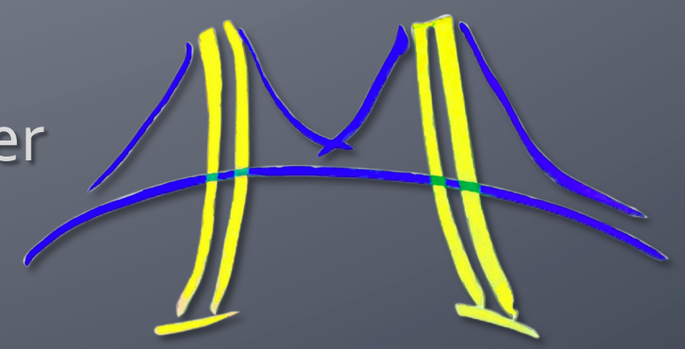
An Introduction to CUDA/OpenCL and Manycore Graphics Processors

Bryan Catanzaro, UC Berkeley
with help from the PALLAS research group

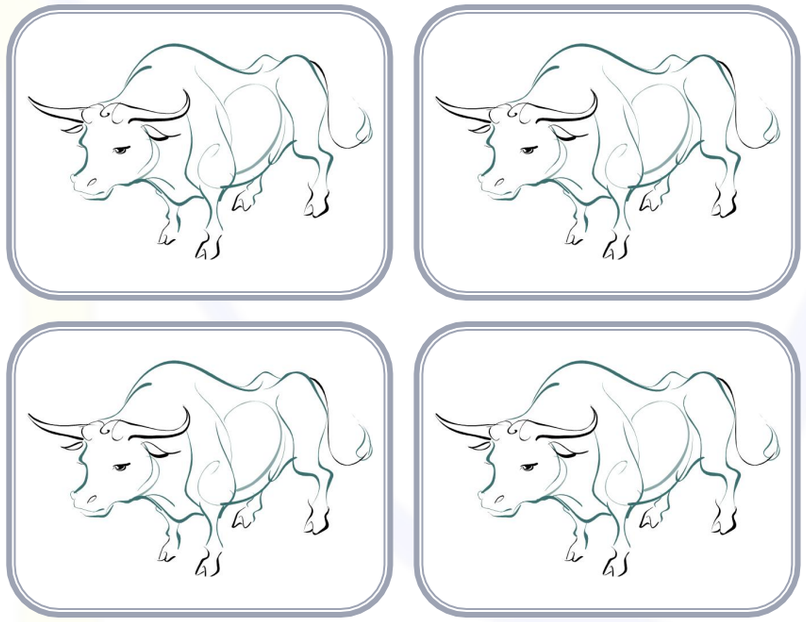
Universal Parallel Computing Research Center
University of California, Berkeley



Overview

- Terminology: Multicore, Manycore, SIMD
- The CUDA and OpenCL programming models
- Mapping CUDA to Nvidia GPUs
- OpenCL

Multicore and Manycore



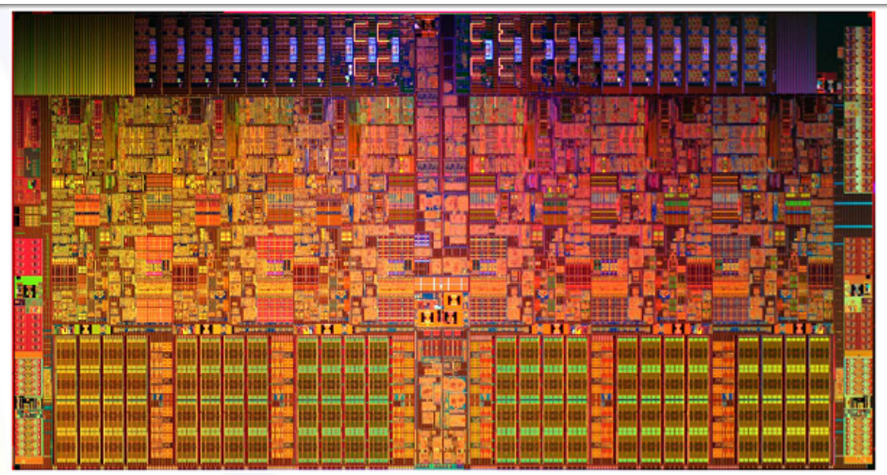
Multicore

Manycore

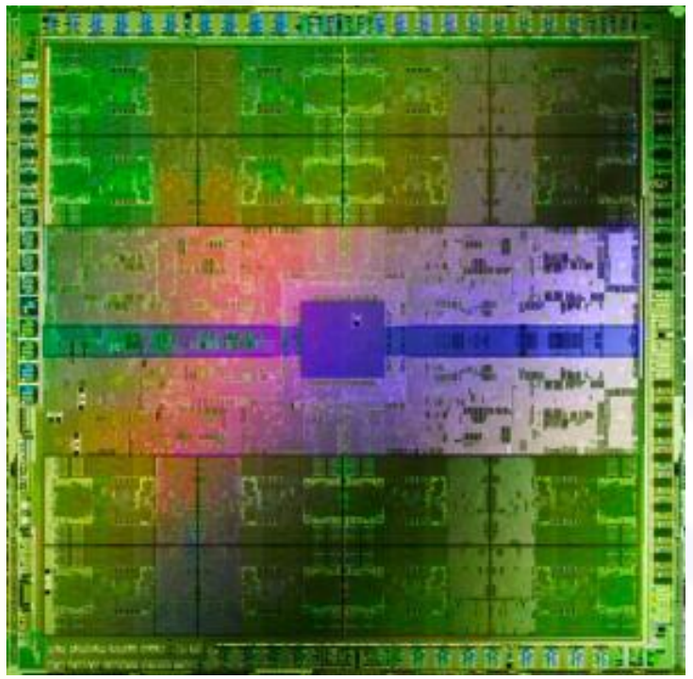
- Multicore: yoke of oxen
  - Each core optimized for executing a single thread
- Manycore: flock of chickens
  - Cores optimized for aggregate throughput, deemphasizing individual performance

Multicore & Manycore, *cont.*

Specifications	Westmere-EP	Fermi (GF110)
Processing Elements	6 cores, 2 issue, 4 way SIMD @3.46 GHz	16 cores, 2 issue, 16 way SIMD @1.54 GHz
Resident Strands/ Threads (max)	6 cores, 2 threads, 4 way SIMD: 48 strands	16 cores, 48 SIMD vectors, 32 way SIMD: 24576 threads
SP GFLOP/s	166	1577
Memory Bandwidth	32 GB/s	192 GB/s
Register File	6 kB (?)	2 MB
Local Store/L1 Cache	192 kB	1024 kB
L2 Cache	1536 kB	0.75 MB
L3 Cache	12 MB	-



Westmere-EP (32nm)



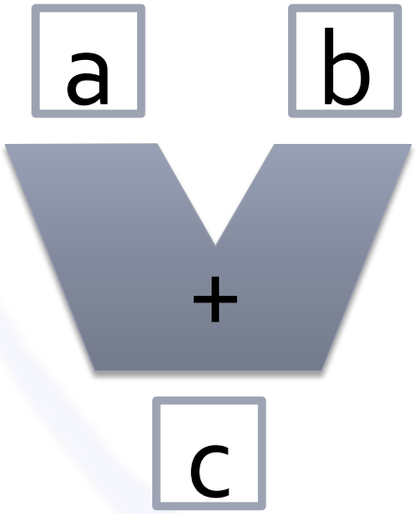
Fermi (40nm)

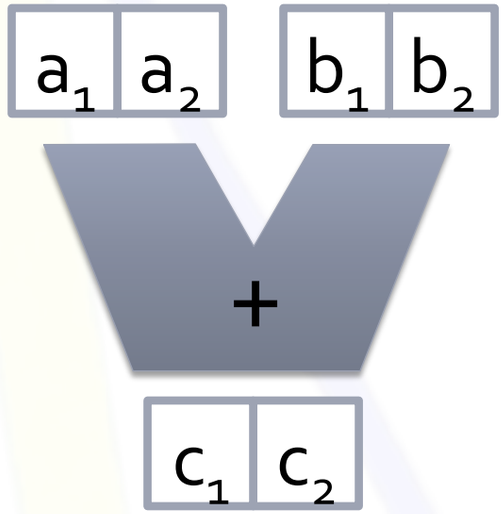
What is a core?

- Is a core an ALU?
  - ATI: We have 1600 streaming processors!!
    - Actually, we have 5 way VLIW * 16 way SIMD * 20 "SIMD cores"
- Is a core a SIMD vector unit?
  - Nvidia: We have 512 streaming processors!!
    - Actually, we have 16 way SIMD * 16 "multiprocessors" * 2-way SMT
- In this lecture, we're using core consistent with the CPU world
  - Superscalar, VLIW, SIMD, SMT are part of a core's architecture, not the number of cores

