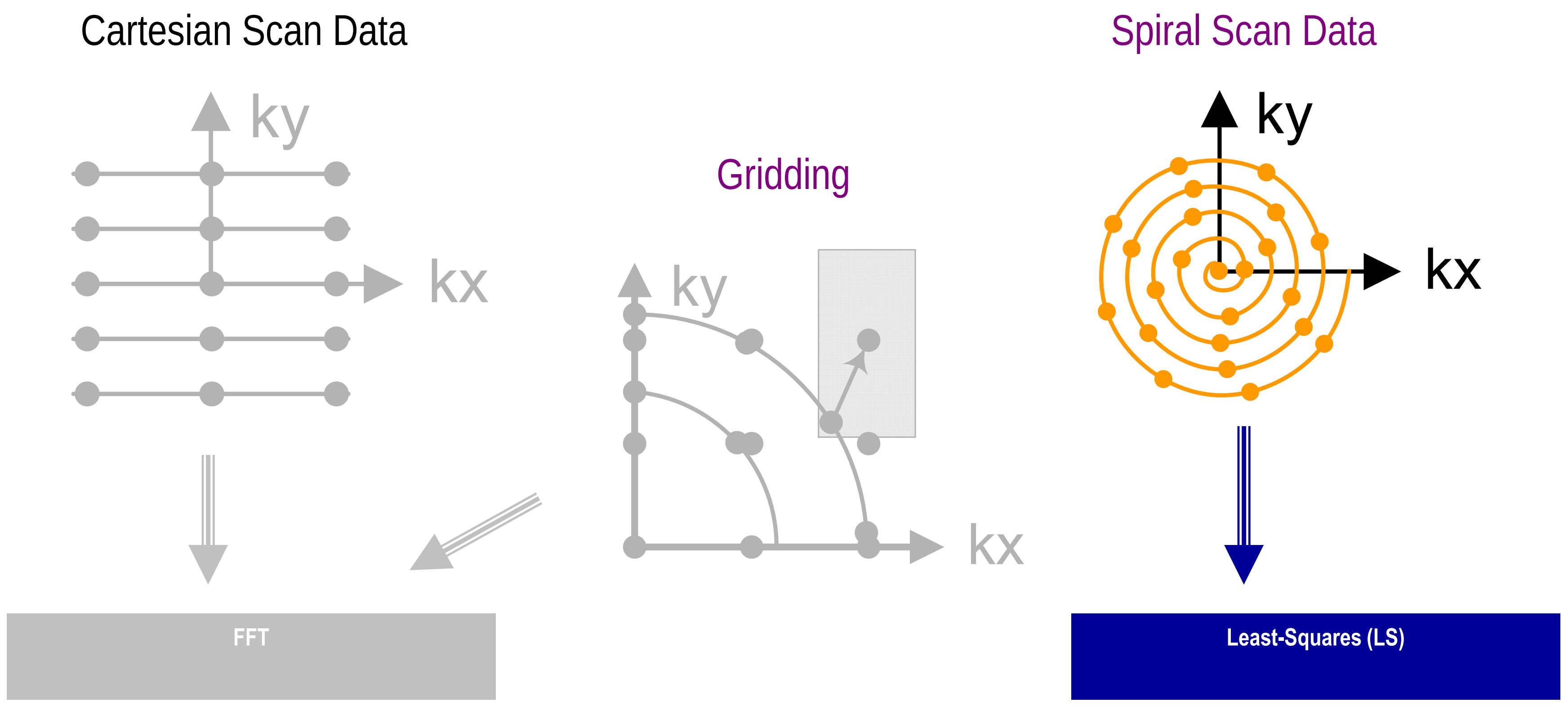


Spiral scan data + Gridding + FFT:

Fast scan, fast reconstruction, better images

[1] Based on Fig 1 of Lustig et al, Fast Spiral Fourier Transform for Iterative MR Image Reconstruction, IEEE Int'l Symp. on Biomedical Imaging, 2004

