CS 677: Parallel Programming for Many-core Processors
Lecture 10

Instructor: Philippos Mordohai

Webpage: mordohai.github.io

E-mail: Philippos.Mordohai@stevens.edu

Logistics

- Project progress reports due next week

1. What is the status of the CPU version? If you are using existing code for this part, cite the source of the code.
2. What is the status of the GPU version in terms of completeness? Which functionalities have been implemented and what is missing?
3. What is the status of the GPU version in terms of correctness? Is the, potentially unoptimized, GPU version correct? If not, what is your plan for achieving correctness?

Outline

- Sparse matrix and vector operations
- Summed area tables
- Parallel Sorting

Sparse Matrix-Vector Multiplication

slides by

Jared Hoberock and David Tarjan

(Stanford CS 193G)

Overview

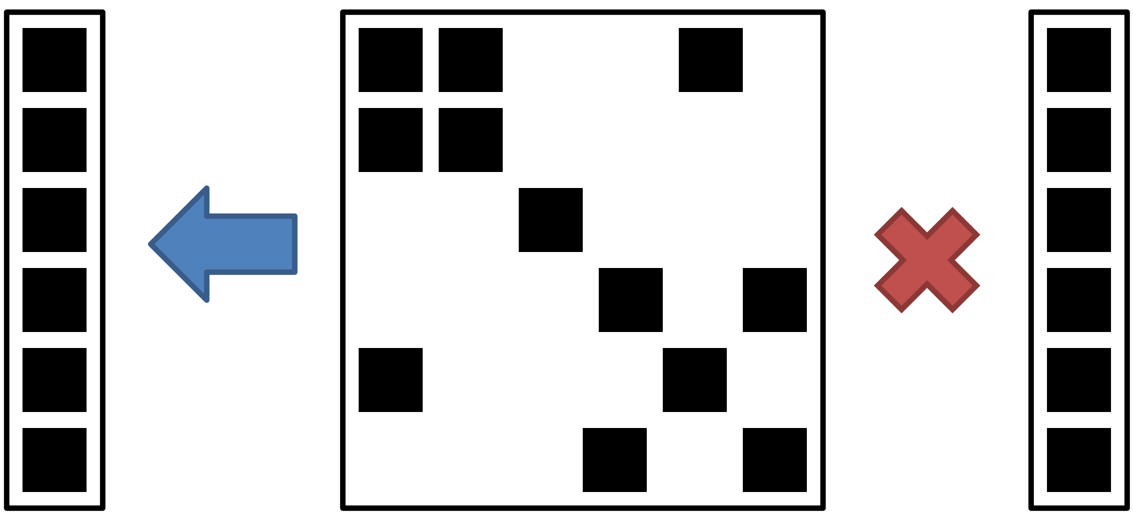
- GPUs deliver high Sparse Matrix Vector (SpMV) performance

- No one-size-fits-all approach
  – Match method to matrix structure

- Exploit structure when possible
  – Fast methods for regular portion
  – Robust methods for irregular portion

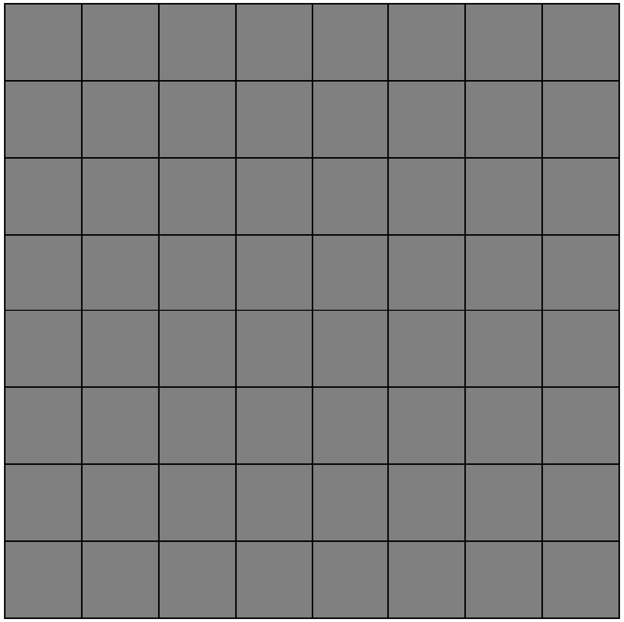
Characteristics of SpMV

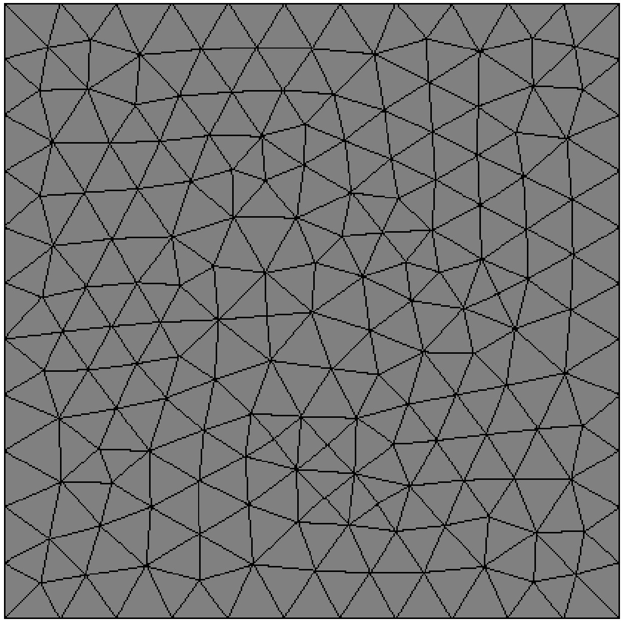
- Memory bound
  - FLOP : MemOp ratio is very low
- Generally irregular & unstructured
  - Unlike dense matrix operations



Finite-Element Methods

- Discretized on structured or unstructured meshes
  - Determines matrix sparsity structure

