Irregular Algorithms on the GPU

P. J. Narayanan
Centre for Visual Information Technology
International Institute of Information Technology
Hyderabad

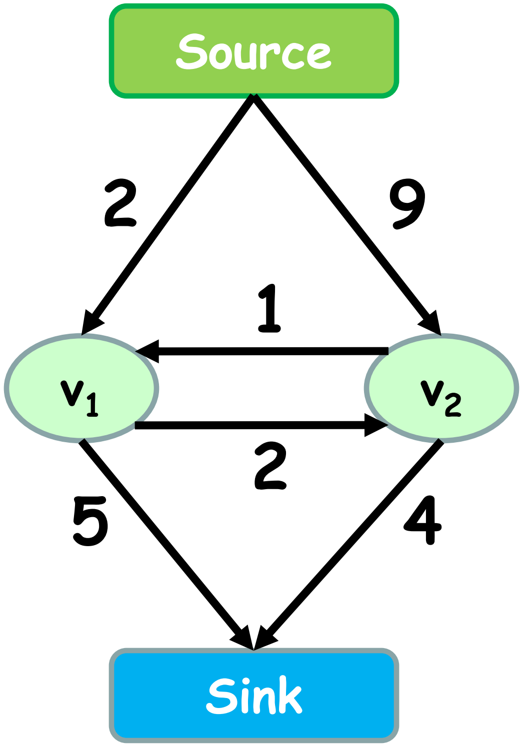
Graph Cuts for Computer Vision on the GPU

Work done with Vibhav Vineet
(CVGPU08 Workshop)

Graph Cuts in Computer Vision

- Several optimization problems have been mapped to *maxflow* on a graph built from the pixels with a special *s* node and *t* node.
  - Segmentation: Assign binary labels to pixels
    - Pixels connected to *s* after cut is foreground and the rest are background.
  - Stereo matching: Assign integer labels to pixels
    - Disparity is the standard label.
    - Framework works for many problems
- Many sequential algorithms exist. Goldberg-Tarjan (push-relabel) and Edmonds-Karp (augmenting path based) are popular.
  - Former is more parallelizable

The st-Mincut Problem



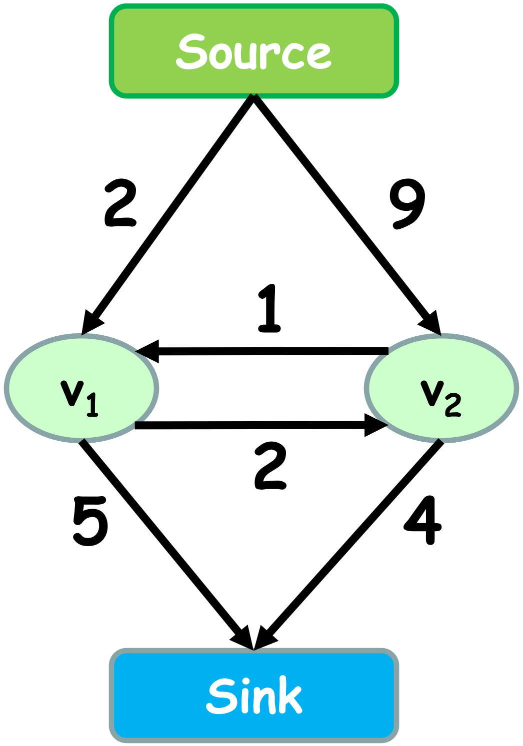
Graph (V, E, C)