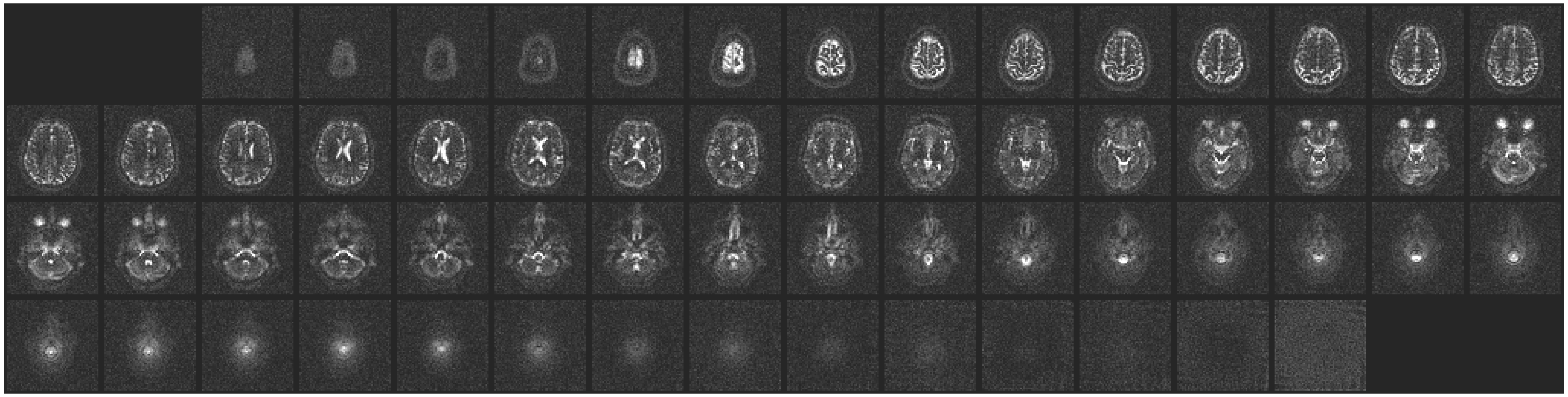
Reconstructing MR Images



Spiral scan data + LS

Superior images at expense of significantly more computation

An Exciting Revolution - Sodium Map of the Brain



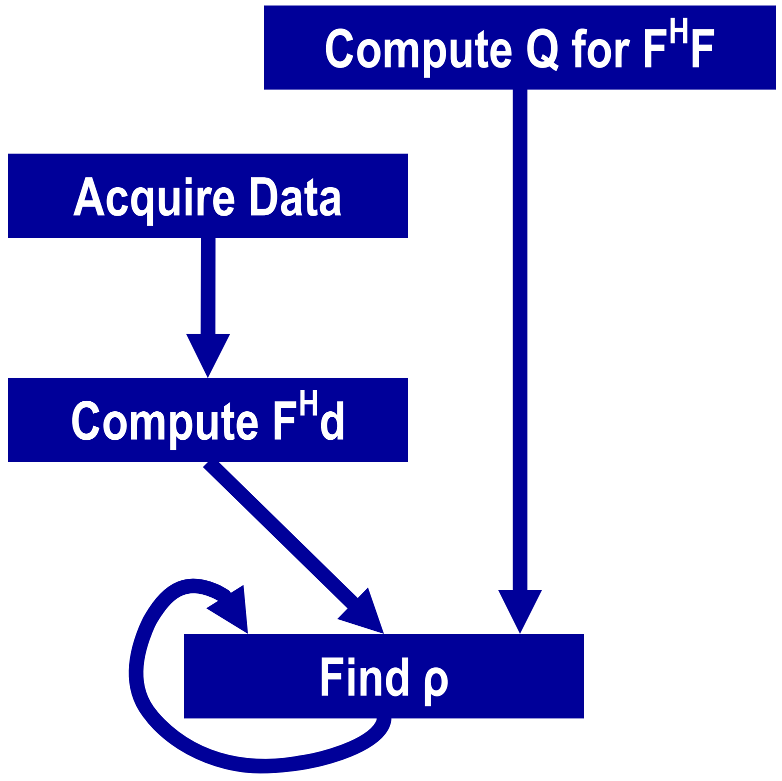
- Images of sodium in the brain
  - Very large number of samples for increased SNR
  - Requires high-quality reconstruction

- Enables study of brain-cell viability before anatomic changes occur in stroke and cancer treatment – within days!

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

Least-Squares Reconstruction

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



- $F^H F$ depends only on scanner configuration
- $W^H W$ incorporates prior information, such as anatomical constraints
- $F^H d$ depends on scan data
- ρ vector containing voxel values of reconstructed image - found using linear solver
  - 99.5% of the reconstruction time for a single image is devoted to computing $F^H d$
  - computing Q is even more expensive, but depends only on the scanner configuration and can be amortized

Least-Squares Reconstruction

- The solution is:

$$\rho = (F^H F + W^H W)^{-1} F^H d$$

- but for a relatively low-res reconstruction of 128^3 voxels, the inverted matrix contains well over four trillion complex-valued elements
- Use conjugate gradient to solve

Least-Squares Reconstruction

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

- $W^H W$ is sparse
- $F^H F$ has convolutional structure
  - each descending diagonal from left to right is constant
- Efficient FFT-based matrix multiplication is possible
  - Out of scope for CS 677

Least-Squares Reconstruction

- What has to be computed is the Q matrix which depends only on the scan trajectory, but not the scan data

$$Q(x_n) = \sum_{m=1}^{M} |\varphi(k_m)|^2 e^{(i2\pi k_m \cdot x_n)}$$

- where:
  - k_m is the location of the mth sample
  - x_n is the nth voxel
  - φ() is the Fourier transform of the voxel basis function