SIMD

SISD





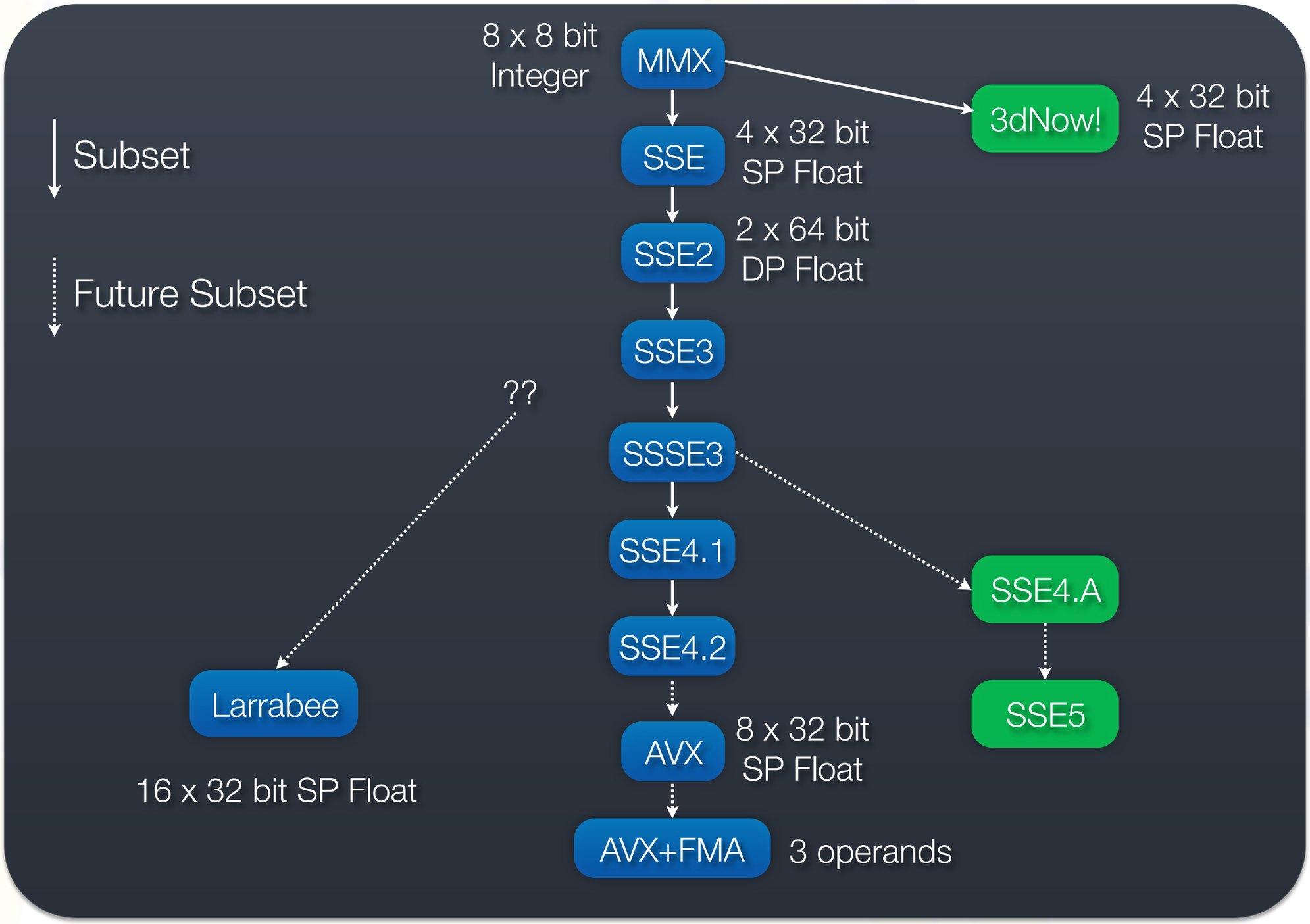
SIMD width=2

- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

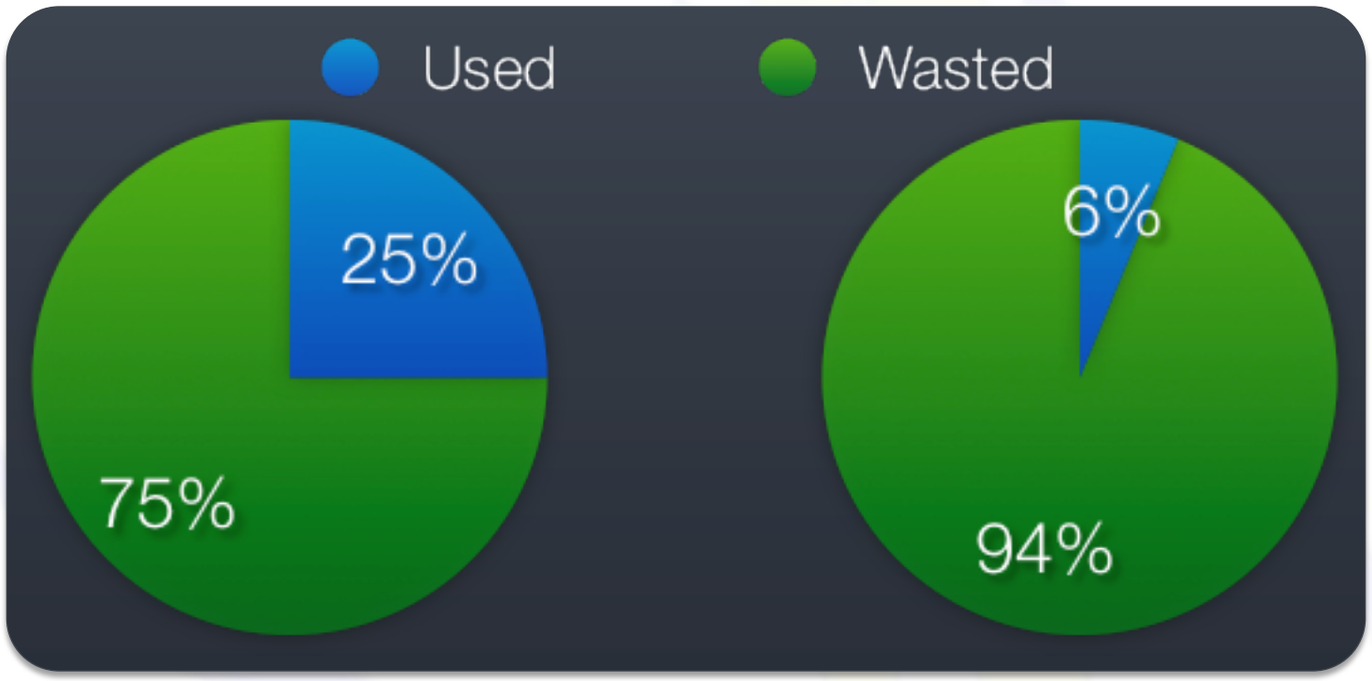
SIMD: Neglected Parallelism

- It is difficult for a compiler to exploit SIMD
- How do you deal with sparse data & branches?
  - Many languages (like C) are difficult to vectorize
  - Fortran is somewhat better
- Most common solution:
  - Either forget about SIMD
    - Pray the autovectorizer likes you
  - Or instantiate intrinsics (assembly language)
  - Requires a new code version for every SIMD extension

A Brief History of x86 SIMD



What to do with SIMD?



4 way SIMD (SSE) 16 way SIMD (LRB)

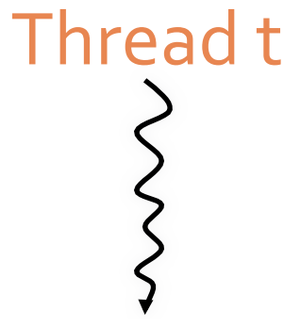
- Neglecting SIMD is becoming more expensive
  - AVX: 8 way SIMD, Larrabee: 16 way SIMD, Nvidia: 32 way SIMD, ATI: 64 way SIMD
- This problem composes with thread level parallelism
- We need a programming model which addresses both problems

The CUDA Programming Model

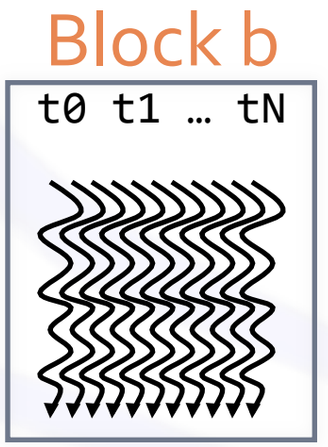
- CUDA is a recent programming model, designed for
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability
- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchronization & data sharing between small groups of threads
- CUDA programs are written in C + extensions
- OpenCL is inspired by CUDA, but HW & SW vendor neutral
  - Similar programming model

Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program



- Threads are grouped into thread blocks
  - threads in the same block can cooperate



- Threads/blocks have unique IDs

What is a CUDA Thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled

- CUDA threads might be physical threads
  - as on NVIDIA GPUs

- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU

What is a CUDA Thread Block?

- Thread block = virtualized multiprocessor
  - freely choose processors to fit data
  - freely customize for each kernel launch

- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want

- Thread blocks of kernel must be independent tasks
  - program valid for ***any interleaving*** of block executions

Mapping back

- Thread parallelism
  - each thread is an independent thread of execution

- Data parallelism
  - across threads in a block
  - across blocks in a kernel

- Task parallelism
  - different blocks are independent
  - independent kernels

Synchronization

- Threads within a block may synchronize with barriers

```
… Step 1 …
__syncthreads();
… Step 2 …
```

- Blocks coordinate via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`

- Implicit barrier between dependent kernels

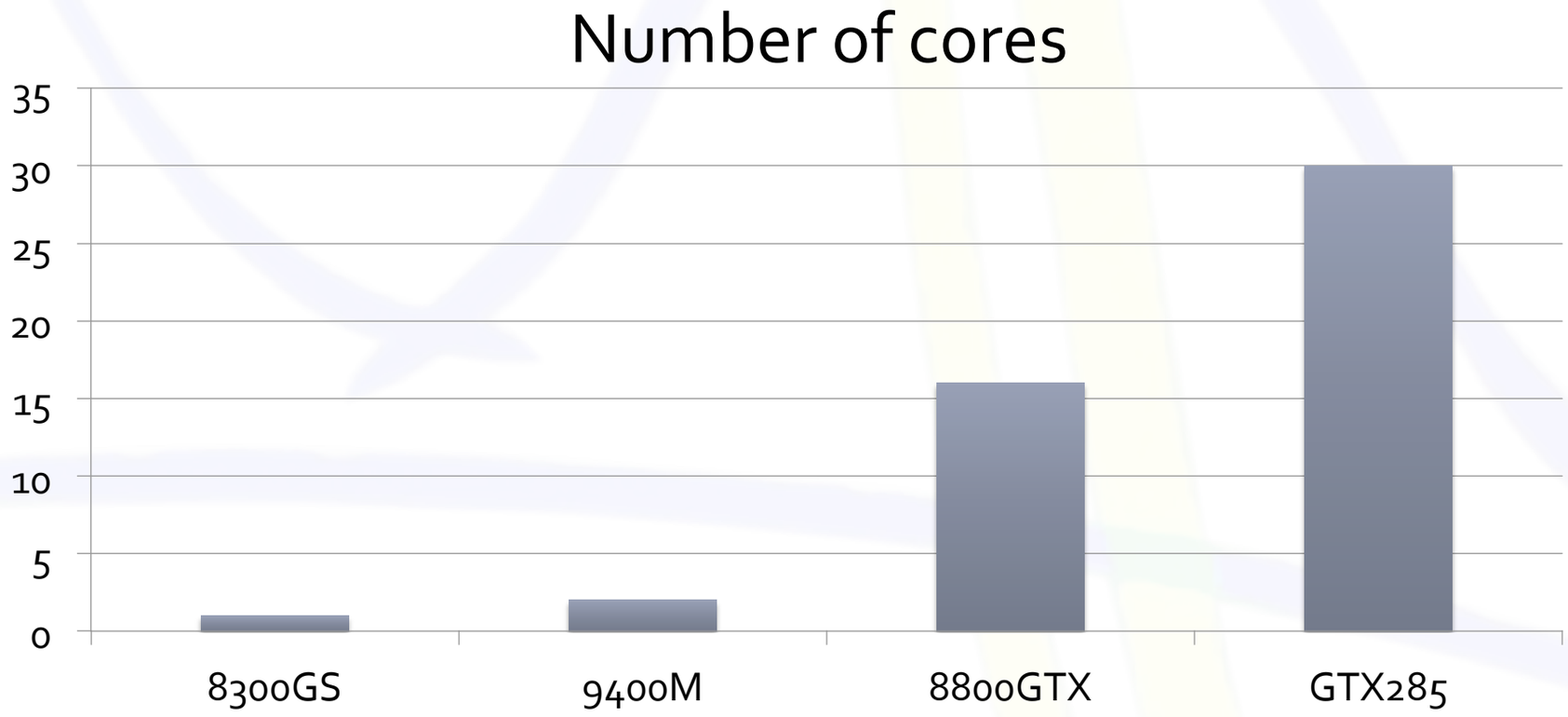
```
vec_minus<<<nblocks, blksize>>>(a, b, c);
- - - - - - - - - - - - - - - - - - - - - - - - - -
vec_dot<<<nblocks, blksize>>>(c, c);
```

Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement gives scalability

Scalability

- Manycore chips exist in a diverse set of configurations



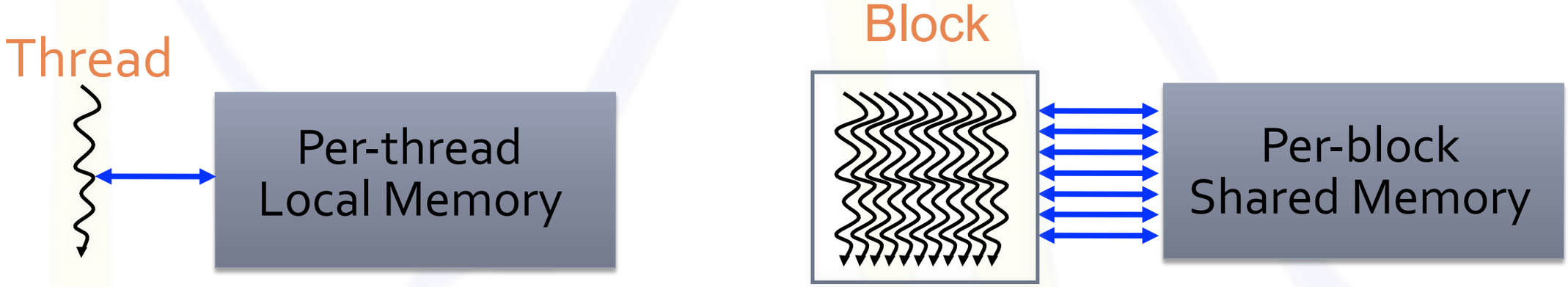
- CUDA allows one binary to target all these chips
- Thread blocks bring scalability!

Hello World: Vector Addition

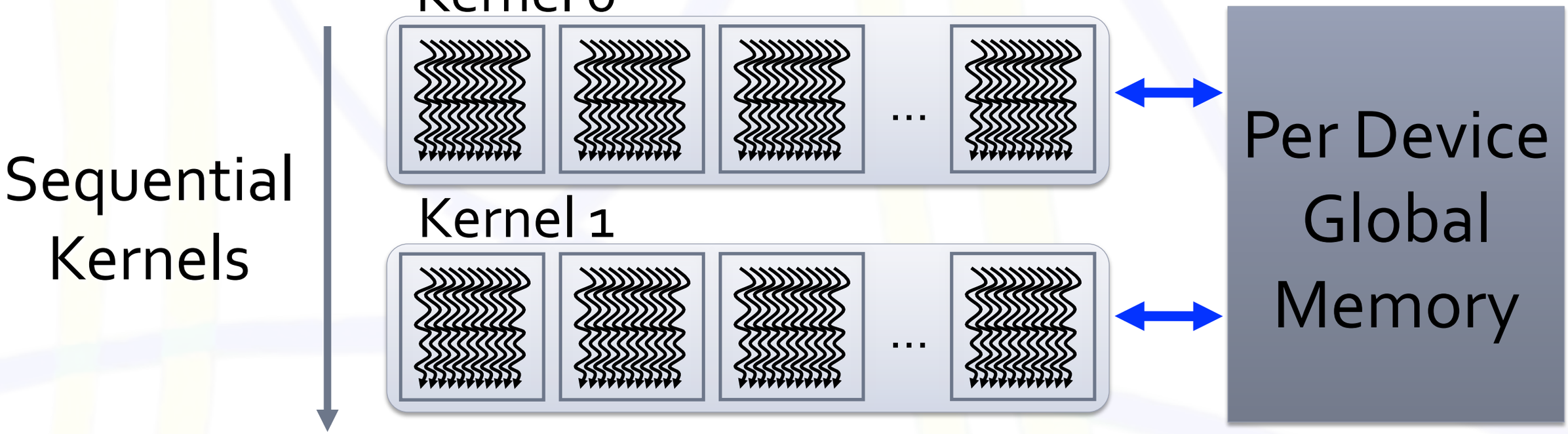
```
//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   c[i] = a[i] + b[i];
}

int main() {
   //Run N/256 blocks of 256 threads each
   vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
```