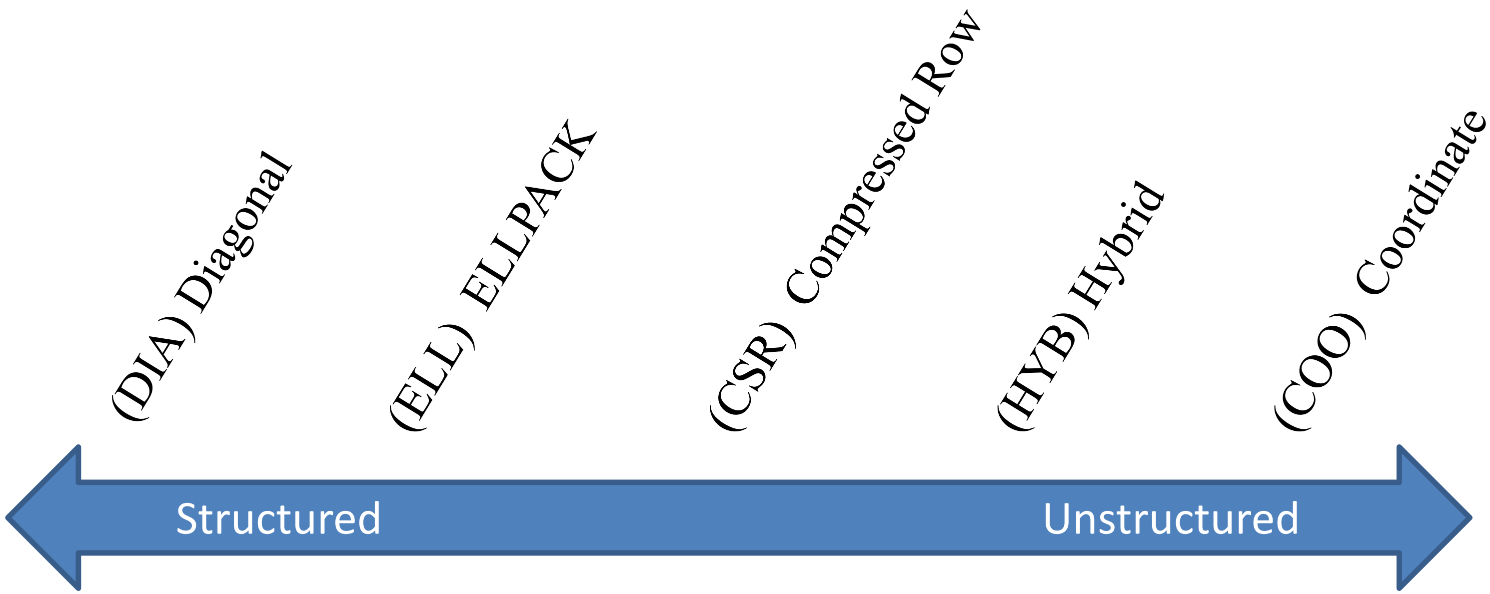


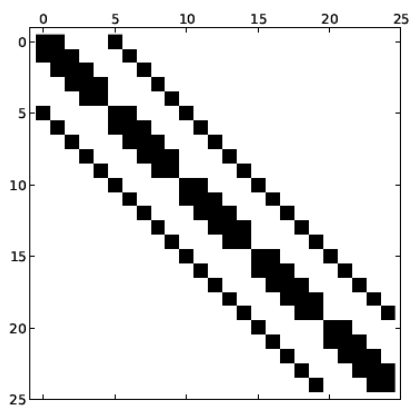


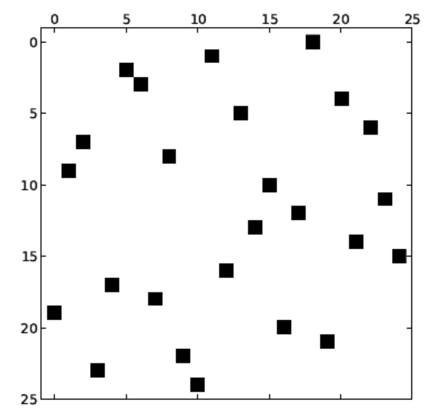
Objectives

- Expose sufficient parallelism
  - Develop 1000s of independent threads
- Minimize execution path divergence
  - SIMD utilization
- Minimize memory access divergence
  - Memory coalescing

Sparse Matrix Formats

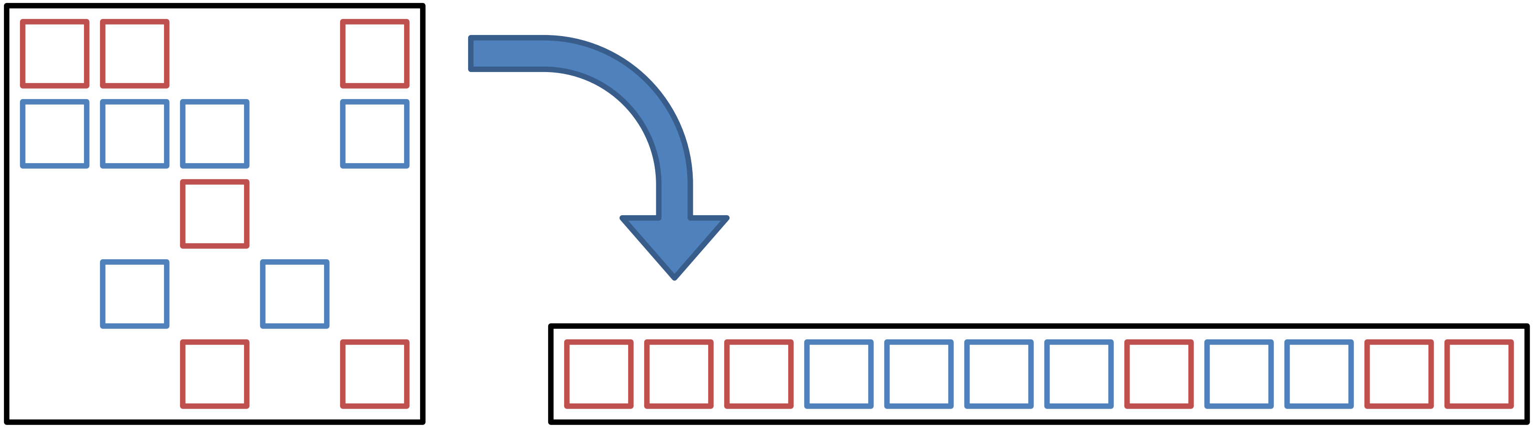






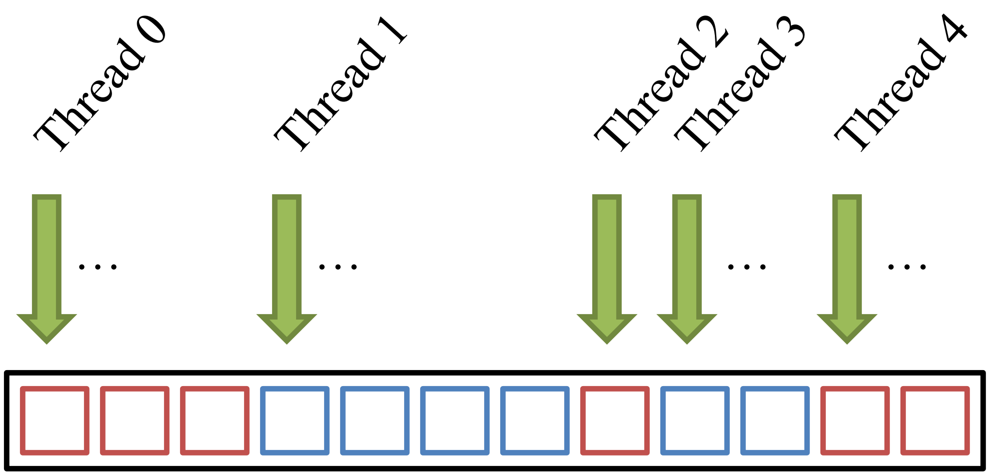
Compressed Sparse Row (CSR)