Least-Squares Reconstruction

- What also needs to be computed is the vector F^{H}d which depends on the data

$$[F^H d]_n = \sum_{m=1}^{M} \varphi^*(k_m) d(k_m) e^{(i2\pi k_m \cdot x_n)}$$

- These two equations look similar but the computation of Q requires oversampling by a factor of 2 in each dimension
  - Q is O(8MN) and F^{H}d is O(MN)

Least-Squares Reconstruction - Complexity

- Q: 1-2 days on CPU
- $F^H d$: 6-7 hours on CPU
- ρ: 1.5 minutes on CPU

- Therefore, accelerate Q and $F^H d$ computations

```
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

```
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^Hd computation

Q v.s. F^HD

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
- Voxel data
  - N = # voxels
  - x, y, z = input 3D voxel data
  - rFhD, iFhD= output voxel data
- Complexity is O(MN)
- Inner loop
  - 14 FP MUL or ADD ops
  - 2 FP trig ops (12-13 FL OPs)
  - 12 loads, 2 stores

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;
  }
}
```

1. Each thread executes an iteration of the outer loop
   => Problem: Each thread is trying to accumulate a partial sum to rFhD and iFhD (requires a reduction)
2. Each thread executes an iteration of the inner loop.
   - Avoids the reduction problem
   - But now each thread is doing very little work
   - We need one grid for each outer loop iteration.
   - Performance limited by overheads for launching M grids and writing 2N values to global memory for each grid

One Possibility (Wrong)

```
__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;
  }
}
```

One Possibility (Wrong) - Improved

```
__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;
  float rMu_reg, iMu_reg;

  rMu_reg = rMu[m] = rPhi[m]*rD[m] + iPhi[m]*iD[m];
  iMu_reg = 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_reg*cArg – iMu_reg*sArg;
    iFhD[n] +=  iMu_reg*cArg + rMu_reg*sArg;
  }
}
```

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++) {
 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

```
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];
}
```

A Second Option for the cmpFHd Kernel

```
__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;

  for (n = 0; n < N; 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;
  }
}
```

Problem: Each thread is trying to accumulate a partial sum to rFhD and iFhD

We do have another option

```
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
```

Loop interchange of the F^HD computation

A Third Option for the FHd kernel

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

  int n = blockIdx.x * FHD_THREADS_PER_BLOCK + threadIdx.x;

 for (m = 0; m < M; m++) {
    float rMu_reg = rMu[m];
   float iMu_reg = iMu[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_reg*cArg – iMu_reg*sArg;
    iFhD[n] +=  iMu_reg*cArg + rMu_reg*sArg;
  }
}
```

From C to CUDA: Step 2 Getting around Memory Bandwidth Limitations

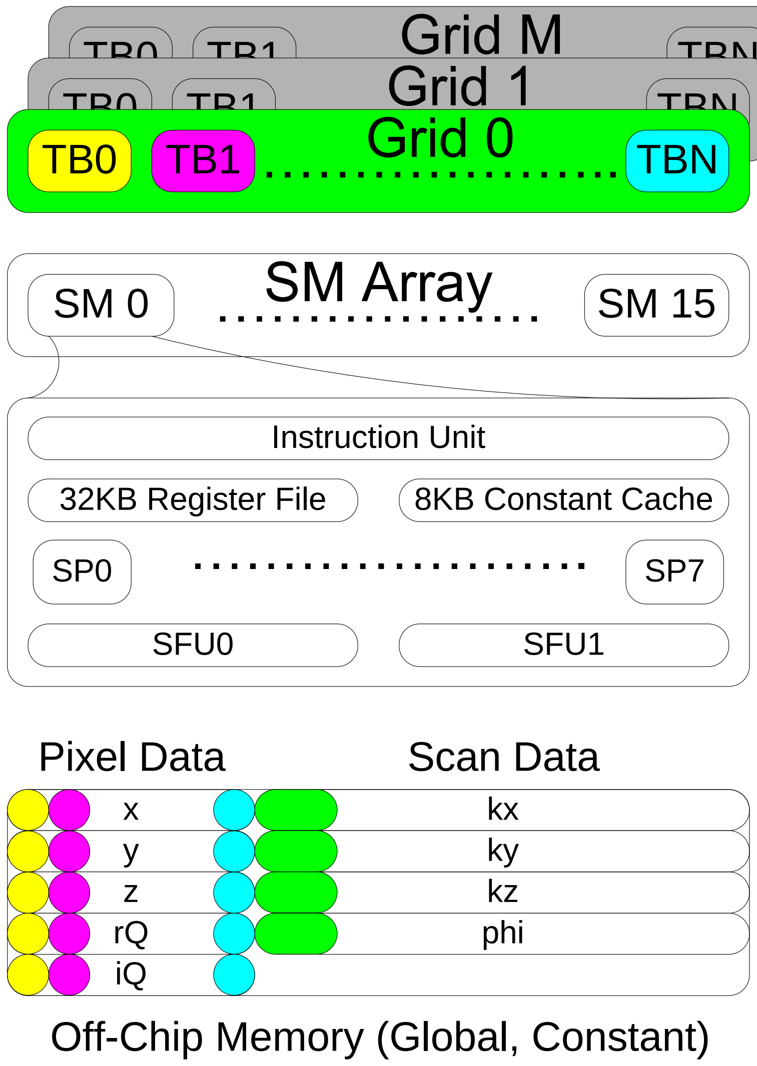
- Using registers
- Using constant memory