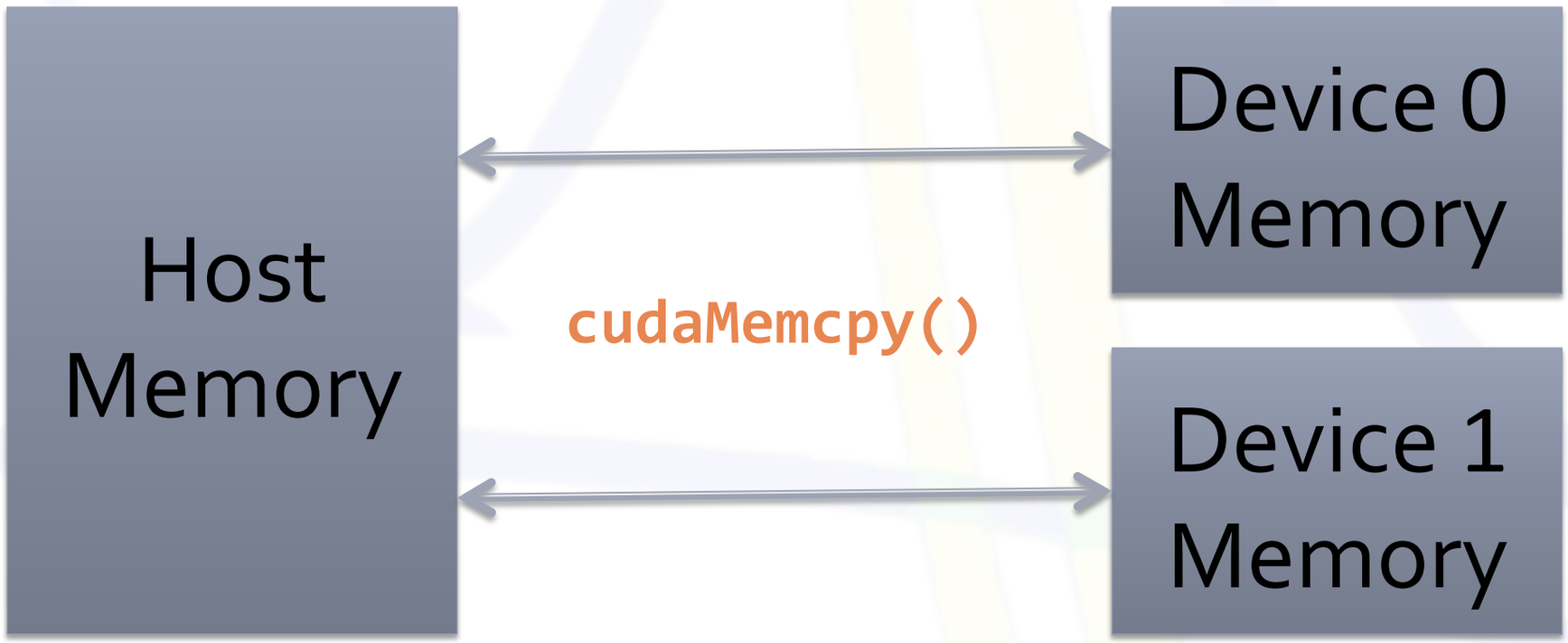
Memory model



Memory model



Memory model



Hello World: Managing Data

```
int main() {
   int N = 256 * 1024;
   float* h_a = malloc(sizeof(float) * N);
   //Similarly for h_b, h_c. Initialize h_a, h_b

   float *d_a, *d_b, *d_c;
   cudaMalloc(&d_a, sizeof(float) * N);
   //Similarly for d_b, d_c

   cudaMemcpy(d_a, h_a, sizeof(float) * N, cudaMemcpyHostToDevice);
   //Similarly for d_b

   //Run N/256 blocks of 256 threads each
   vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);

   cudaMemcpy(h_c, d_c, sizeof(float) * N, cudaMemcpyDeviceToHost);
}
```

Using per-block shared memory

- Variables shared across block

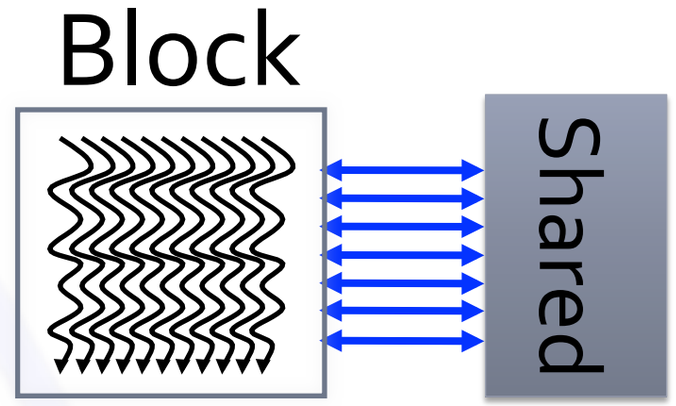
```
__shared__ int *begin, *end;
```

- Scratchpad memory

```
__shared__ int scratch[BLOCKSIZE];
scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

```
scratch[threadIdx.x] = begin[threadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```



- Per-block shared memory is faster than L1 cache, slower than register file
- It is relatively small: register file is 2-4x larger

CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   //  kernel callable from host
__device__ void DeviceFunc(...);   //  function callable on device
__device__ int  GlobalVar;         //  variable in device memory
__shared__ int  SharedVar;         //  in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);     //  500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                   //  barrier synchronization
```

CUDA: Features available on GPU

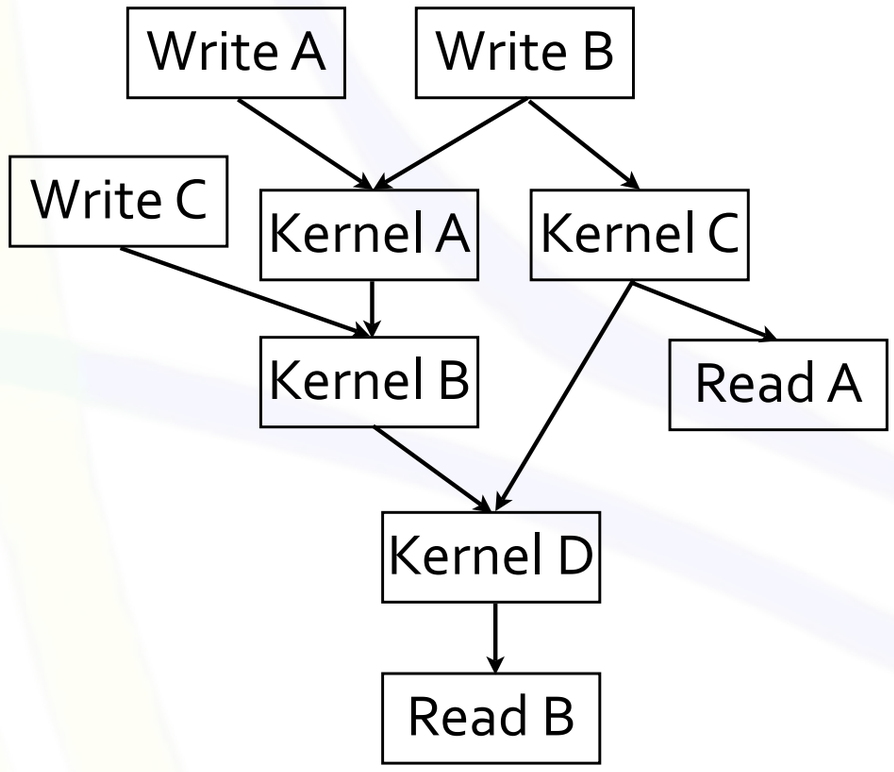
- Double and single precision
- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.
- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`
- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...
- Texture management
  - `cudaBindTexture()`, `cudaBindTextureToArray()`, ...
- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...

OpenCL

- OpenCL is supported by AMD {CPUs, GPUs} and Nvidia
  - Intel, Imagination Technologies (purveyor of GPUs for iPhone/Nexus/etc.) are also on board
- OpenCL's data parallel execution model mirrors CUDA, but with different terminology
- OpenCL has richer task parallelism model
- Runtime walks a dataflow DAG of kernels/memory transfers



OpenCL and SIMD

- SIMD issues are handled separately by each runtime
- AMD GPU
  - Vectorize over 64-way SIMD, but not over 5-way VLIW
    - Use float4 vectors in your code
- AMD CPU
  - No vectorization
    - Use float4 vectors in your code (float8 when AVX appears?)
- Nvidia GPU
  - Full vectorization, like CUDA
    - No need to use float4 vectors

Mapping CUDA to Nvidia GPUs

- CUDA is designed to be functionally forgiving
  - First priority: make things work. Second: get performance.
- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs
- Threads:
  - each thread is a SIMD vector lane
- Warps:
  - A SIMD instruction acts on a "warp"
  - Warp width is 32 elements: ***LOGICAL*** SIMD width
- Thread blocks:
  - Each thread block is scheduled onto a processor
  - Peak efficiency requires multiple thread blocks per processor

Mapping CUDA to a GPU, *continued*

- The GPU is very deeply pipelined
  - Throughput machine, trying to hide memory latency
- This means that performance depends on the number of thread blocks which can be allocated on a processor
- Therefore, resource usage costs performance:
  - More registers => Fewer thread blocks
  - More shared memory usage => Fewer thread blocks
- It is often worth trying to reduce register count in order to get more thread blocks to fit on the chip