- Rows laid out in sequence
- Inconvenient for fine-grained parallelism



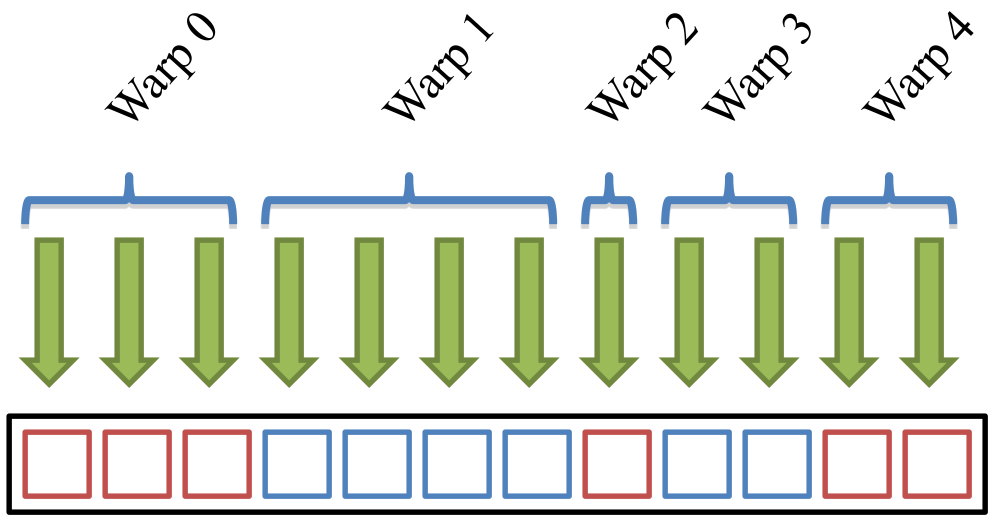
CSR (scalar) kernel

- One thread per row
  - Poor memory coalescing
  - Unaligned memory access



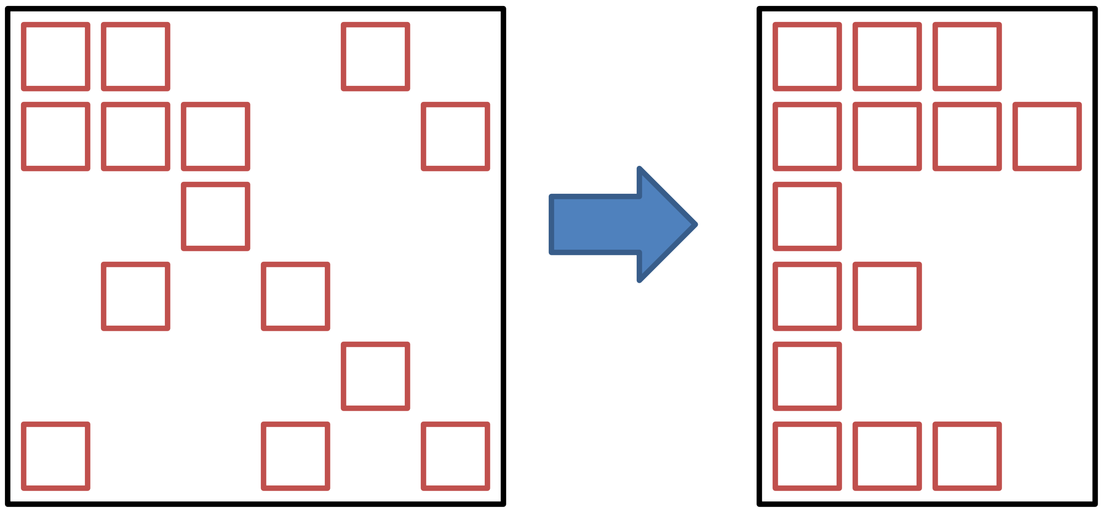
CSR (vector) kernel

- One SIMD vector or *warp* per row
  - Partial memory coalescing
  - Unaligned memory access



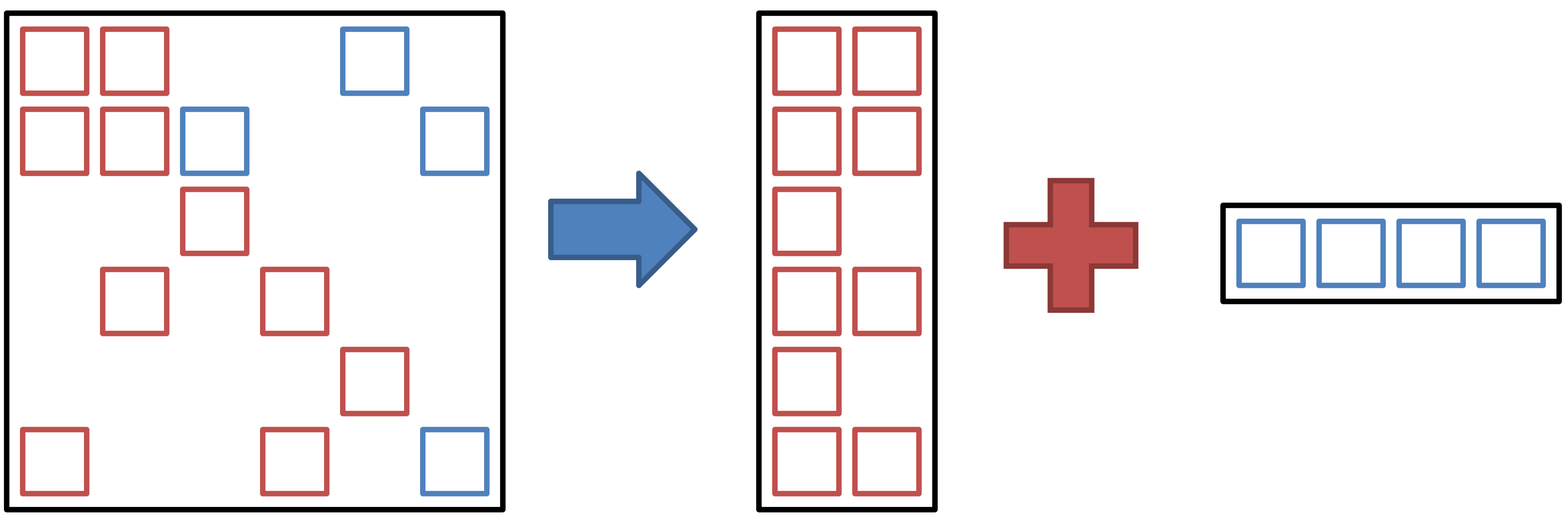
ELLPACK (ELL)

- Storage for K nonzeros per row
  - Pad rows with fewer than K nonzeros
  - Inefficient when row length varies



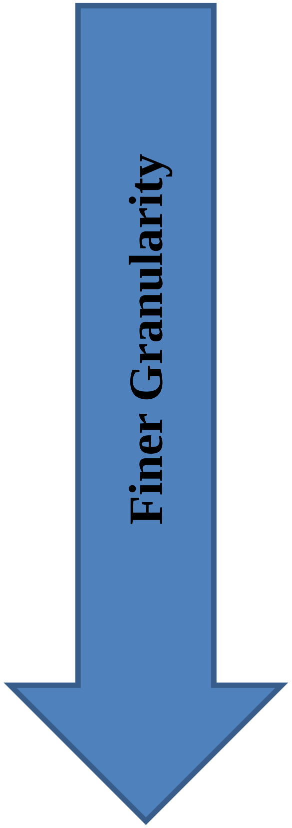
Hybrid Format

- ELL handles *typical* entries
- COO handles *exceptional* entries
  - Implemented with segmented reduction