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;
}
```

Compute-to-memory access ratio 14:7 (inside the loop)
Was 14:14 before (approx.)

Tiling of Scan Data



LS reconstruction uses multiple grids

- Each grid operates on all scan data
- Each grid operates on a distinct subset of voxels
- Each thread in the same grid operates on a distinct voxel

Thread n operates on voxel 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)
}
```

Using Constant Memory

- All threads access scan data (kx, ky, kz) in the same order
- Threads don't modify scan data

- ➢ Put scan data in constant memory
  - ➢ Limited to 64kB (larger than shared memory)
  - ➢ But cached, for every 32 accesses to constant memory, at least 31 will be cached (96% reduction in time, no bank conflicts – broadcast mode to all threads in warp)

Chunking k-space Data to Fit into Constant Memory

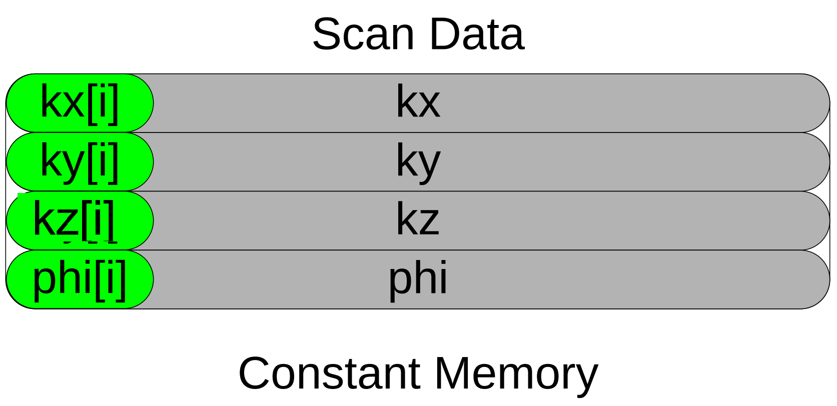
```
__constant__ float  kx_c[CHUNK_SIZE],
                    ky_c[CHUNK_SIZE], kz_c[CHUNK_SIZE];
…
__void main() {

  int i;
  for (i = 0; i < M/CHUNK_SIZE; i++);
    cudaMemcpyToSymbol(kx_c,&kx[i*CHUNK_SIZE],4*CHUNK_SIZE,
                     cudaMemCpyHostToDevice);
    cudaMemcpyToSymbol(ky_c,&ky[i*CHUNK_SIZE],4*CHUNK_SIZE,
                     cudaMemCpyHostToDevice);
    cudaMemcpyToSymbol(kz_c,&kz[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 */
}
```

Revised Kernel for Constant Memory

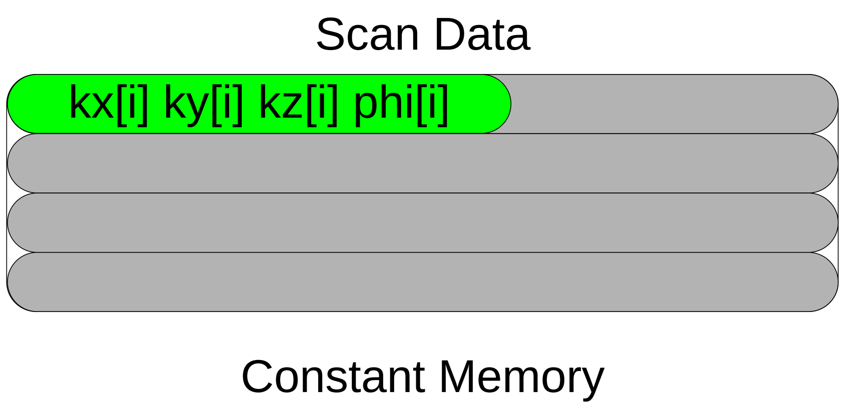
```
__global__ void cmpFHd(float* rPhi, iPhi, phiMag,
      x, y, z, rMu, iMu, int M) {

  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*(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;
}
```

kx_c, ky_c and kz_c are no longer arguments but global variables

Compute-to-memory access ratio 14:4 (inside the loop)
Can be 14:2 if compiler stores rMu[m] and iMu[m] in temporary registers



(a) k-space data stored in separate arrays.



(b) k-space data stored in an array whose elements are structs.

Effect of k-space data layout on constant cache efficiency.