  - For Fermi, target 20 registers or less per thread

Occupancy (Constants for GF100)

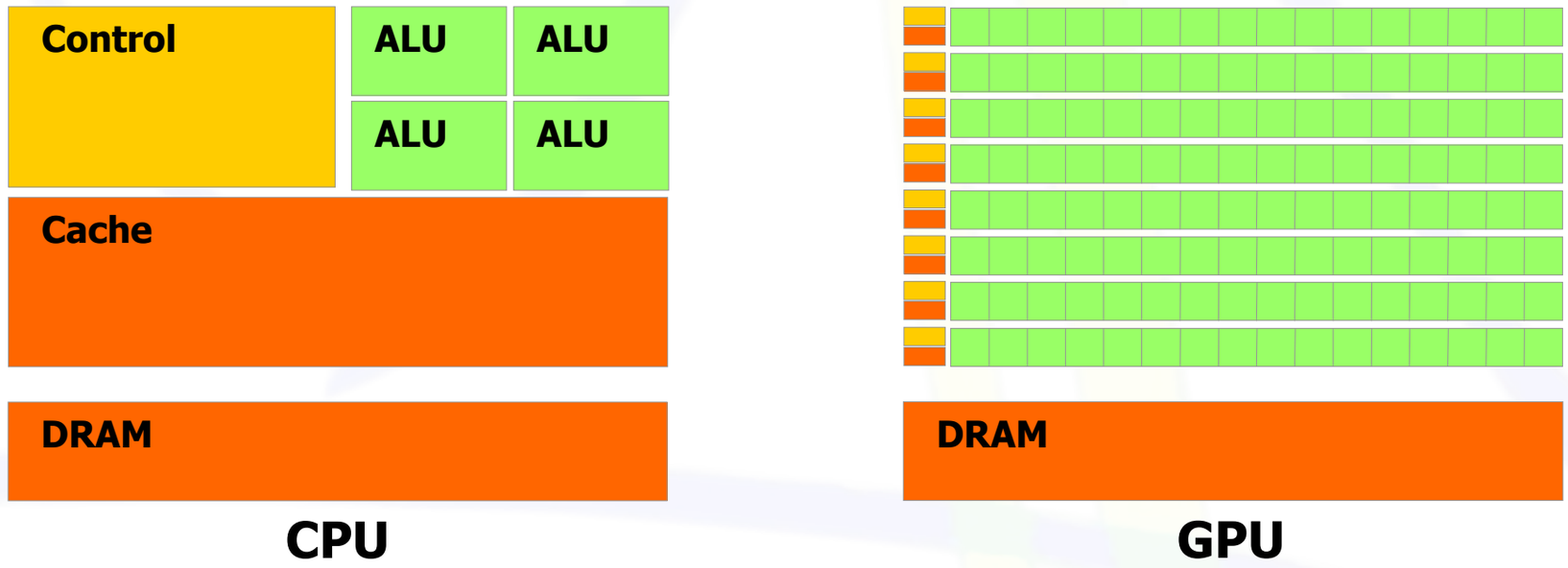
- The GPU tries to fit as many thread blocks simultaneously as possible on to a core (SM)
  - The number of simultaneous thread blocks (B) is ≤ 8
- The number of warps per thread block (T) ≤ 32
- B * T ≤ 48
- The number of threads per warp (V) is 32
- B * T * V * Registers per thread ≤ 32768
- B * Shared memory (bytes) per block ≤ 49152/16384
  - Depending on Shared memory/L1 cache configuration
- Occupancy is reported as B * T / 48

SIMD & Control Flow

- Nvidia GPU hardware handles control flow divergence and reconvergence
  - Write scalar SIMD code, the hardware schedules the SIMD execution
- One caveat: __syncthreads() can't appear in a divergent path
  - This will cause programs to hang
- Good performing code will try to keep the execution convergent within a warp
  - Warp divergence only costs because of a finite instruction cache

Memory, Memory, Memory

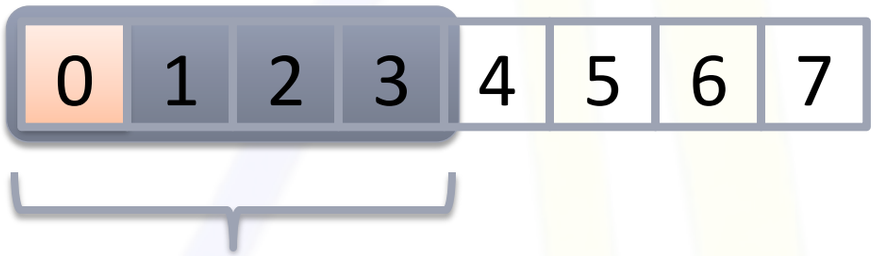
- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem



- Lots of processors, only one socket
- Memory concerns dominate performance tuning

Memory is SIMD too

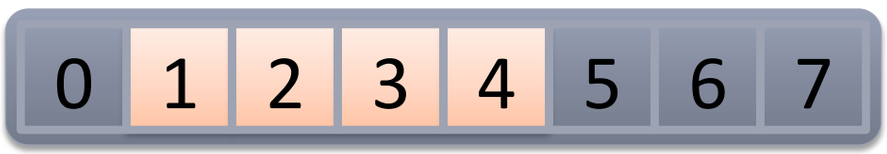
- Virtually all processors have SIMD memory subsystems



- This has two effects:
  - Sparse access wastes bandwidth

2 words used, 8 words loaded:
¼ effective bandwidth

  - Unaligned access wastes bandwidth

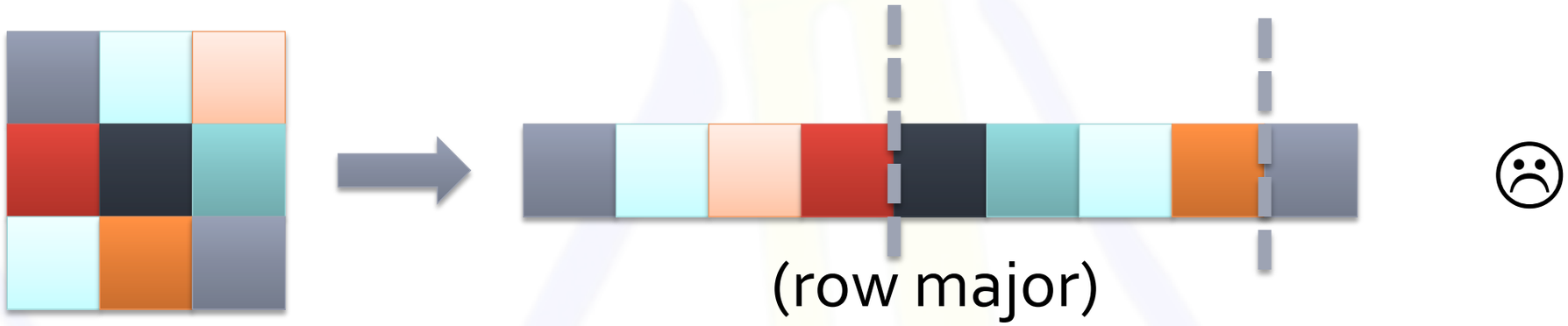


4 words used, 8 words loaded:
½ effective bandwidth

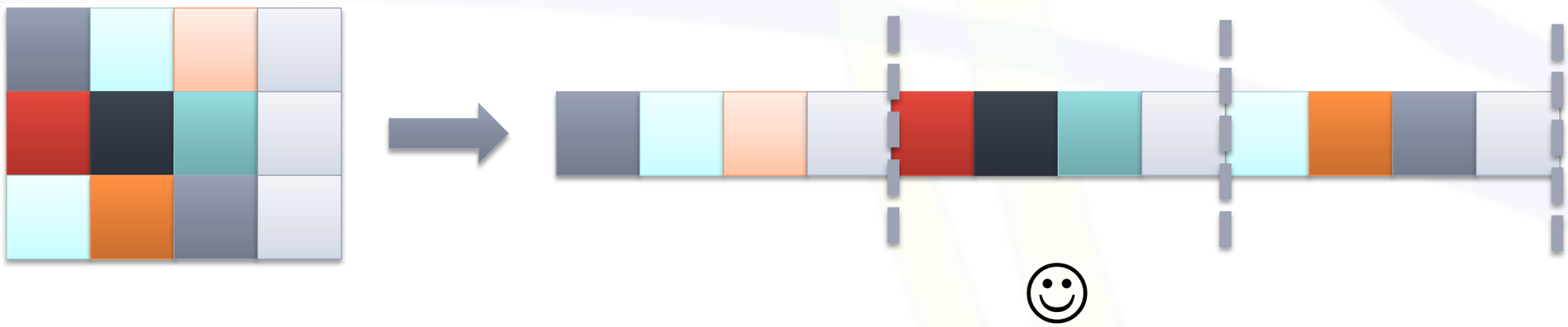
Coalescing

- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests ("cache line")
- GPUs have a "coalescer", which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries

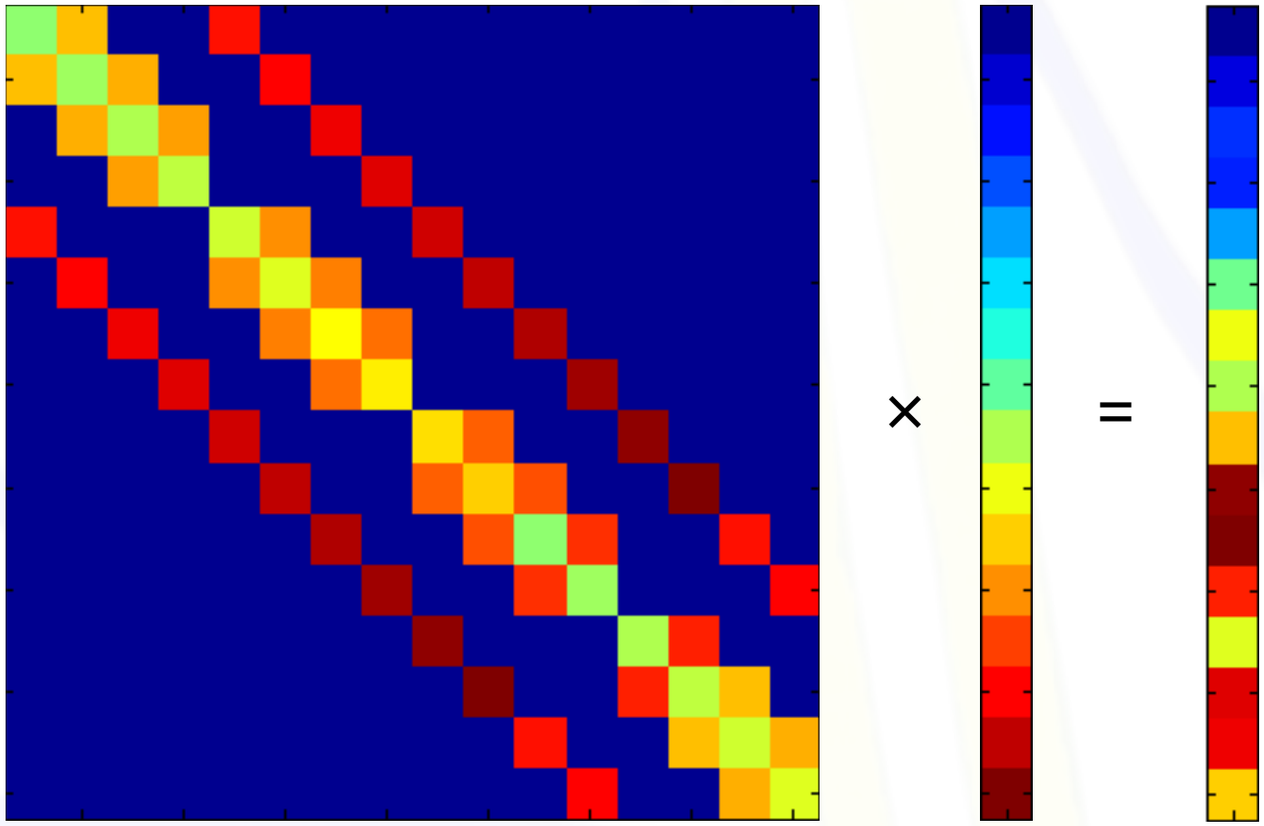
Data Structure Padding



- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern

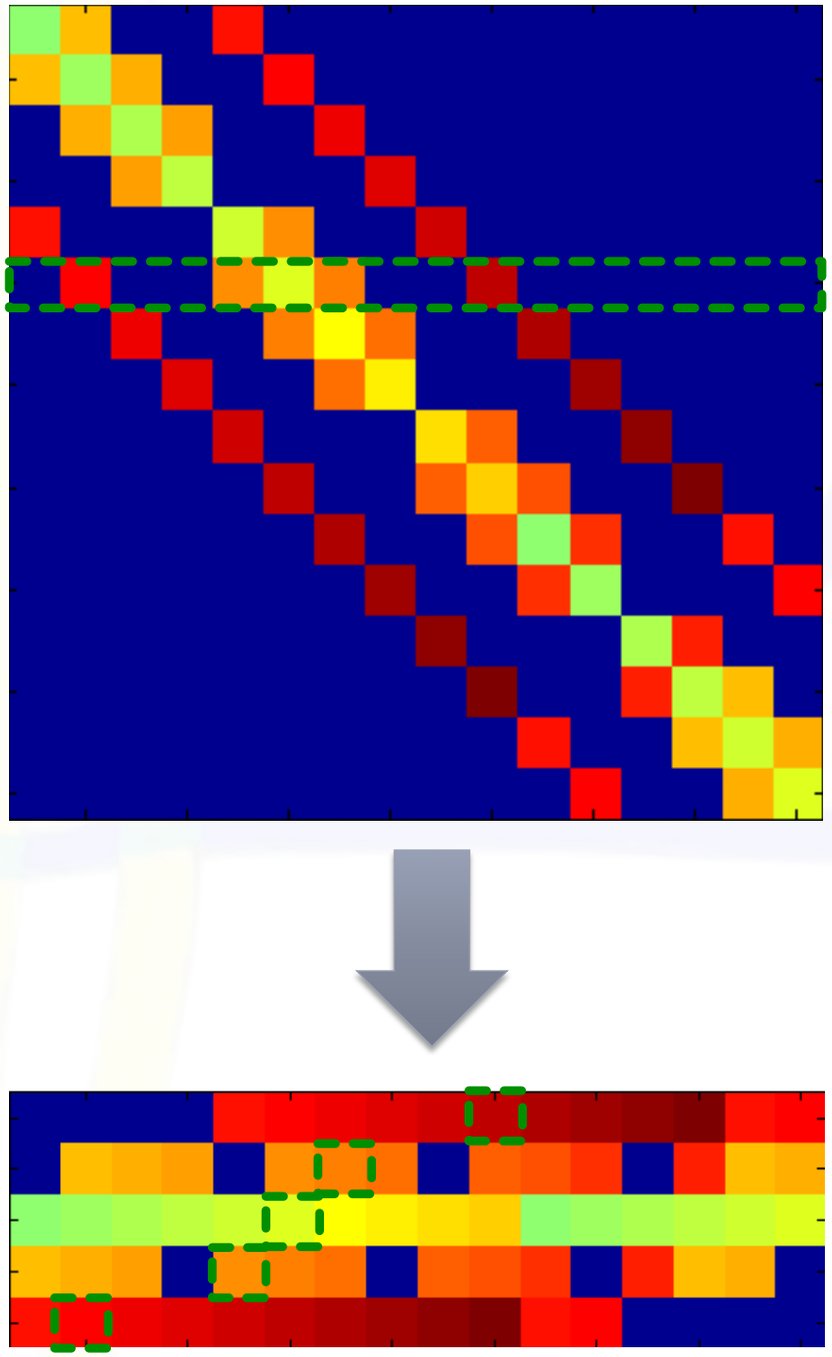


Sparse Matrix Vector Multiply



- Problem: Sparse Matrix Vector Multiplication
- How should we represent the matrix?
  - Can we take advantage of any structure in this matrix?

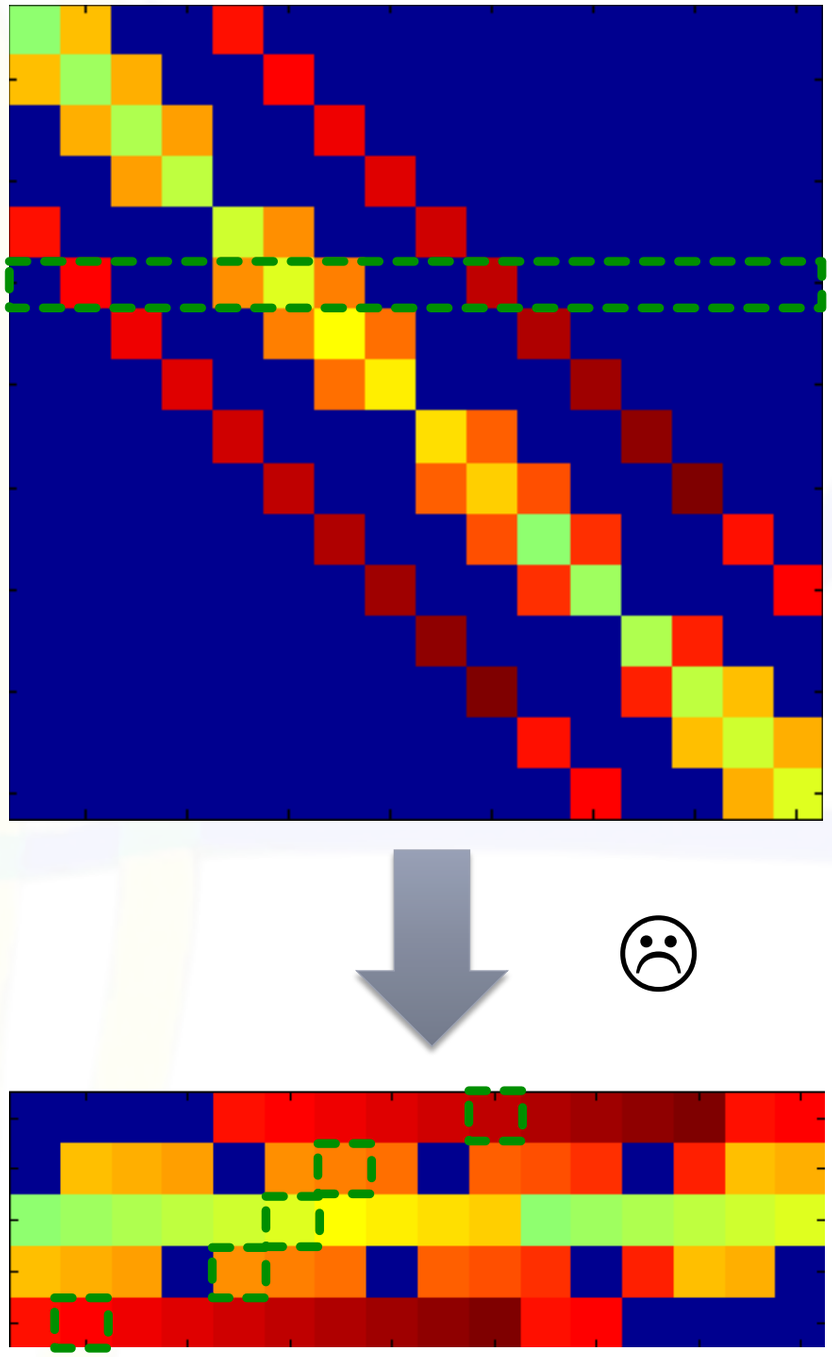
Diagonal representation



- Since this matrix has nonzeros only on diagonals, let's project the diagonals into vectors
- Sparse representation becomes dense
- Launch a thread per row
- Are we done?

- The straightforward diagonal projection is not aligned

Optimized Diagonal Representation



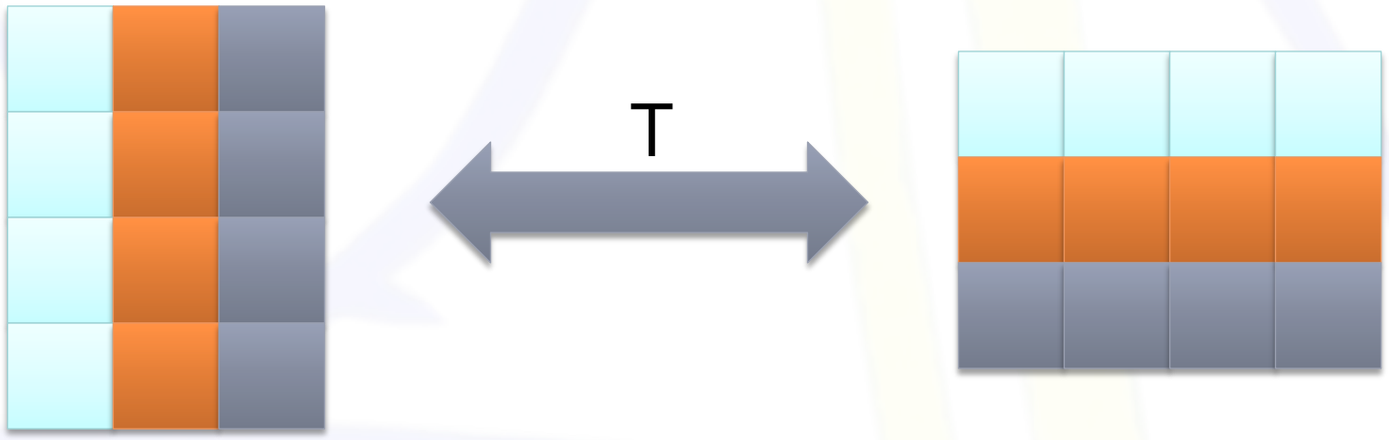
☺

☹

- Skew the diagonals again