Vertices V = $\{v_1, v_2 \ldots v_n\}$

Edges E = $\{(v_1, v_2) \ldots\}$

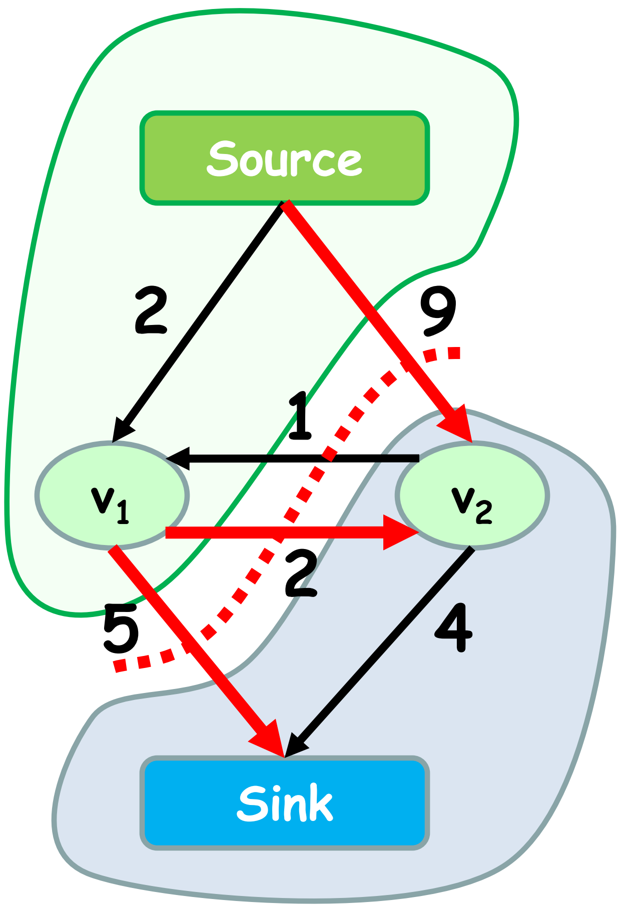
Costs C = $\{c_{(1, 2)} \ldots\}$

The st-Mincut Problem



What is an st-cut?

The st-Mincut Problem



5 + 2 + 9 = 16

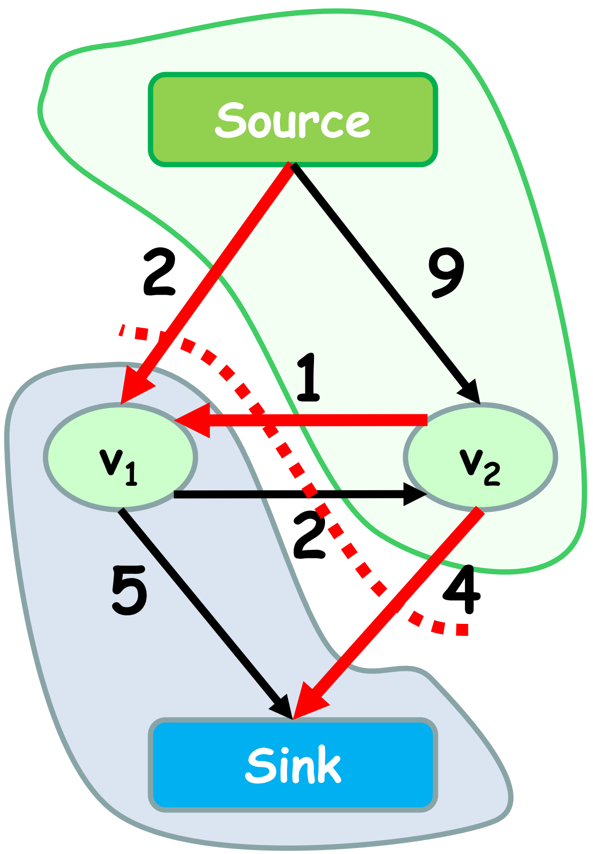
What is an st-cut?

An st-cut (**S**,**T**) divides the nodes between source and sink.

What is the cost of a st-cut?

Sum of cost of all edges going from S to T

The st-Mincut Problem



$2 + 1 + 4 = 7$

What is an st-cut?

An st-cut (**S**,**T**) divides the nodes between source and sink.

What is the cost of a st-cut?