- The previous implementations leads to bad (slow) performance
- Each constant cache entry is designed to store multiple consecutive words
- There are very few such entries – insufficient for all active warps in an SM
- Solution: use array of struct (contrary to last week’s advice)

```
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,
       cudaMemCpyHostToDevice);

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

  }
```

Adjusting k-space data layout to improve cache efficiency

```
__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;
}
```

Adjusting the k-space data memory layout in the F^{H}d kernel

From C to CUDA: Step 3 Where are the potential bottlenecks?

Bottlenecks

- Memory Bandwidth
  - See previous slides
- Trig operations
- Overhead (branches, address calculations)
  - These are important due to short inner loop

Trigonometric Operations

- Use SFUs (Super Function Units)
  - __sin and __cos are implemented as hardware instructions
    - Require 4 cycles (vs. 12 and 13 FLOP for software versions)
    - Reduced accuracy

- Performance: from 22.8 GFLOPS to 92.2 GFLOPS

Address Calculations

- Last bottleneck: Overhead of branches and address calculations
- Solution: Loop unrolling and experimental tuning
  - Loop unrolling factors (1,2,4,8,16)
  - Also experimentally tuned the number of threads per block and the number of scan points per grid (see following slides)
- Performance:179 GFLOPS (Q), 145 GFLOPS ($F^H d$)

Experimental Methodology

- Reconstruct a 3D image of a human brain[1]
  - 3.2 M scan data points acquired via 3D spiral scan
  - 256K voxels
- Compare performance of several reconstructions
  - Gridding + FFT reconstruction[1] on CPU (Intel Core 2 Extreme Quadro)
  - LS reconstruction on CPU (double-precision, single-precision)
  - LS reconstruction on GPU (NVIDIA GeForce 8800 GTX)
- Metrics
  - Reconstruction time: compute $F^H d$ and run linear solver
  - Run time: compute Q or $F^H d$

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

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 applications are resistant to approximations; others are very sensitive

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

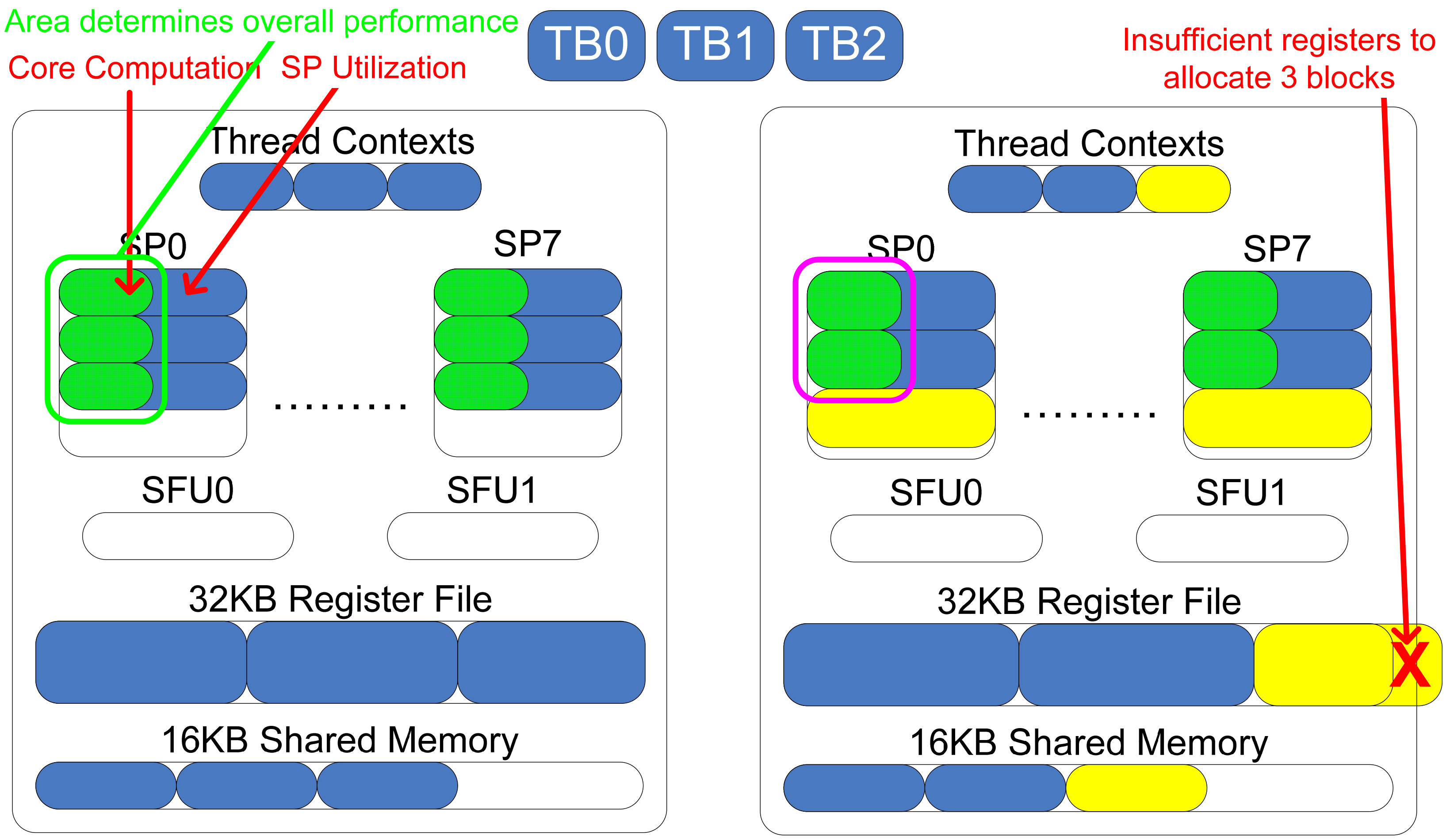
$$PSNR = 20\log_{10}\left(\frac{\max(I_0(i,j))}{\sqrt{MSE}}\right) \qquad SNR = 20\log_{10}\left(\frac{\sqrt{A_s}}{\sqrt{MSE}}\right)$$