- This ensures that all memory loads from matrix are coalesced
- Don’t forget padding!

SoA, AoS

- Different data access patterns may also require transposing data structures



Array of Structs

Structure of Arrays

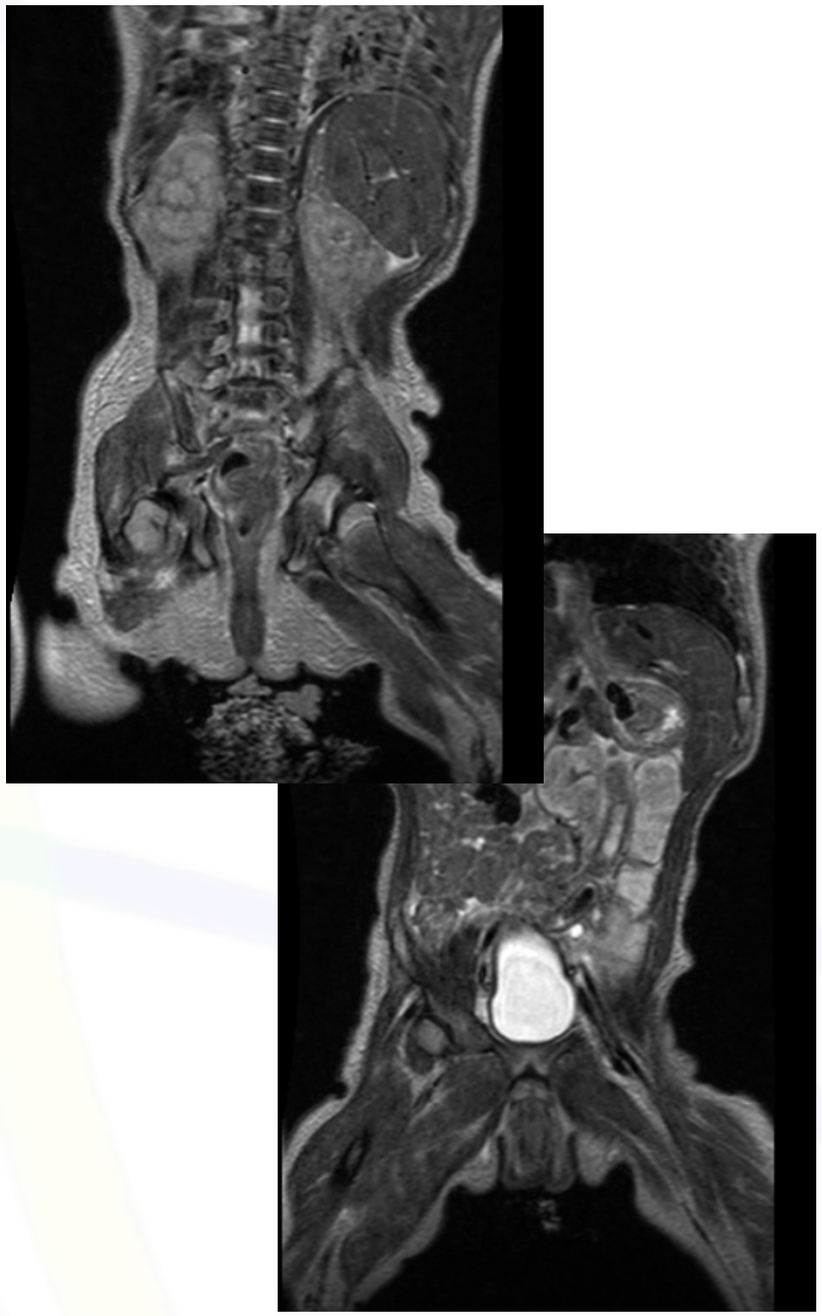
- The cost of a transpose on the data structure is often much less than the cost of uncoalesced memory accesses

Experiences with CUDA

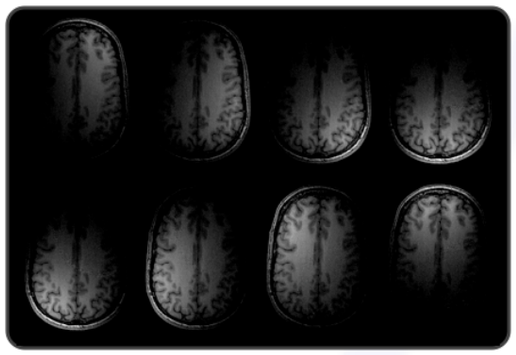
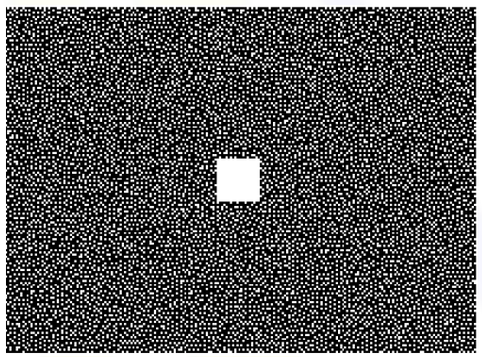
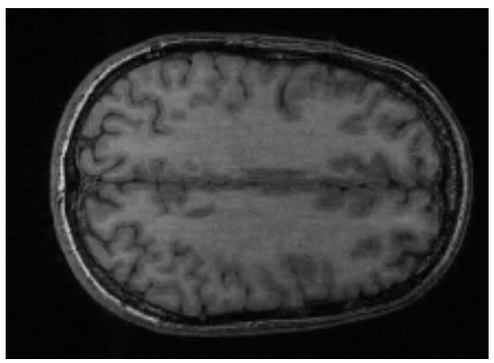
- MRI
- Speech Recognition
- Computational Finance
- Human Detection
- Image Contour Detection
- Support Vector Machines

Fast, Robust Pediatric MRI

- Pediatric MRI is difficult:
  - Children cannot sit still, breathhold
  - Low tolerance for long exams
  - Anesthesia is costly and risky
- Like to accelerate MRI acquisition
  - Advanced MRI techniques exist, but require data- and compute- intense algorithms for image reconstruction
- Reconstruction must be fast, or time saved in accelerated acquisition is lost in computing reconstruction
  - Non-starter for clinical use



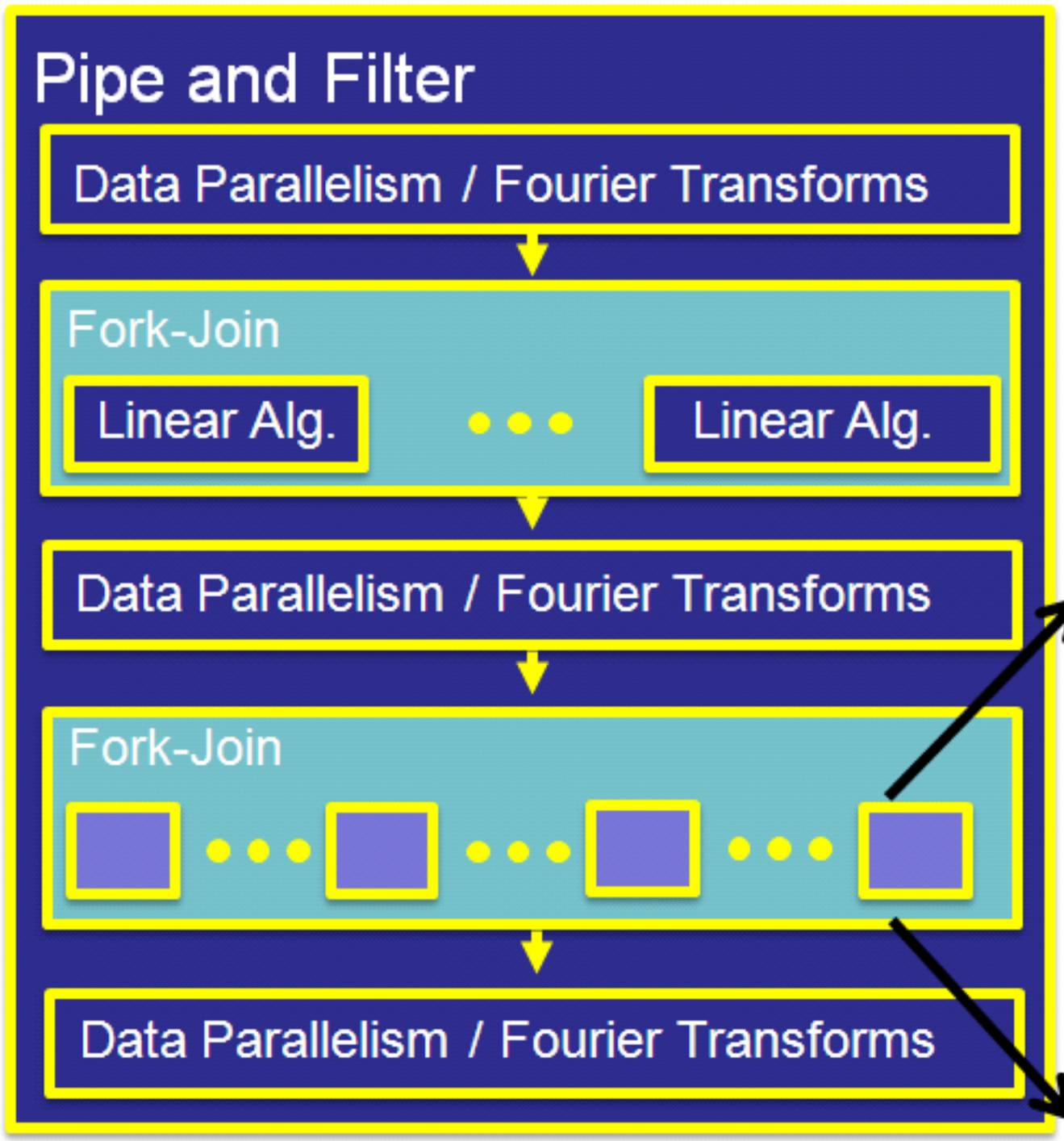
Compressed sensing

- Computational IOU: Must solve constrained L1 minimization

$$\begin{aligned} \text{minimize} \quad & ||Wx||_1 \\ \text{s.t} \quad & \mathbf{F}_\Omega x = y, \\ & ||\mathbf{G}x - x||_2 < \varepsilon \end{aligned}$$

More detail



Iterative POCS Algorithm:

1. Apply SPIRiT Operator:
$$x_c \leftarrow \sum_j g_{cj} * x_j$$
2. Wavelet Soft-Thresholding
$$x \leftarrow W S_\lambda \{W^* x\}$$
3. Fourier-space projection
$$x \leftarrow F(P^T y + P_c^T P_c F^* x)$$

Iter. Refinement / Spectral Method

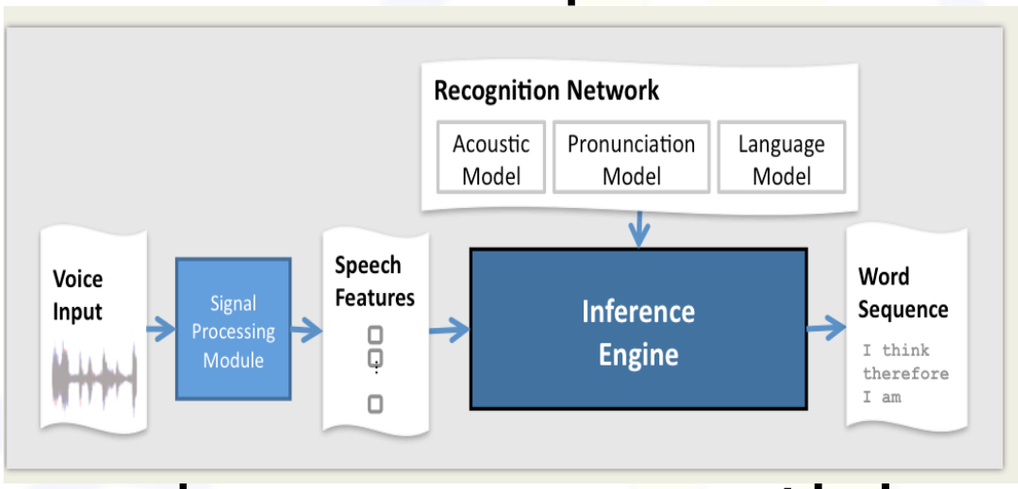
- Data Parallelism / Convolutions
- Data Parallelism / Wavelet xforms
- Data Parallelism / Fourier xforms

Results

- 100X faster reconstruction
- Higher-quality, faster MRI
- This image: 8 month-old patient with cancerous mass in liver
  - 256 x 84 x 154 x 8 data size
  - Serial Recon: 1 hour
  - Parallel Recon: 1 minute
- Fast enough for clinical use
  - Software currently deployed at Lucile Packard Children's Hospital for clinical study of the reconstruction technique

Speech Recognition

- Input: Speech audio waveform
- Output: Recognized word sequences

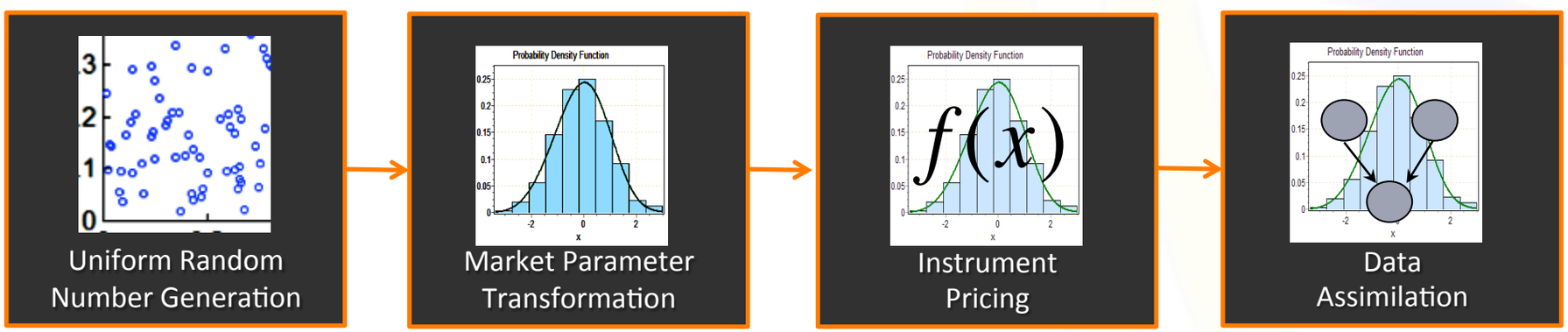


- Achieved 11x speedup over sequential version with same accuracy
- Allows 3.5x faster than real time recognition
- We have deployed this in a hotline call-center data analytics company