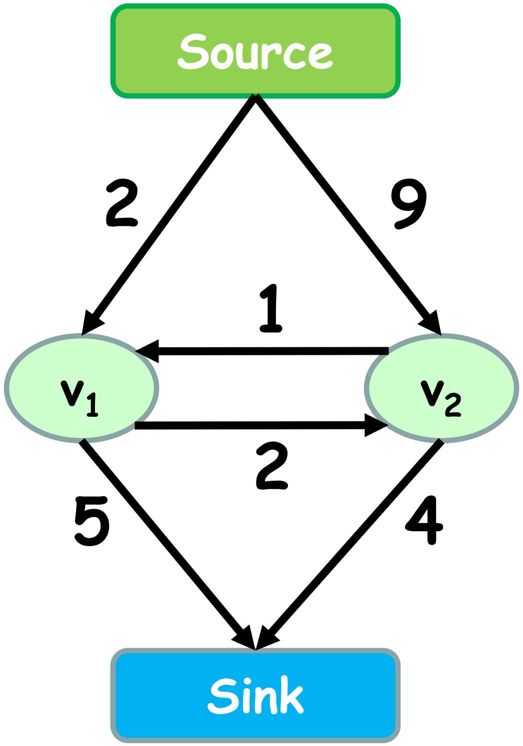
Sum of cost of all edges going from S to T

What is the st-mincut?

st-cut with the minimum cost

Maxflow Algorithms

Flow = 0

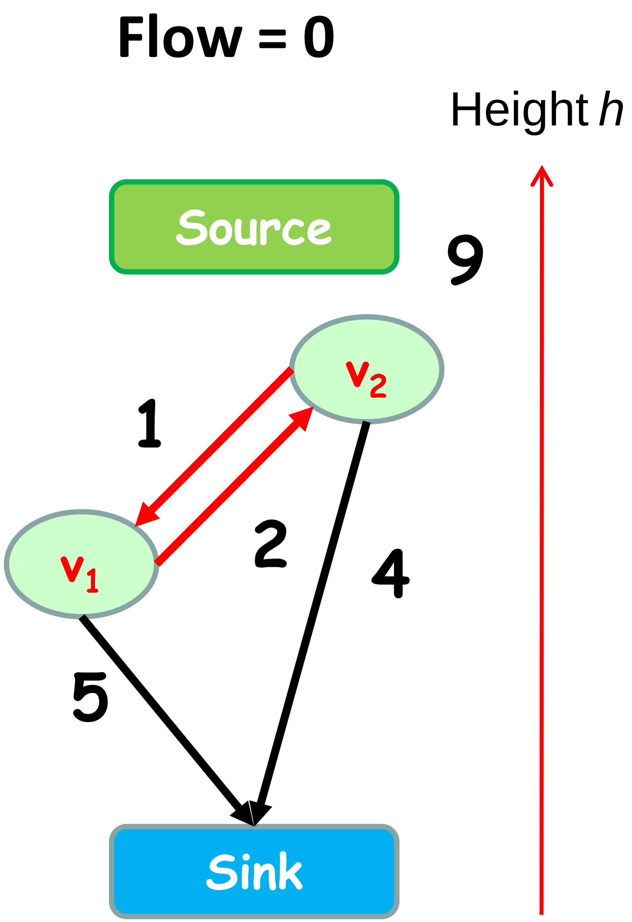


Goldberg's generic Push-Relabel Algorithm

1. Intialize-Preflow(G,s)
2. Perform an applicable push or relabel operation
3. Repeat untill there exists no applicable push or relabel operation