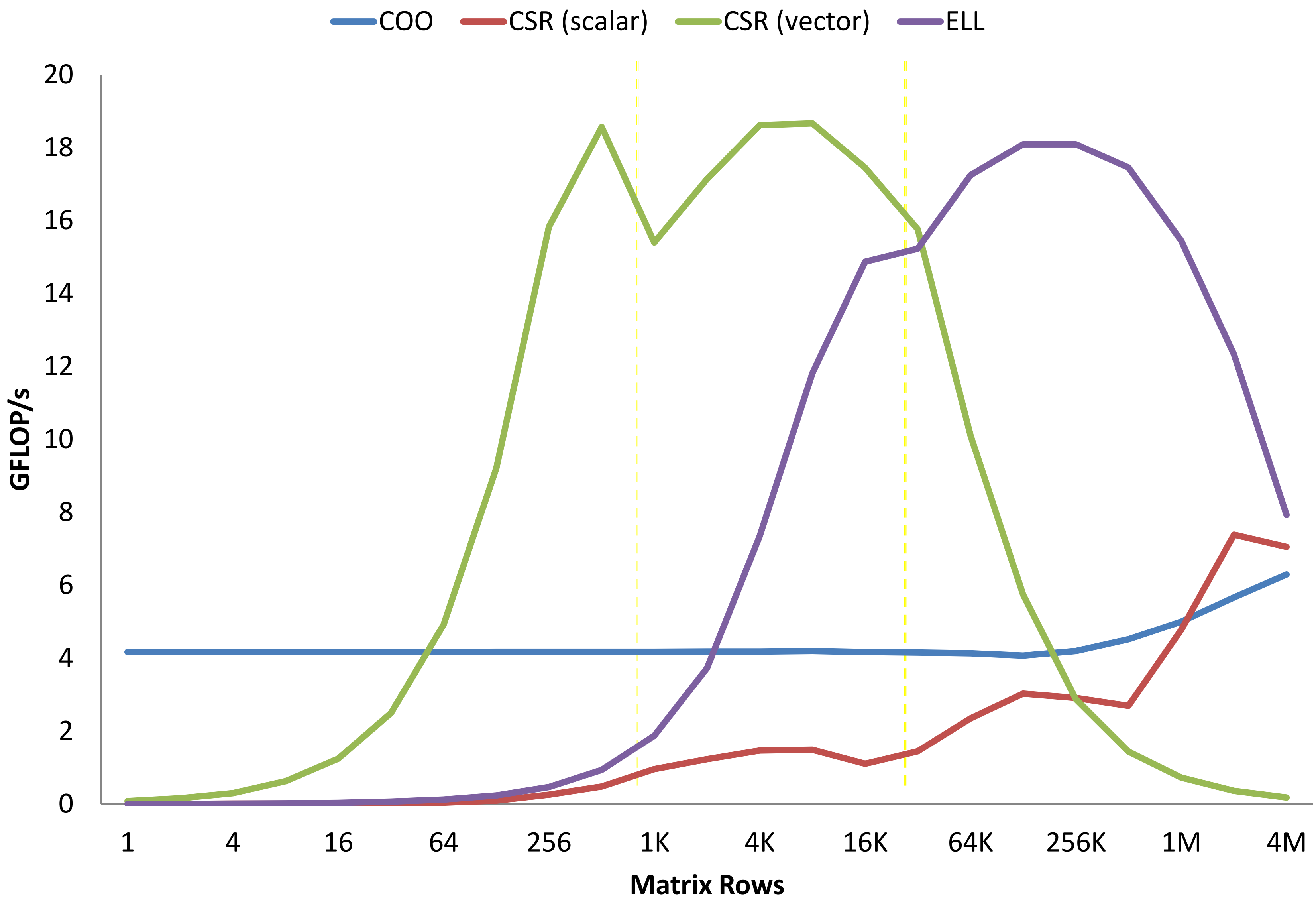


Exposing Parallelism

- DIA, ELL & CSR (scalar)
  - One thread per row
- CSR (vector)
  - One warp per row
- COO
  - One thread per nonzero



Exposing Parallelism



Execution Divergence

- Variable row lengths can be problematic
  - Idle threads in CSR (scalar)
  - Idle processors in CSR (vector)

- Robust strategies exist
  - COO is insensitive to row length

Memory Access Divergence

- Uncoalesced memory access is costly
  - Sometimes mitigated by cache
- Misaligned access is suboptimal
  - Align matrix format to coalescing boundary
- Access to matrix representation
  - DIA, ELL and COO are fully coalesced
  - CSR (vector) is partially coalesced
  - CSR (scalar) is seldom coalesced

Performance Comparison

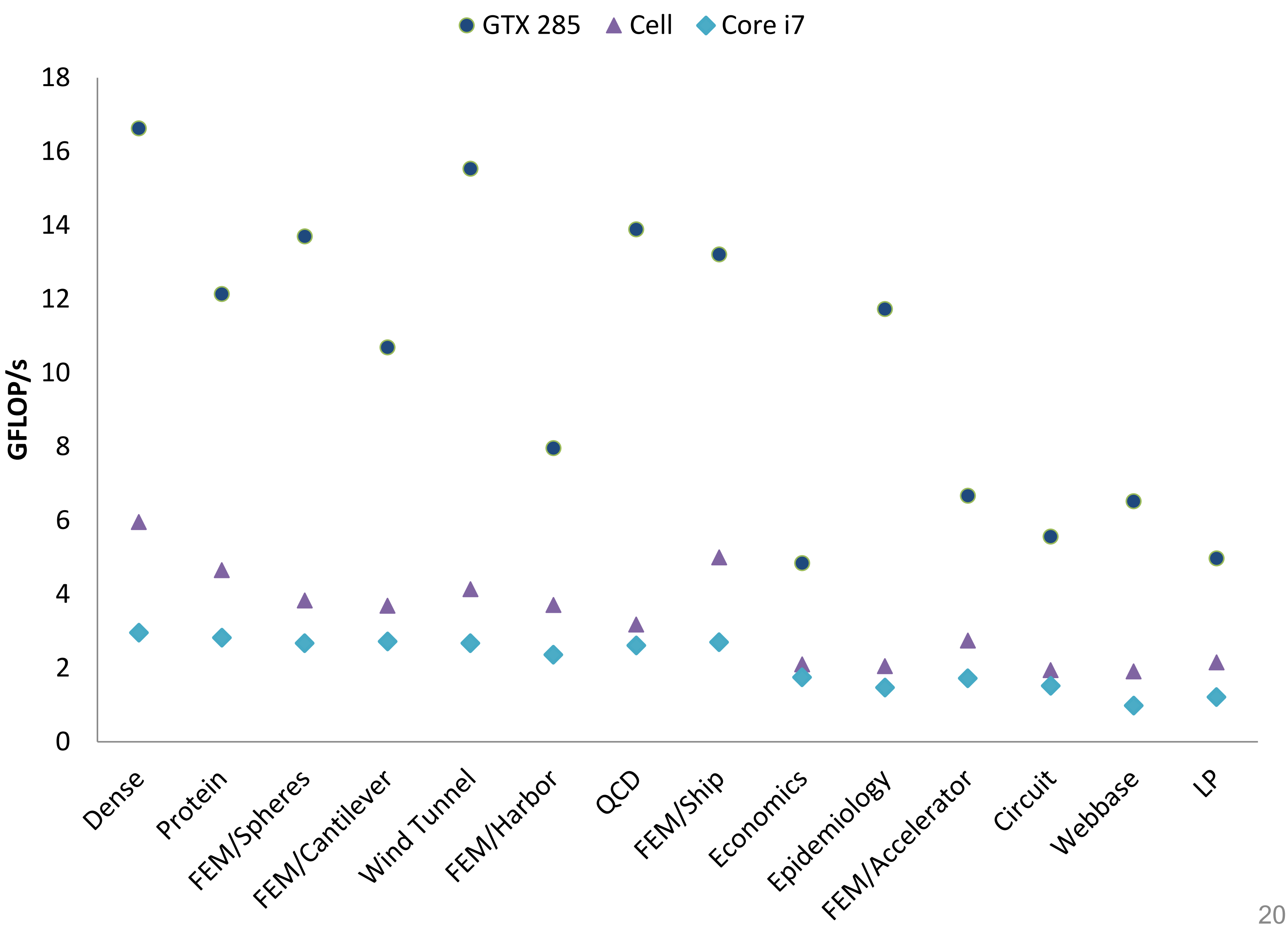
System	Cores	Clock (GHz)	Notes
GTX 285	240	1.5	NVIDIA GeForce GTX 285
Cell	8 (SPEs)	3.2	IBM QS20 Blade (half)
Core i7	4	3.0	Intel Core i7 (Nehalem)

Sources:

Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors
N. Bell and M. Garland, Proc. Supercomputing '09, November 2009

Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms
Samuel Williams et al., Supercomputing 2007.