- Used to search content, track service quality and provide early detection of service issues

Computational Finance

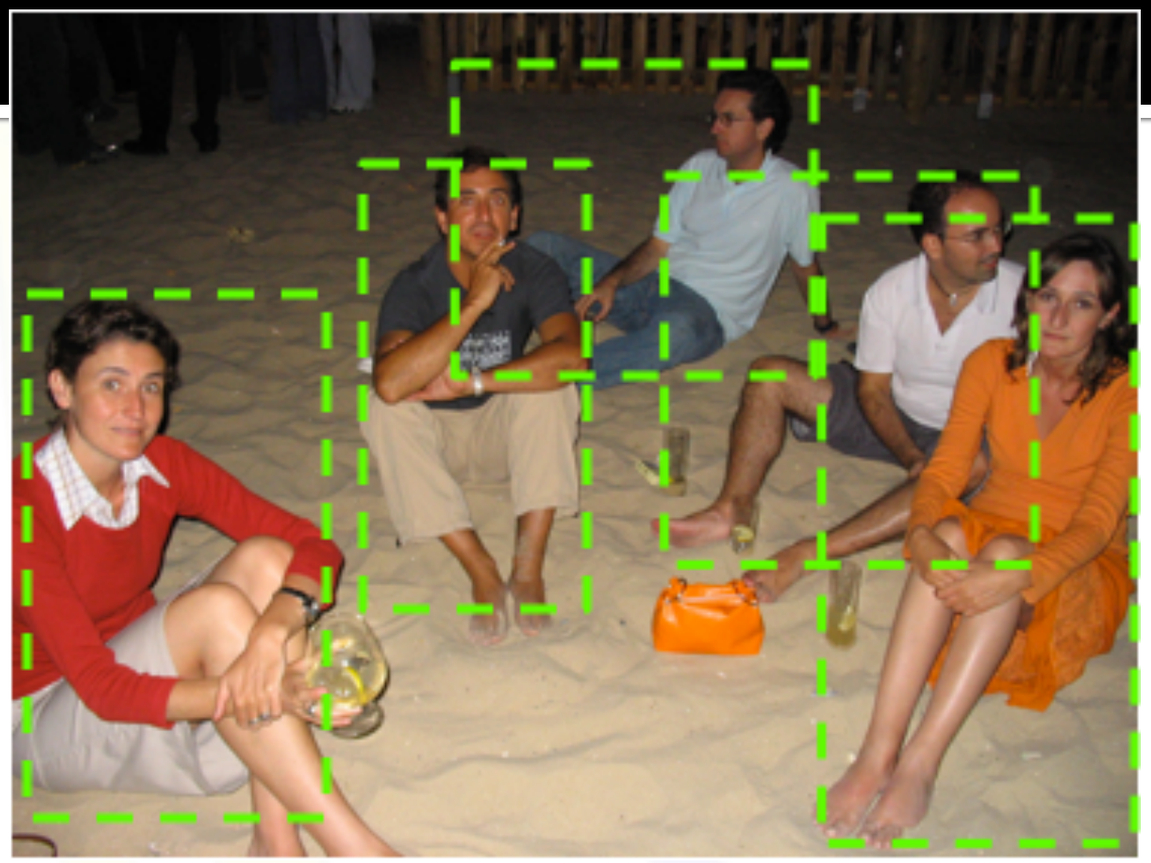
- **Value-at-Risk Computation** with **Monte Carlo Method**
- Summarizes a portfolio's vulnerabilities to market movements
- Important to algorithmic trading, derivative usage and highly leveraged hedge funds
- Improved implementation to run **6ox faster** on a parallel microprocessor

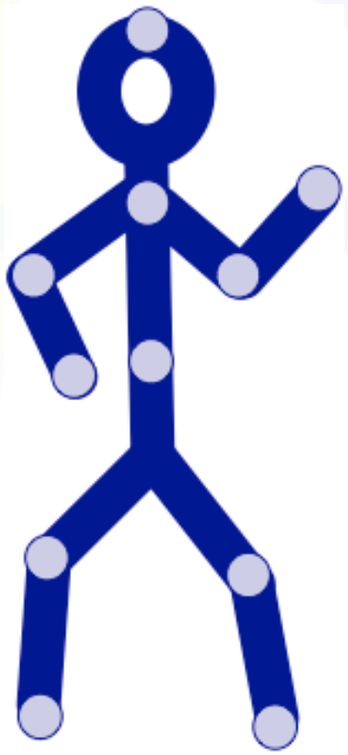
Four Steps of Monte Carlo Method in Finance



Poselet Human Detection

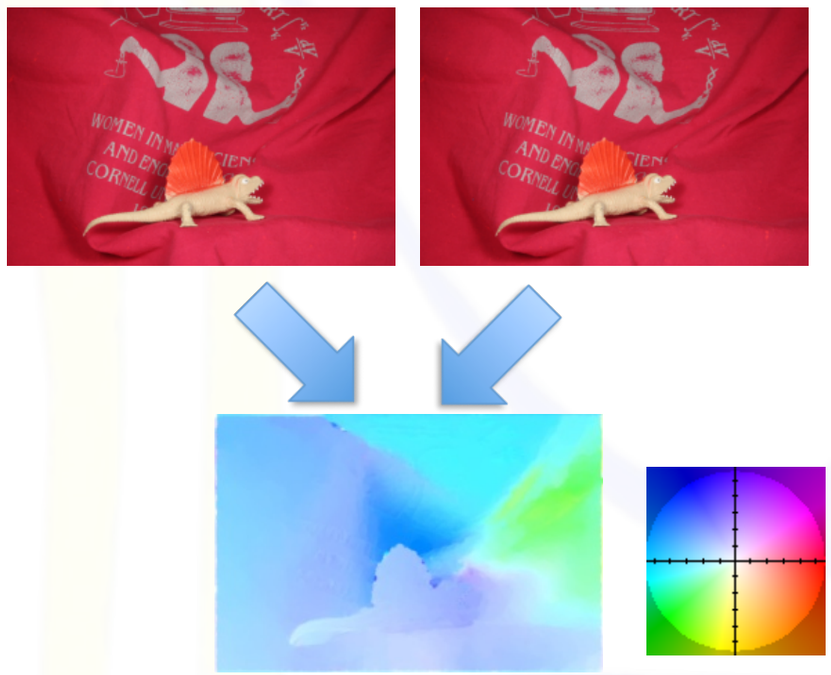
- Can locate humans in images
- **20x speedup** through algorithmic improvements and parallel implementation





- Work can be extended to *pose estimation* for controller-free video game interfaces using ordinary web cameras

Optical Flow



Speedup

32x linear solver
7x overall

- Optical Flow involves computing the motion vectors ("flow field") between the consecutive frames of a video
- Involves solving a non-linear optimization problem

$$J(w) = \int_\Omega \psi_1(|I(x+w) - I(x)|^2)dx +$$
$$\gamma \int_\Omega \psi_2(|\nabla I(x+w) - \nabla I(x)|^2)dx +$$
$$\alpha \int_\Omega \psi_3(|\nabla u|^2 + |\nabla v|^2)dx$$

Image Contours

- Contours are subjective – they depend on personal perspective
- Surprise: Humans agree (more or less)
- J. Malik's group has developed a "ground truth" benchmark

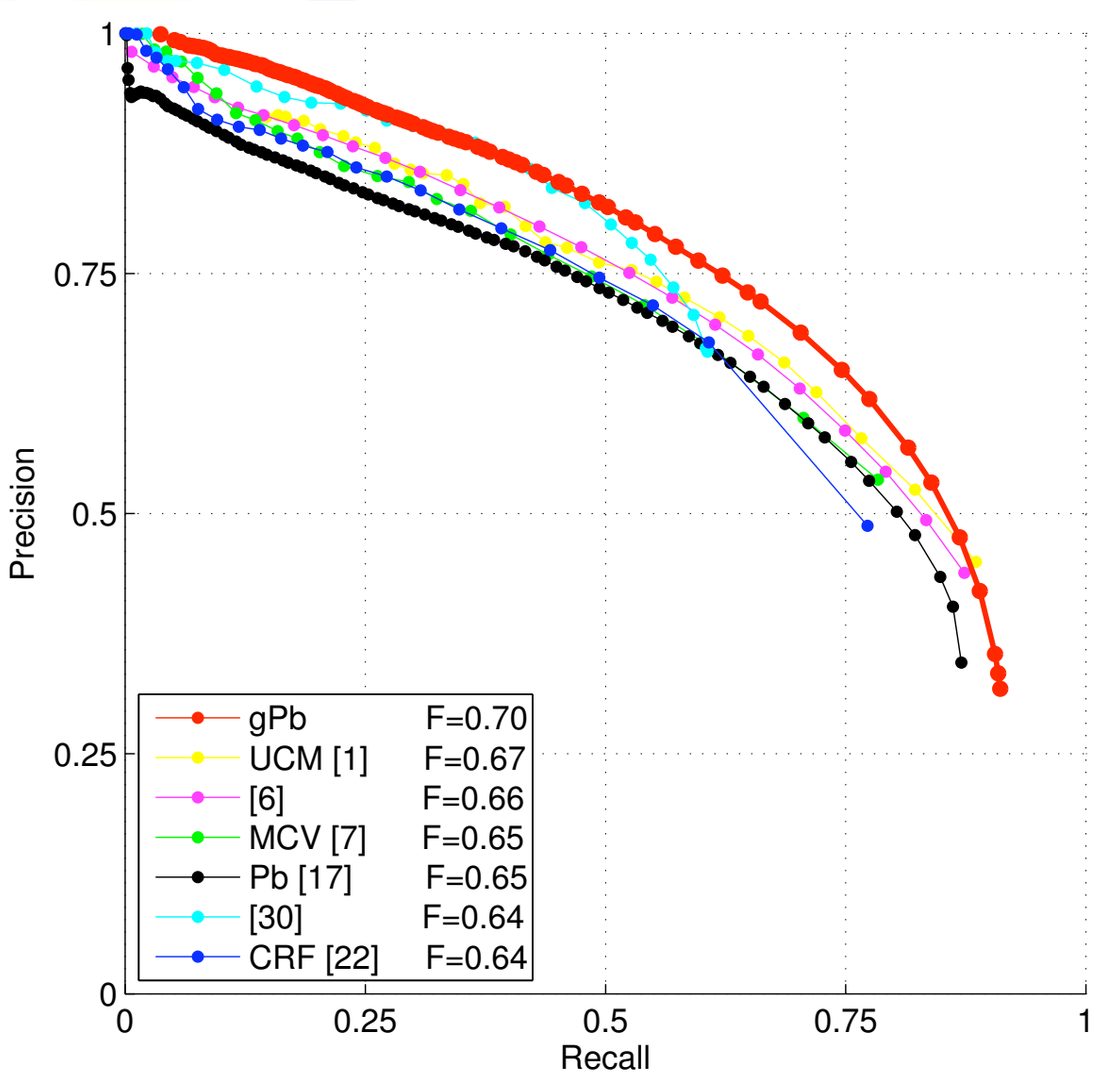
Image

Human Contours

Machine Contours

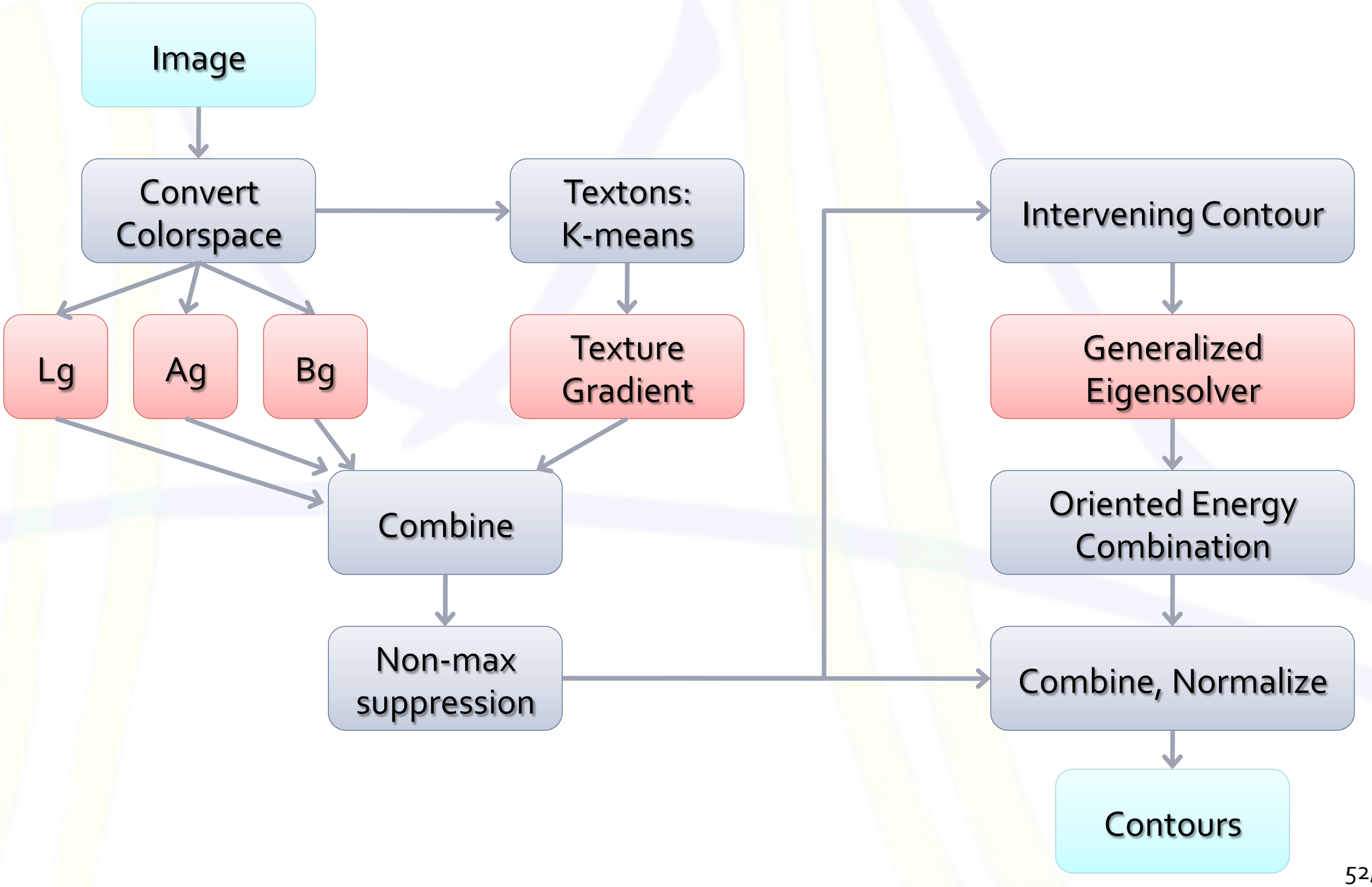
gPb Algorithm: Current Leader

- **g**lobal **P**robability of **b**oundary
- Currently, the most accurate image contour detector
- 7.8 mins per small image (0.15 MP) limits its applicability
  - ~3 billion images on web
  - 10000 computer cluster would take 5 years to find their contours
  - How many new images would there be by then?



Maire, Arbelaez, Fowlkes, Malik, CVPR 2008

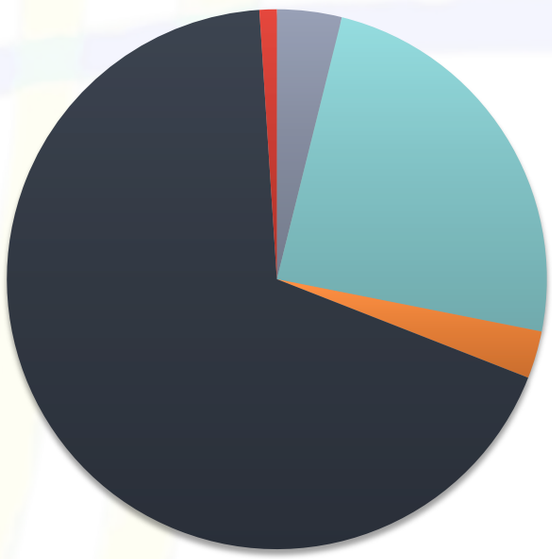
gPb Computation Outline



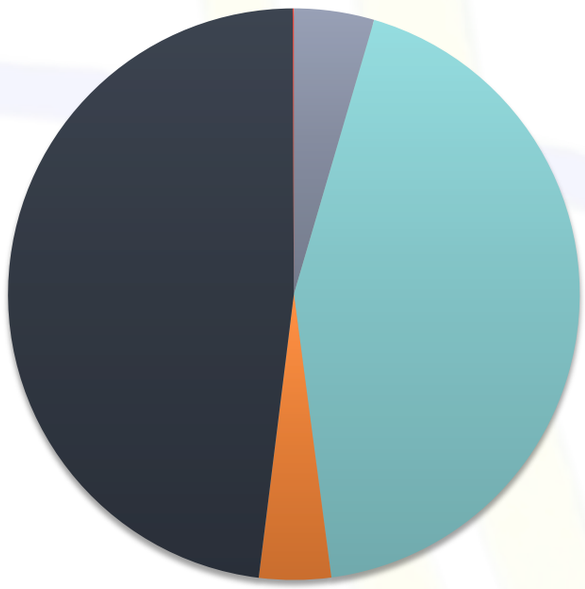
Performance Results

Computation	Original MATLAB/C++	C + Pthreads (8 threads, 2 sockets)	Damascene (GTX280)
Textons	8.6	1.35	0.152
Gradients	53.8	12.92	0.75
Intervening Contour	6.3	1.21	0.03
Eigensolver	151.0	14.29	0.81
Overall	222 seconds	29.79 seconds	1.8 seconds

gPb: CVPR 2008

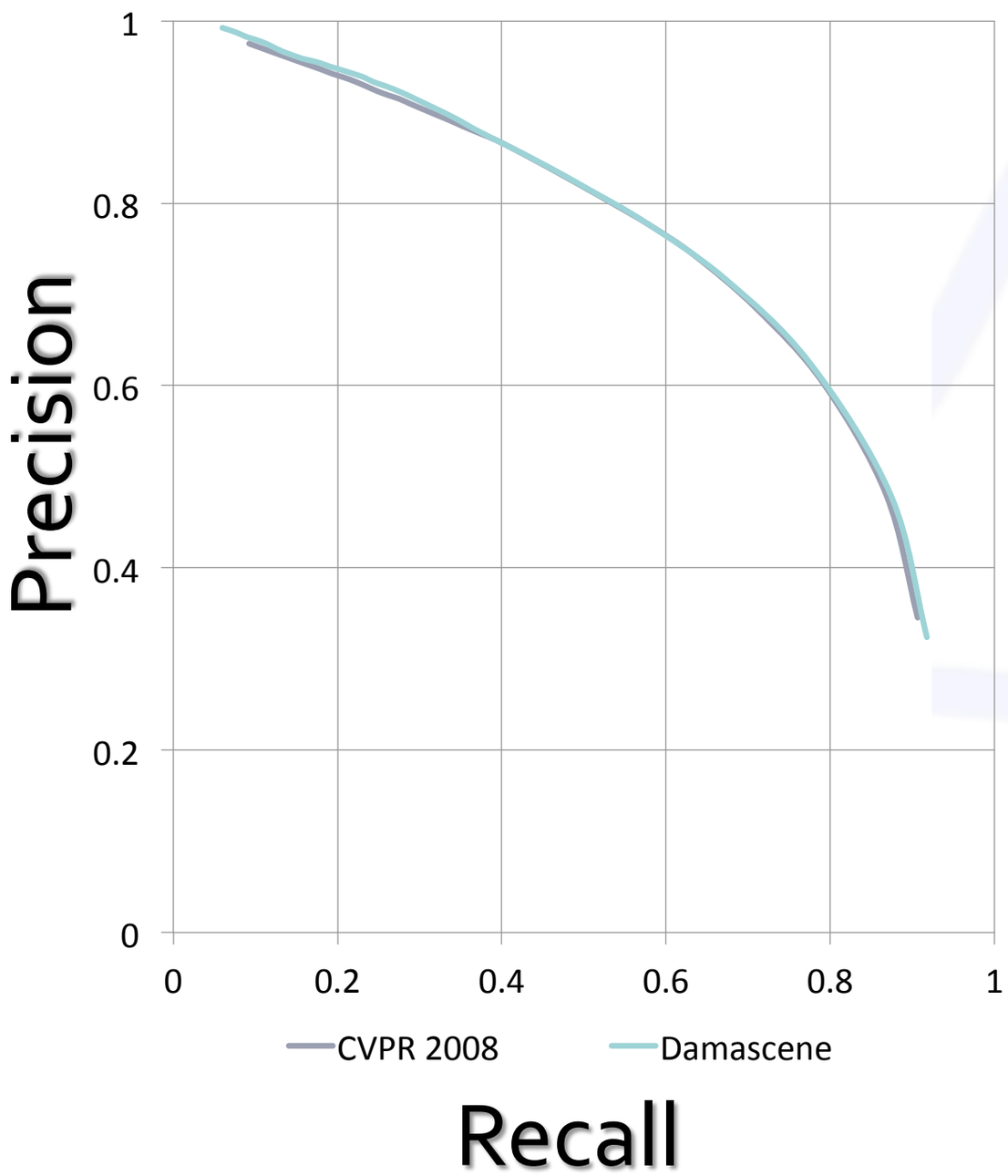


Pthreads



GTX280

Accuracy & Summary



- We achieve equivalent accuracy on the Berkeley Segmentation Dataset
  - Comparing to human segmented "ground truth"
- F-measure 0.70 for both
- Human agreement = 0.79