Algorithms assume non-negative capacity

Maxflow Algorithms



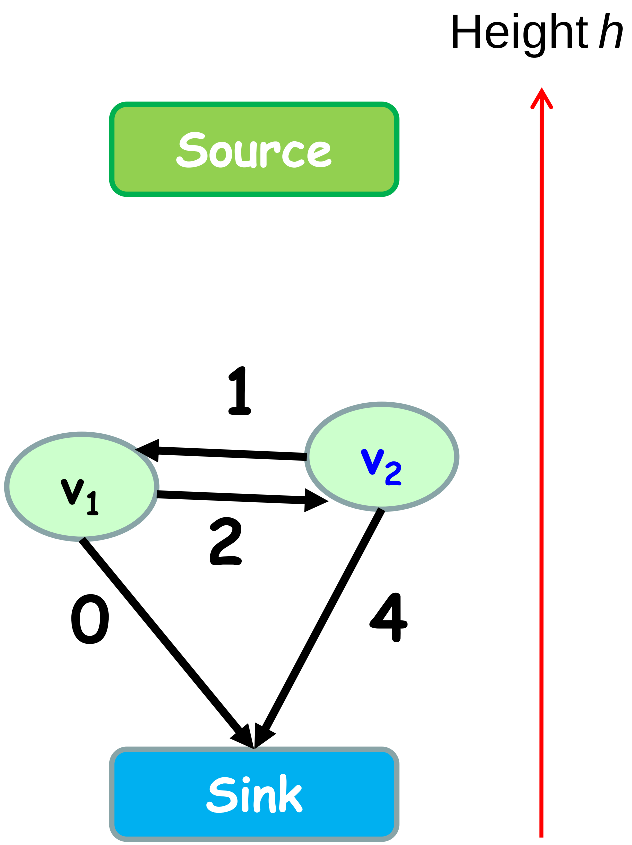
Push Operation

1. V_2 is overflowing
2. Height $h(V_2) == h(V_1) + 1$
3. Push as much unit of flows from V_2 to V_1

Algorithms assume non-negative capacity

Maxflow Algorithms

Flow = 0

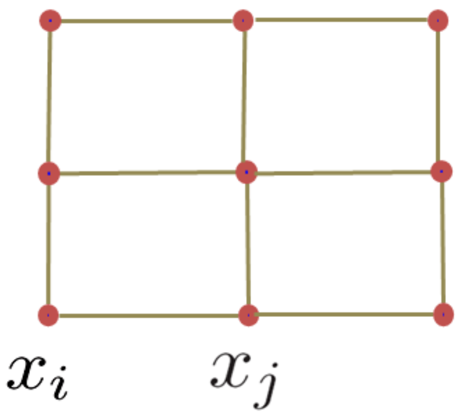