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)
- Cannot 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

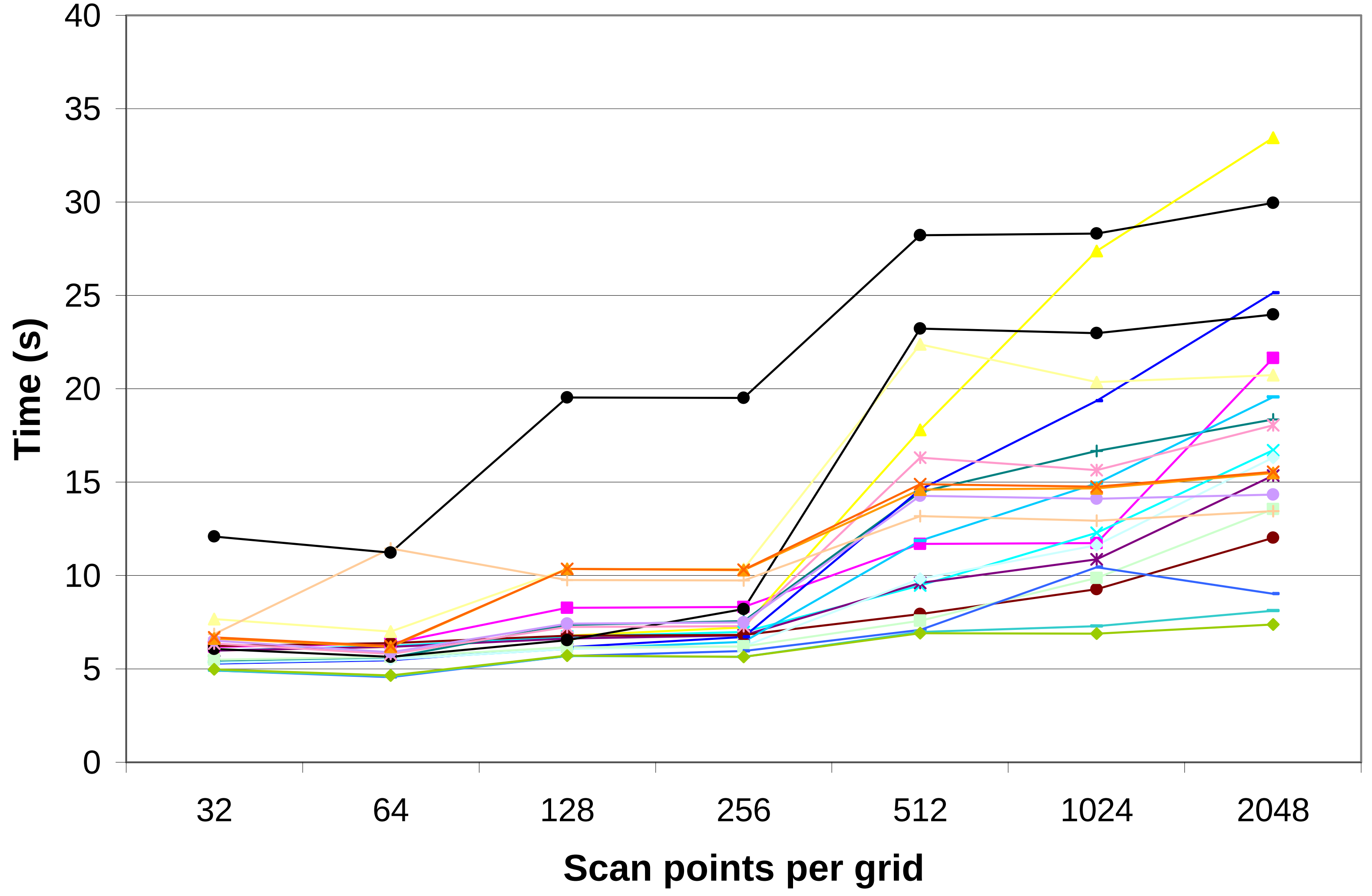


(a) Pre-"optimization"