- 3.7 minutes to 1.8 seconds

SVM Training: Quadratic Programming

Quadratic Program

$$F(\alpha) = \max \sum_{i=1}^{l} \alpha_i - \frac{1}{2}\alpha^T Q \alpha$$

$$s.t. \;\; 0 \leq \alpha_i \leq C, \quad \forall i \in [1, l]$$

$$y^T \alpha = 0$$

$$Q_{ij} = y_i y_j \Phi(x_i, x_j)$$

Variables:

α: Weight for each training point (determines classifier)

Data:

l: number of training points

y: Label (+/- 1) for each training point

x: training points

Example Kernel Functions:

$$\Phi(x_i, x_j) = x_i \cdot x_j$$

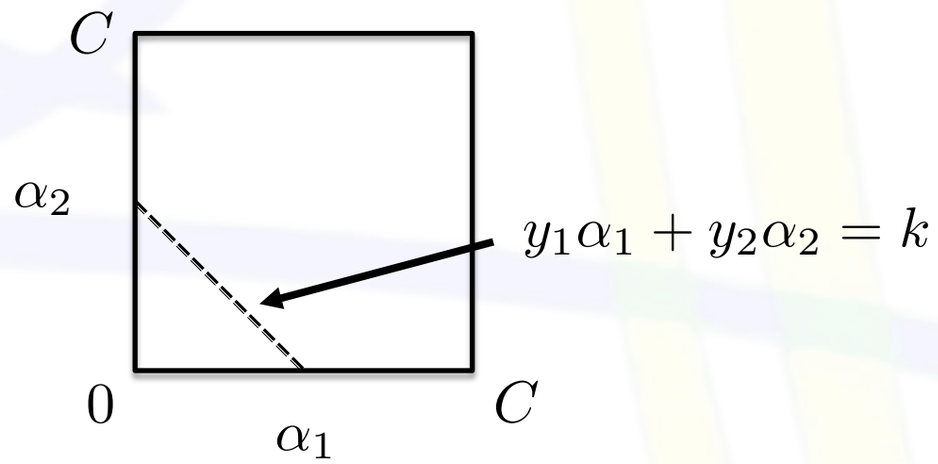
$$\Phi(x_i, x_j; a, r) = \tanh(a x_i \cdot x_j + r)$$

$$\Phi(x_i, x_j; a, r, d) = (a x_j \cdot x_j + r)^d$$

$$\Phi(x_i, x_j; \gamma) = \exp\{-\gamma ||x_i - x_j||^2\}$$

SMO Algorithm

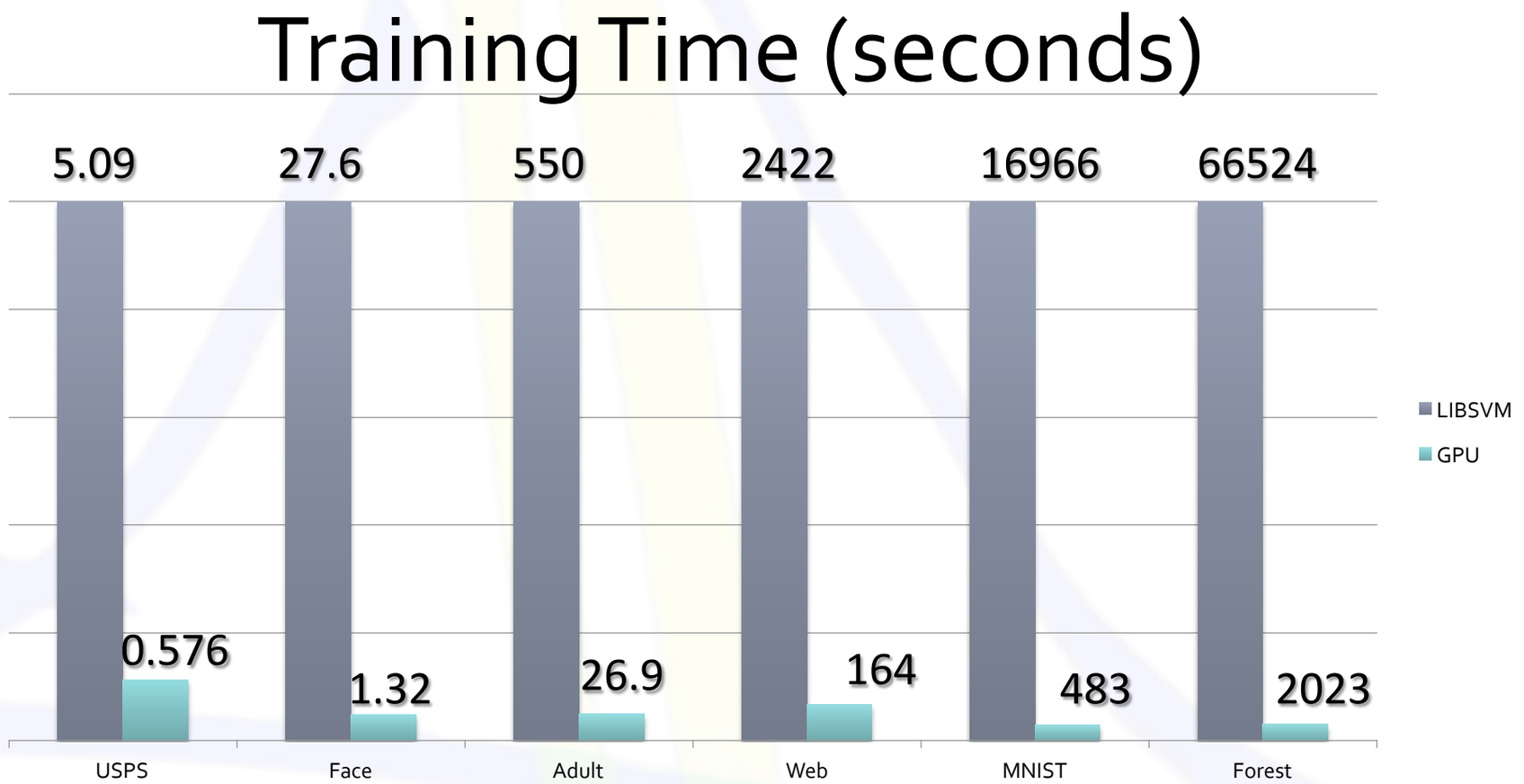
- The Sequential Minimal Optimization algorithm (Platt, 1999) is an iterative solution method for the SVM training problem
- At each iteration, it adjusts only 2 of the variables (chosen by heuristic)
  - The optimization step is then a trivial one dimensional problem:



- Computing full kernel matrix Q not required
- Despite name, algorithm can be quite parallel
- Computation is dominated by KKT optimality condition updates

Training Results

Name	#points	#dim
USPS	7291	256
Face	6977	381
Adult	32561	123
Web	49749	300
MNIST	60000	784
Forest	561012	54



- LibSVM running on Intel Core 2 Duo 2.66 GHz
- Our solver running on Nvidia GeForce 8800GTX
- Gaussian kernel used for all experiments
- 9-35x speedup

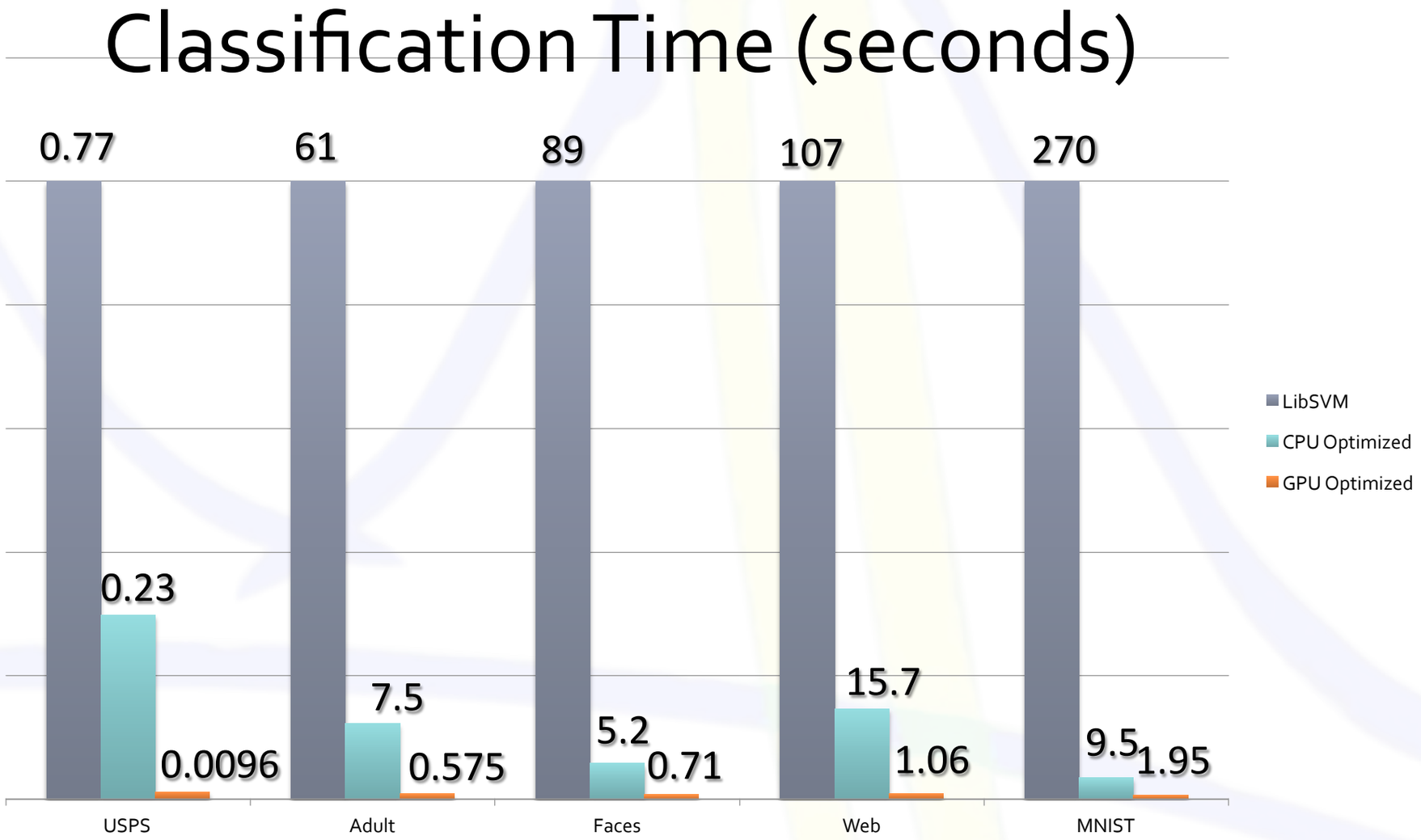
SVM Classification

- To classify a point z, evaluate :

$$\hat{z} = \left\{ b + \sum_{i=1}^{l} y_i \alpha_i \Phi(x_i, z) \right\}$$

- For standard kernels, SVM Classification involves comparing all support vectors and all test vectors with a dot product
- We take advantage of the common situation when one has multiple data points to classify simultaneously
- We cast the dot products as a Matrix-Matrix multiplication, and then use Map Reduce to finish the classification

Classification Results



- CPU optimized version achieves 3-30x speedup
- GPU version achieves an additional 5-24x speedup, for a total of 81-138x speedup
- Results identical to serial version

Summary

- Manycore processors provide useful parallelism
- Programming models like CUDA and OpenCL enable productive parallel programming
- They abstract SIMD, making it easy to use wide SIMD vectors
- CUDA and OpenCL encourages SIMD friendly, highly scalable algorithm design and implementation