Relabel Operation

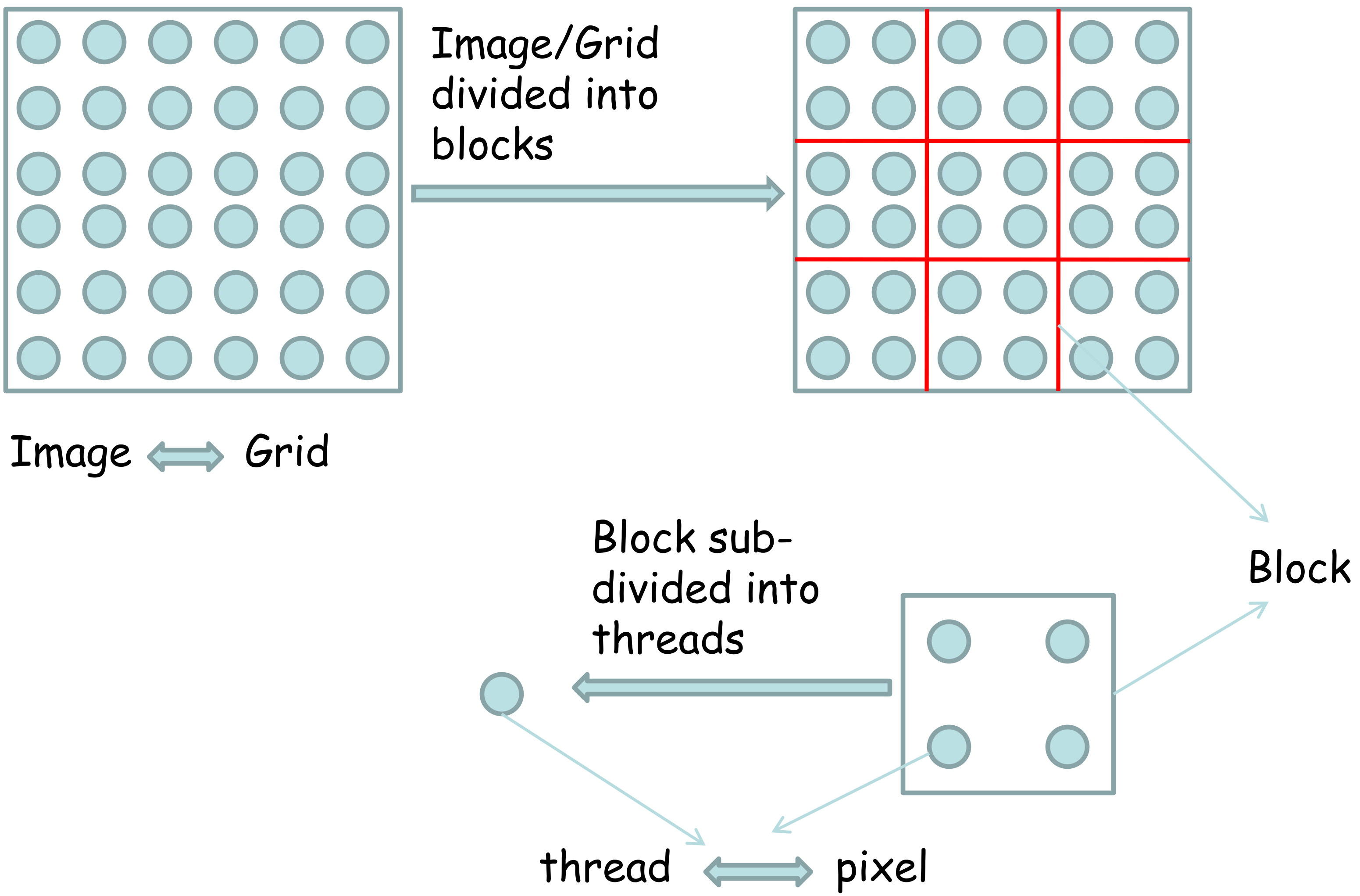
1. V_2 is overflowing and is in residual graph
2. Height $h(V_2) <= h(V_1)$
3. Increase the height of V_2

GraphCuts on Images

- Specialized algorithms for vision problems
  - Grid graphs
  - Low connectivity; typically limited to 4, 8 or 27