(b) Post-"optimization"

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

Experimental Tuning: Scan Points Per Grid

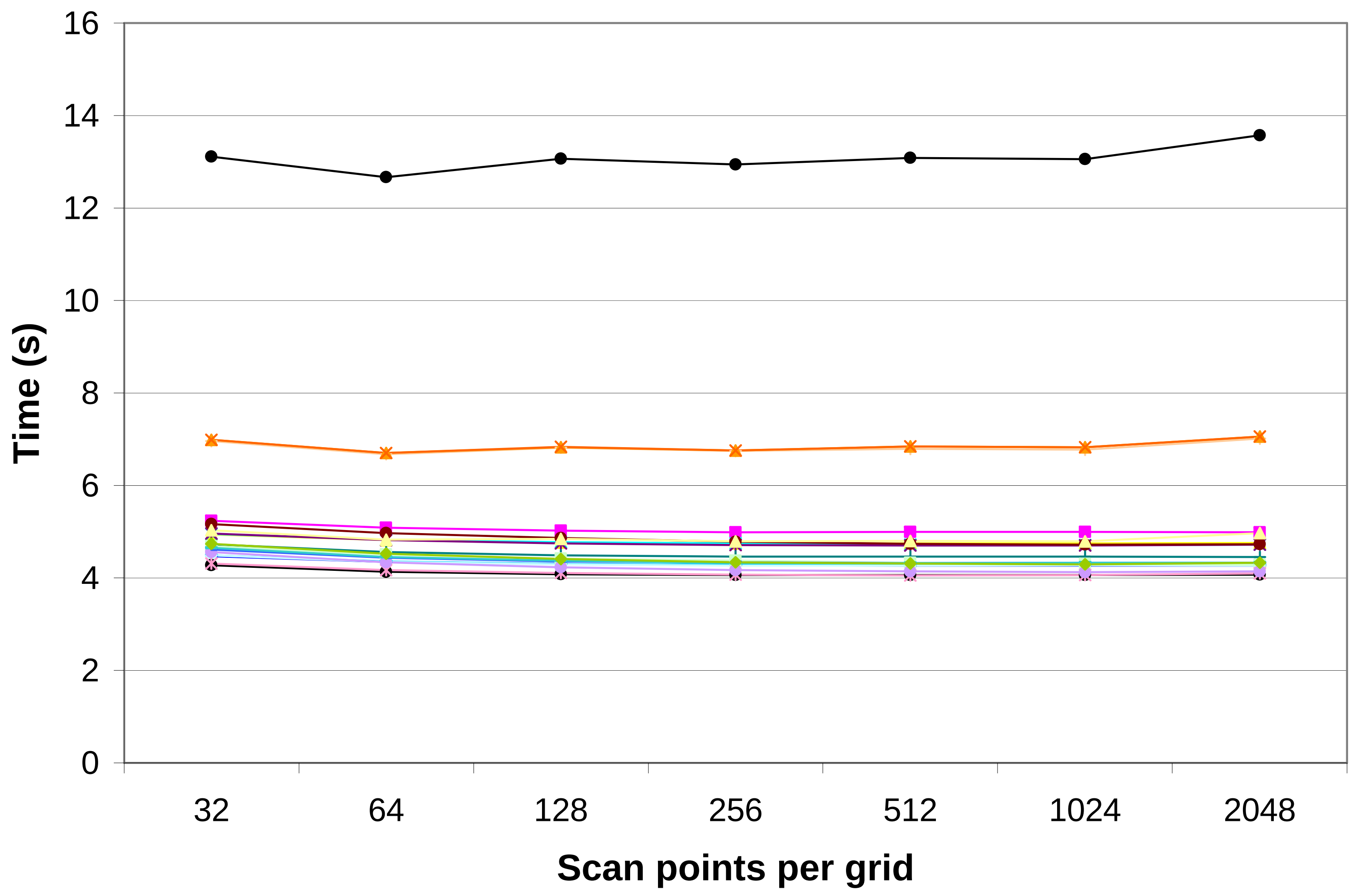


Experimental Tuning: Scan Points Per Grid

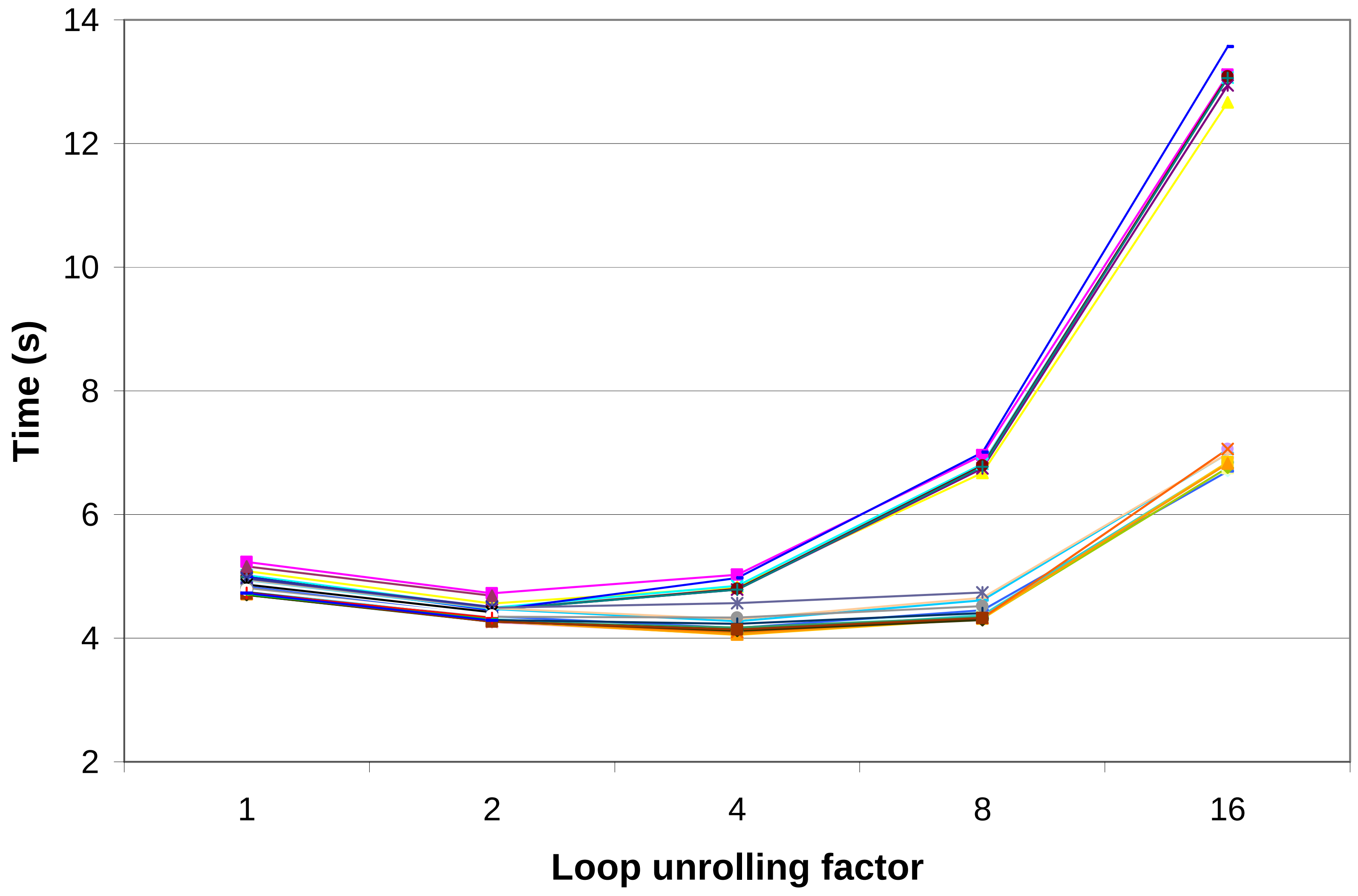
- Each line in previous plot represents a combination of loop unrolling factor and threads per block
- The y-axis represents runtime, so lower is better

- Runtime tends to increase as the number of scan points per grid increases
- That’s counter-intuitive. Why would performance get worse as the amount of data processed by each kernel increased?
  - ➢ Conflicts in the constant cache (across different blocks)

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