Performance Comparison



ELL kernel

```
__global__ void ell_spmv(const int num_rows,          const int num_cols,
                         const int num_cols_per_row, const int stride,
                         const double * Aj,          const double * Ax,
                         const double * x,                 double * y)
   {
       const int thread_id = blockDim.x * blockIdx.x + threadIdx.x;
       const int grid_size = gridDim.x * blockDim.x;

       for (int row = thread_id; row < num_rows; row += grid_size) {
           double sum = y[row];

           int offset = row;

           for (int n = 0; n < num_cols_per_row; n++) {
               const int col = Aj[offset];

               if (col != -1)
                   sum += Ax[offset] * x[col];

               offset += stride;
           }

           y[row] = sum;
       }
   }
```

```
#include <cusp/hyb_matrix.h>
#include <cusp/io/matrix_market.h>
#include <cusp/krylov/cg.h>

int main(void)
{
    // create an empty sparse matrix structure (HYB format)
    cusp::hyb_matrix<int, double, cusp::device_memory> A;

    // load a matrix stored in MatrixMarket format
    cusp::io::read_matrix_market_file(A, "5pt_10x10.mtx");

    // allocate storage for solution (x) and right hand side (b)
    cusp::array1d<double, cusp::device_memory> x(A.num_rows, 0);
    cusp::array1d<double, cusp::device_memory> b(A.num_rows, 1);

    // solve linear system with the Conjugate Gradient method
    cusp::krylov::cg(A, x, b);

    return 0;
}
```

cusplibrary.github.com

A library for **sparse linear algebra** and **graph** computations on CUDA

Summed Area Tables

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

Gabriel Zachmann
University of Bremen
Massively Parallel Algorithms - 2018

Summed Area Table

- Summed Area Table (SAT): 2D table where each element stores the sum of all elements in an input image between the lower left corner and the entry location.

Summed Area Table

- Example:

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

$$(1 + 1 + 0) + (1 + 2 + 1) + (0 + 1 + 2) = 9$$

Summed Area Table

- Benefit
  - Used to compute different width filters at every pixel in the image in constant time per pixel
  - Just sample four pixels in SAT:

$$s_{filter} = \frac{s_{ur} - s_{ul} - s_{lr} + s_{ll}}{w \times h},$$

Summed Area Table

- Uses
  - Glossy environment reflections and refractions
  - Approximate depth of field

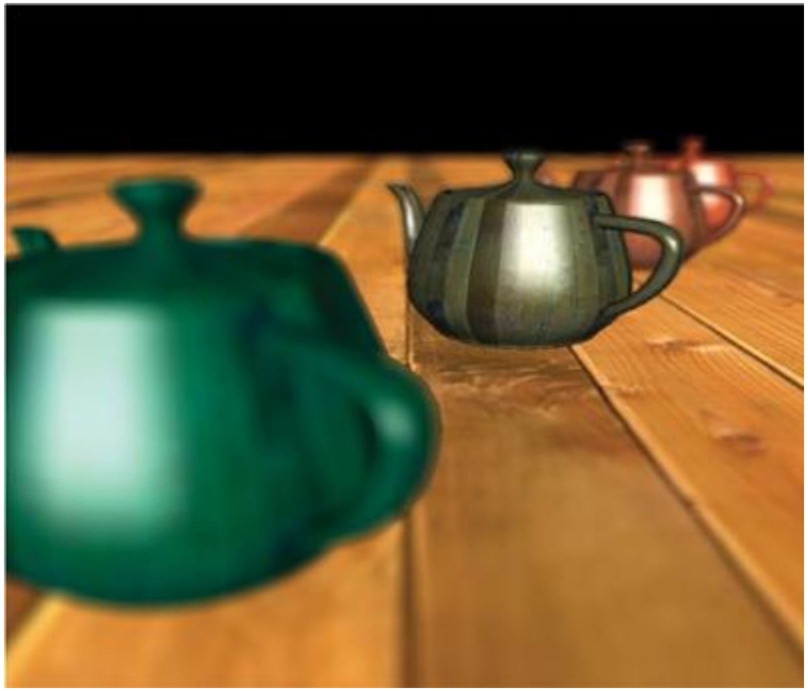


Image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

Slides by P. Cozzi, UPenn

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1			

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2		

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2	2	

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

2			
1	2	2	4

Slides by P. Cozzi, UPenn

Summed Area Table

Input image

2	1	0	0
0	1	2	0
$\boxed{1}$	$\boxed{2}$	1	0
$\boxed{1}$	$\boxed{1}$	0	2

SAT

2	$\boxed{5}$		
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9		
2	6	9	11
2	5	6	8
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	
2	6	9	11
2	5	6	8
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

Summed Area Table

How would you implement this on the GPU?

Summed Area Table

- Recall Inclusive Scan:

0	1	2	3	4	5	6	7
0	1	3	6	10	15	21	28

Summed Area Table

- Step 1 of 2:

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

Partial SAT

2	3	3	3
0	1	3	3
1	3	4	4
1	2	2	4

$\longrightarrow$

One inclusive scan for each row

Summed Area Table

- Step 2 of 2:

Partial SAT

2	3	3	3
0	1	3	3
1	3	4	4
1	2	2	4

Final SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

One inclusive scan for each Column, bottom to top

Issues

- Caveat: precision of integer/floating-point arithmetic
  - Assumption: each T_{ij} needs b bits
  - Consequence: number of bits needed for $S_{wh} = logw + logw + b$
  - Example: 1024x1024 grey scale input image, each pixel = 8 bits
    - 28 bits needed in S-pixels

Increasing Precision (1)

- Signed offset representation:
- Set $$T'(i,j) = T(i,j) - \bar{t}$$

$$\text{where } \bar{t} = \text{ average of } T = \frac{1}{wh}\sum_1^w \sum_1^h T(i,j)$$

- Effectively "removes the DC component from the signal"

Increasing Precision (1)

- Consequence:

$$S'(i,j) = \sum_{k=1}^{i} \sum_{l=1}^{j} T'(k,l) = S(i,j) - i \cdot j \cdot \bar{t}$$

i.e., the values of S' are now in the same order as the values of T (fewer bits have to be thrown away during the summation)

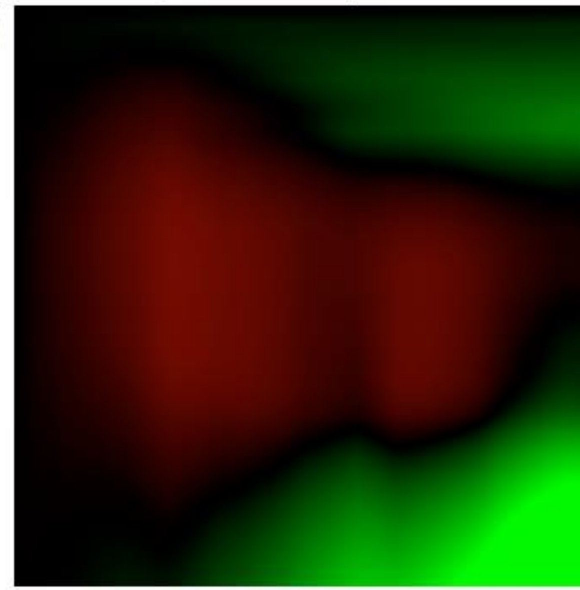
- Note 1: we need to set aside 1 bit (sign bit)
- Note 2: $S'(w,h) = 0$ (modulo rounding errors)