Mapping Image On CUDA

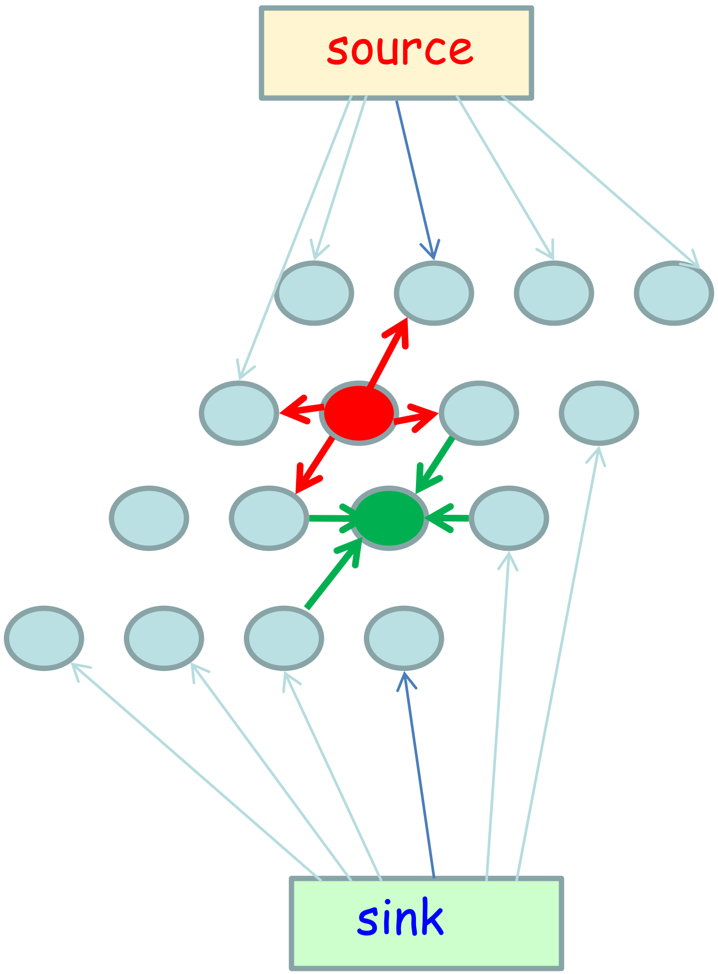


Push Relabel Algorithm on CUDA

1. Push is an local operation with each node sending flows to its neighbors
2. Relabel is also a local operation
3. Problems faced:
   1. RAW problems: (Read after write)
   2. Synchronization is limited to the threads of a block.

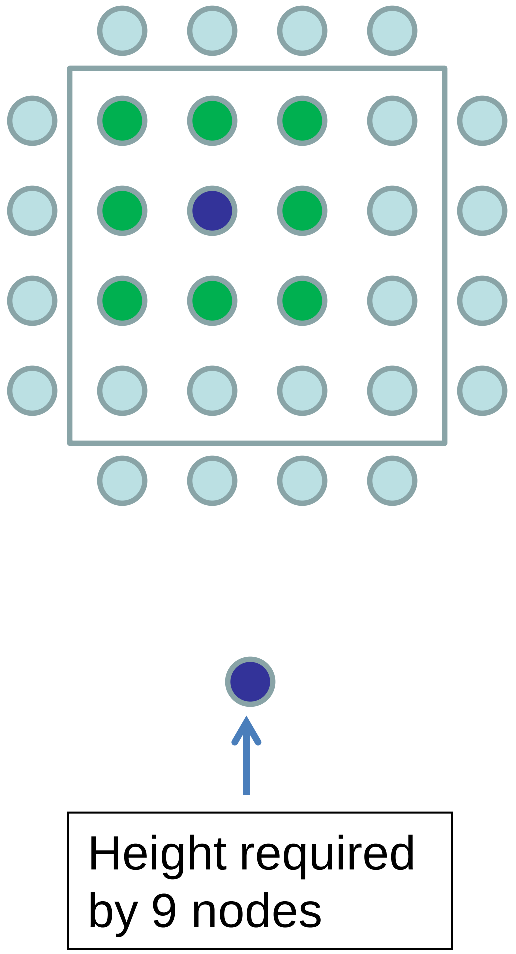
Push Relabel Algorithm on CUDA

1. Push operation is divided into two phases: Push Phase and Pull Phase
2. Relabel is also local operation
3. Naïve Solution: Three Kernels
   1. Push Kernel
   2. Pull Kernel
   3. Relabel Kernel



Push Kernel (node u)

1. Load h(u) from the global memory to shared memory of the block.
2. Synchronize threads to ensure completion of load
3. Push flow to the eligible nodes without violating the preflow conditions.
4. Update the residual capacities of edges(u,v) in the residual graphs.
5. Store the flow pushed to each edge in a special global memory array F.



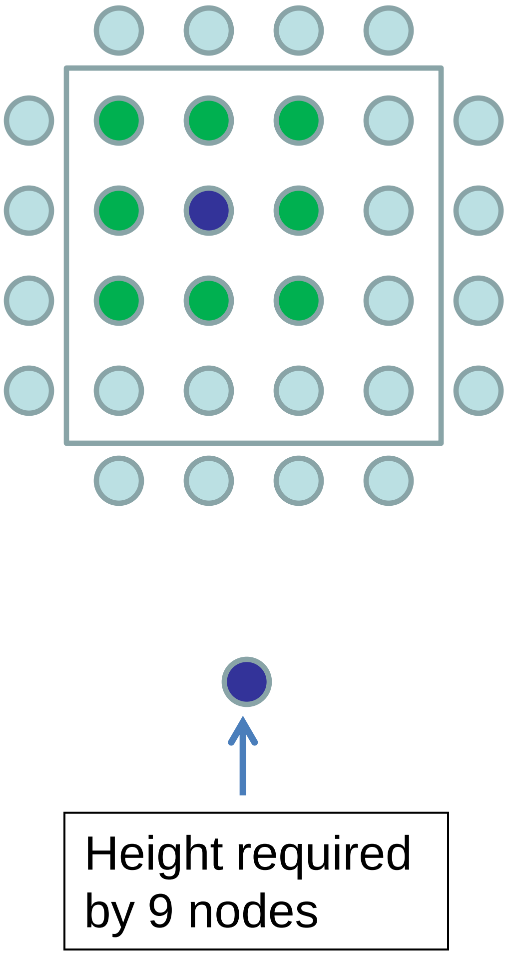
Pull Kernel (node u)