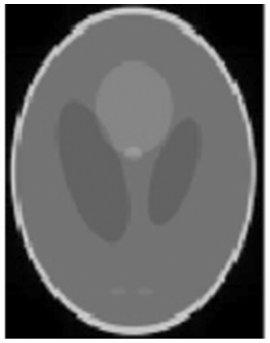
Experimental Tuning: Loop Unrolling Factor

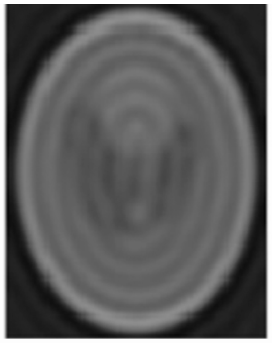


Sidebar: Optimizing the CPU Implementation

- 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)

Quantitative Evaluation


(1) True


(2) Gridded
41.7% error
PSNR = 16.8 dB

(3) CPU.DP
12.1% error
PSNR = 27.6 dB

(4) CPU.SP
12.0% error
PSNR = 27.6 dB

(5) GPU.Base
12.1% error
PSNR = 27.6 dB

(6) GPU.RegAlloc
12.1% error
PSNR = 27.6 dB

(7) GPU.Coalesce
12.1% error
PSNR = 27.6 dB

(8) GPU.ConstMem
12.1% error
PSNR = 27.6 dB

(9) GPU.FastTrig
12.1% error
PSNR = 27.5 dB

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^Hd			
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