Example

Input image

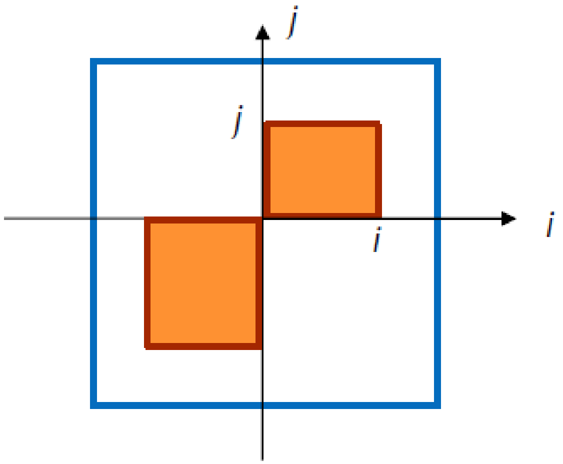
Original summed area table

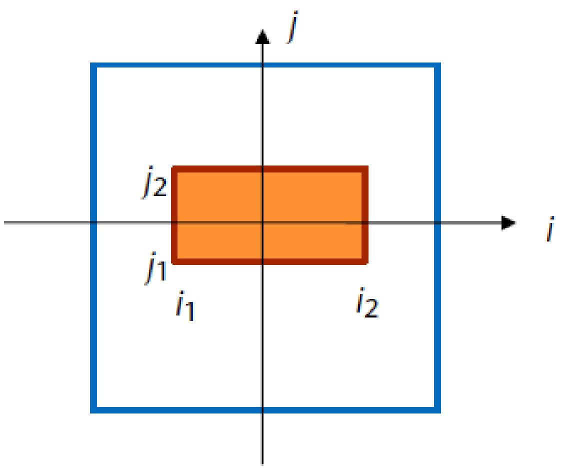
With improved precision
using "offset" representation



Increasing Precision (2)

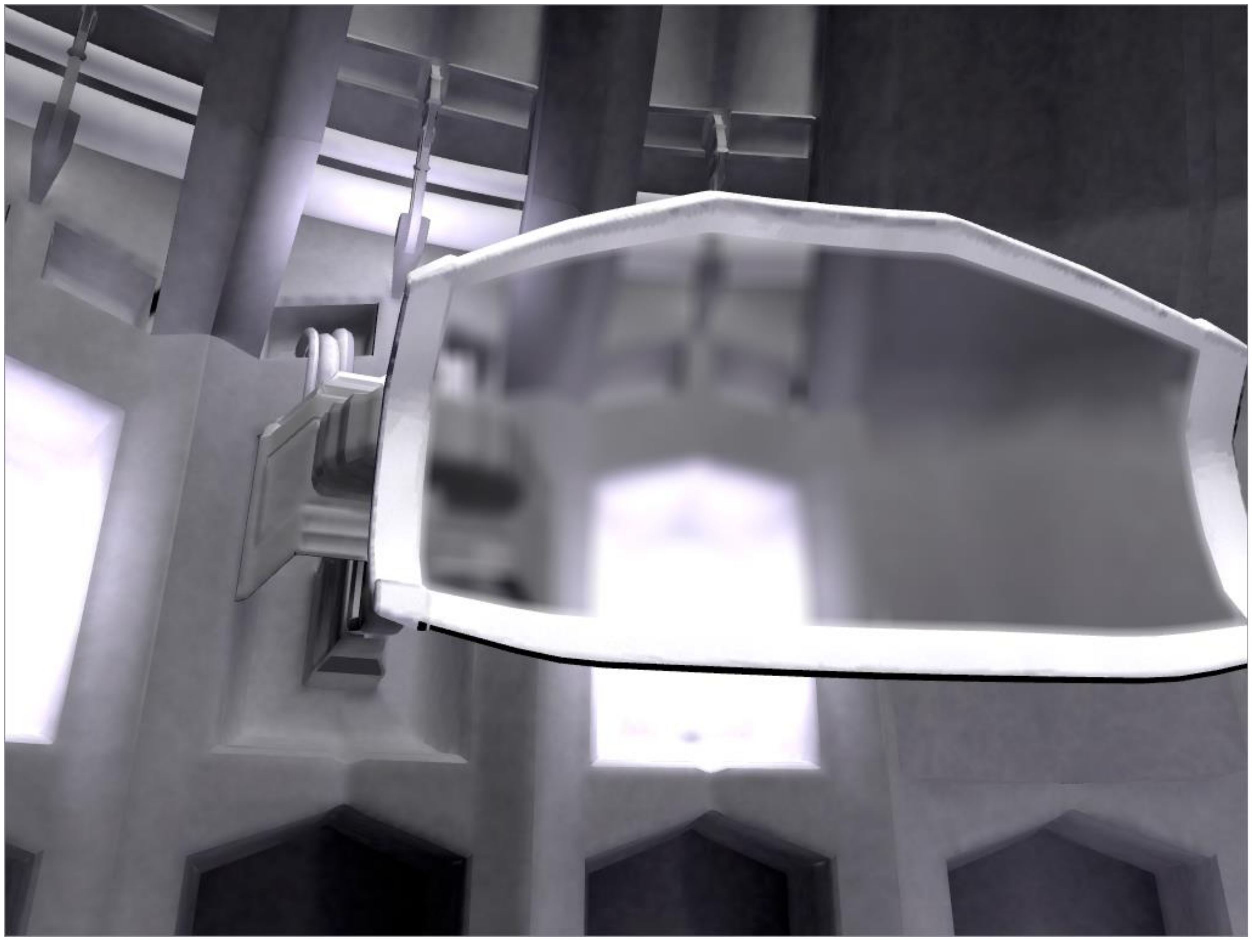
- Move the "origin" of the *i,j* "coordinate
- frame“
- Compute 4 different *S*-tables, one for each quadrant
- Result: each *S*-table comprises only ¼ of the pixels of *T*
- For computation of *T(k,l)* do a simple case switch





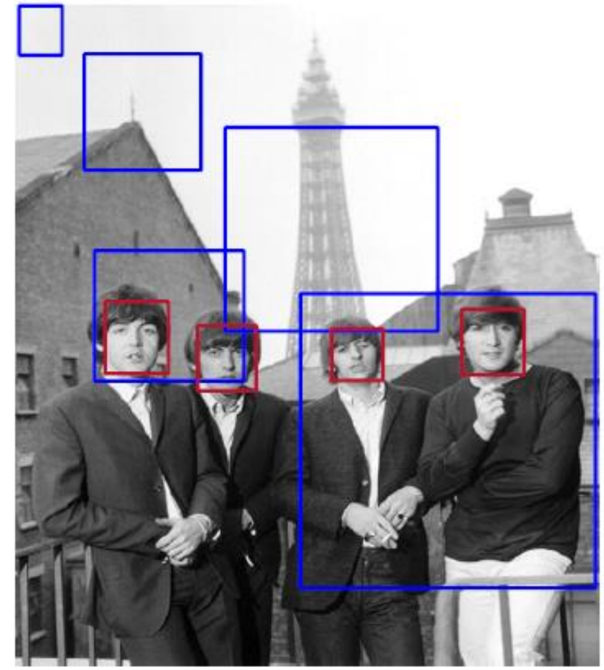
With methods 1 & 2 | Simple method

Application: Depth of Field



Application: Viola-Jones Face Detector

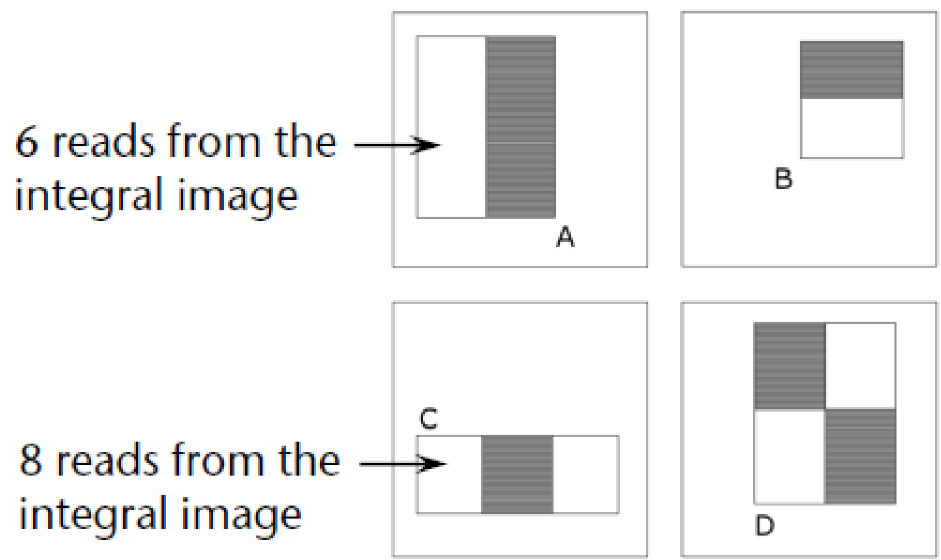
- The (simple) idea:
  - Move sliding window across image (all possible locations, all possible sizes)
  - Check, whether a face is in the window
  - We are interested only in windows that are filled by a face



- Observation:
  - Image contains 10s of faces
  - But ≈ 10^6 candidate windows
- Consequence:
  - To avoid having a false positive in every image, our false positive rate has to be < 10^{-6}

Application: Viola-Jones Face Detector

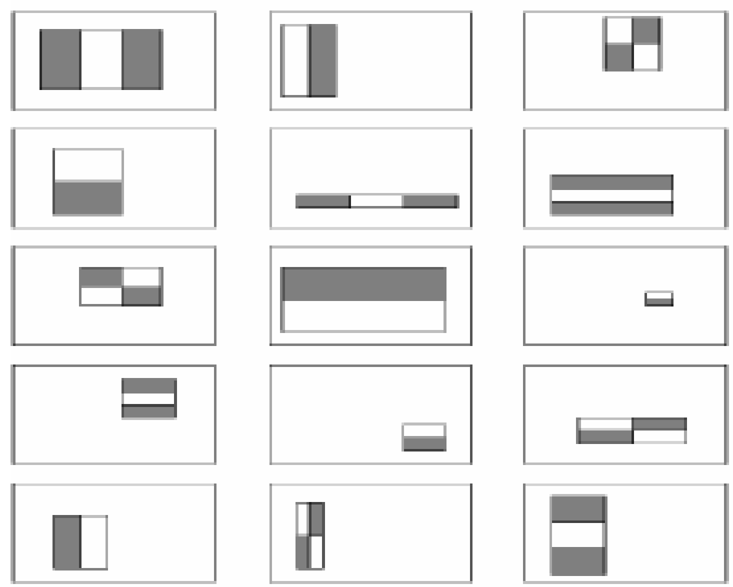
- Feature types used in the Viola-Jones face detector:
  - 2, 3, or 4 rectangles placed next to each other
  - Called Haar features
- Feature value: g_i = pixel-sum(white rectangle(s)) - pixel-sum(black rectangle(s))



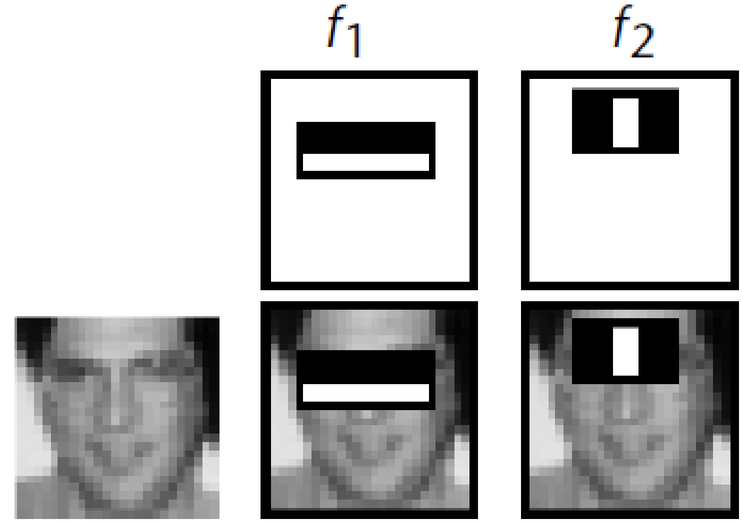
Slides by G. Zachmann, University of Bremen

Application: Viola-Jones Face Detector

- Constant time per feature extraction
  - In a 24x24 window (e.g., one of the sliding windows), there are ≈ 160,000 possible features
  - All variations of type, size, location within the window



Application: Viola-Jones Face Detector



- Define a weak classifier for each feature

$$f_i = \begin{cases} +1 & , g_i > \theta_i \\ -1 & , \text{else} \end{cases}$$

  – "Weak" because such a classifier is only slightly better than a random "classifier"

- Goal: combine lots of weak classifiers to form one strong classifier

$$F(\text{window}) = \alpha_1 f_1 + \alpha_2 f_2 + \ldots$$

Parallel Sorting

Scott B. Baden
UCSD, CSE 160
Winter 2013

Parallel Sorting

- We’ll consider in-memory sorting of integer keys
  - Bucket sort
  - Sample sort
  - Bitonic sort (later)

Rank Sorting

- Compute the rank of each input value
- Move each value in sorted position according to its rank
- Makes idealizing assumptions
  - An ideal parallel computer with no memory contention and an infinite number of processors
  - The forall loops parallelize perfectly

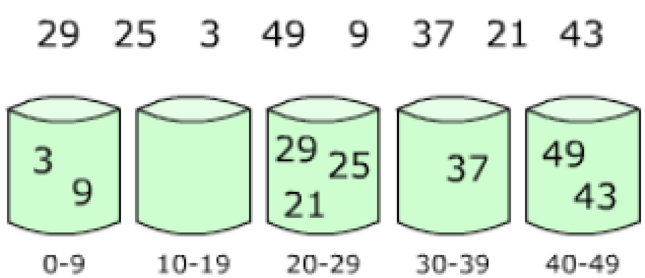
```
forall i=0:n-1, j=0:n-1
        if ( x[i] > x[j] ) then rank[i] += 1 end if
forall i=0:n-1
        y[rank[i]] = x[i]
```

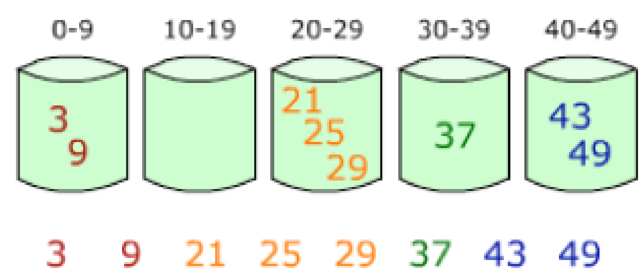
In Search of a Fast and Practical Sort

- Rank sorting is impractical on real hardware
- Let’s borrow the concept: compute the thread owner for each key
- Shuffle data in sorted order in one step
- But how do we know which thread should be the owner?
- Subdivide the key space

First Attempt: Bucket Sort

- Divide the range of keys into equal subranges and associate a *bucket* with each range
- Each processor maintains p local buckets
  - Assigns each key to a bucket: $\lfloor p \times key/(K_{max}-1) \rfloor$
  - Routes the buckets to the correct owner (each local bucket has ≈ **n/p²** elements)
  - Sort all incoming data in each bucket





Wikipedia

Runtime

- Assume that the keys are distributed uniformly over 0 to Kmax-1
- Local bucket assignment: O(n/p)
- Route each local bucket to the correct owner O(n)
- Local sorting (using radix sort) : O(n/p)) http://users.monash.edu/~lloyd/tildeAlgDS/Sort/Radix/

Worst Case Behavior

- The assignment of keys to threads is based solely on the knowledge of Kmax
- If the keys are integers in the range [0,Q-1] ….thread k has keys in the range

$$\left[k\frac{Q}{P},(k+1)\frac{Q}{P}\right]$$

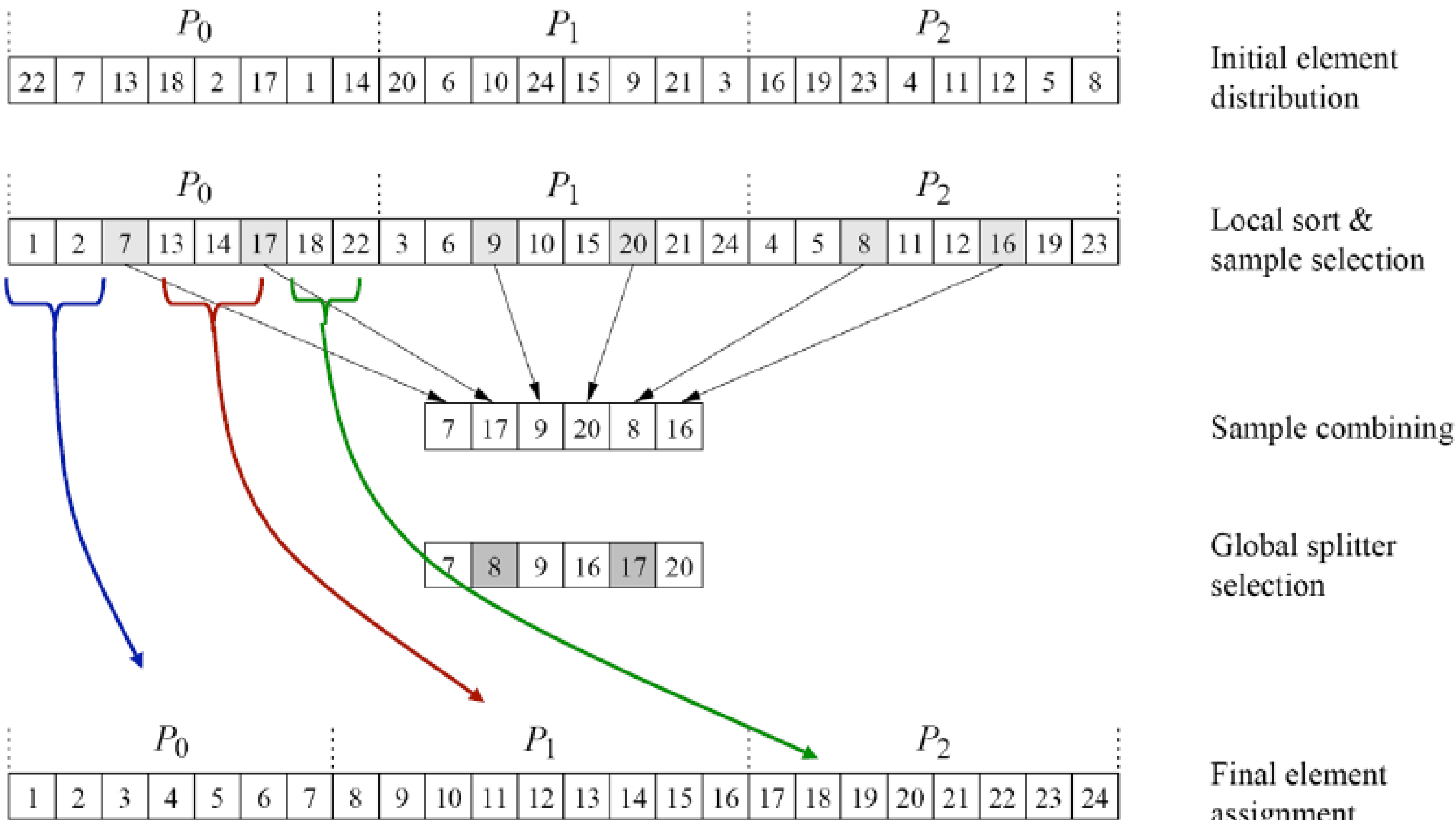
- E.g. for $Q=2^{30}$, P=64, each thread gets 2^{24} = 16 M elements
- For a non-uniform distribution, we need more information to balance keys (and communication) over the processors
- In the worst case, all the keys could go to one processor

Improving on Bucket Sort

Sample sort

- Uses a heuristic to estimate the distribution of the global key range over the p threads
- Each processor gets about the same number of keys
- Sample the keys to determine a set of p-1 *splitters* that partition the key space into p disjoint regions (buckets)

Sample Selection



Introduction to Parallel Computing, 2nd Ed.,, A.Grama, A.I Gupta, G. Karypis, and V. Kumar, Addison-Wesley, 2003.

Splitter Selection: Regular Sampling

- Shi and Schaeffer [1992]
- Each processor sorts its local keys, then selects *s* evenly spaced samples
- These candidate splitters are collected by one thread
  - Sorted
  - Sampled at uniform positions to generate a *p-1* element splitter list

Performance

- Assuming $n \geq p3$ …
- $T_P = O((n/p) \log n)$
- If $s = p$, each processor will merge no more than $2n/p + n/s - p$ elements
- If s > p, each processor will merge no more than
- $(3/2)(n/p) - (n/(ps)) + 1 + d$ elements
- Duplicates d do not impact performance unless $d = O(n/p)$
- Tradeoff: increasing s …
  - Spreads the final distribution more evenly over the processors
  - Increases the cost of determining the splitters
- For some inputs, communication patterns can be highly irregular with some pairs of processors communicating more heavily than others, lowering performance

Radix Sort

- We need a **stable** sorting algorithm to do the local sorts: the output preserves the order of inputs having the same associated key
- *radix sort* meets our needs: sort the keys in passes, choosing an r-bit block at a time, O(n) running time
- Explanation with a demo www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix/

Radix Sort Example

- Consider the input keys

  34, 12, 42, 32, 44, 41, 34, 11, 32, and 23
- Use 4 buckets
- Sort on each digit in succession, least significant to most significant

Radix Sort Example

- Consider the input keys

  34, 12, 42, 32, 44, 41, 34, 11, 32, and 23
- Use 4 buckets
- Sort on each digit in succession, least significant to most significant
- After pass 1

  41 11   12 42 32 32   23   34 44 34

41 11   12 42 32 32   23   34 44 34

Radix Sort Example

- Consider the input keys

  34, 12, 42, 32, 44, 41, 34, 11, 32, and 23
- Use 4 buckets
- Sort on each digit in succession, least significant to most significant
- After pass 1

  41 11   12 42 32 32   23   34 44 34
- After pass 2

  11 12   23   32 32 34 34   41 42 44