1. Read the flows pushed to u from the F array of its neighbors.

2. Compute the final excess flow by aggregating all incoming flows. Store it as the e(u) value in global memory.

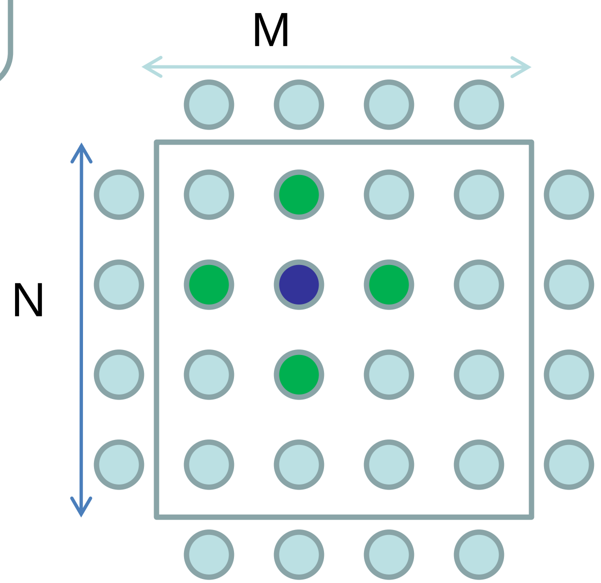
Relabel Kernel (node u)

1. Load h(u) from the global memory to the shared memory.
2. Synchronize to ensure the completion of load of heights.
3. Compute the minimum heights of neighbors of node u.
4. Write the new height to global memory location h(u).



Push and Relabel Kernels (Shared Memory)

1. Load h(u) from the global memory to shared memory of the block.



Shared Memory Used:

- Each Block has MxN threads.

Internal Nodes:

Each Internal Node ( ) requires heights of 4 other nodes ( ) from the same block.

IIIT Hyderabad

Push and Relabel Kernels (Shared Memory)

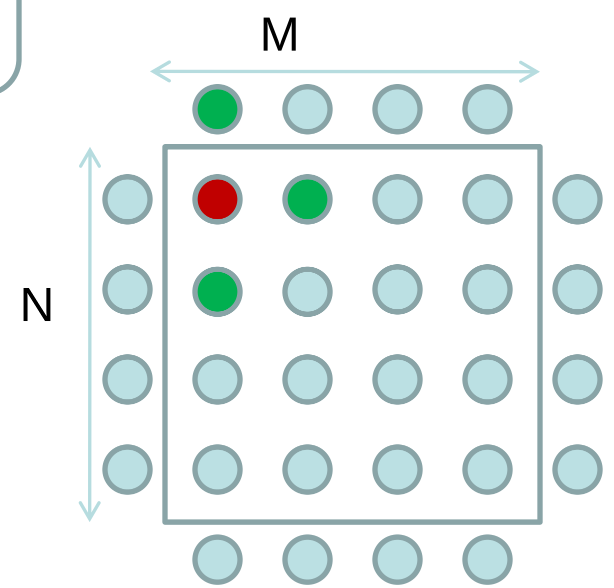
1. Load h(u) from the global memory to shared memory of the block.

Shared Memory Used:

- Each Block has MxN threads.

Border Nodes:

Each Border Node( ) requires heights of other nodes( ) from the different blocks.

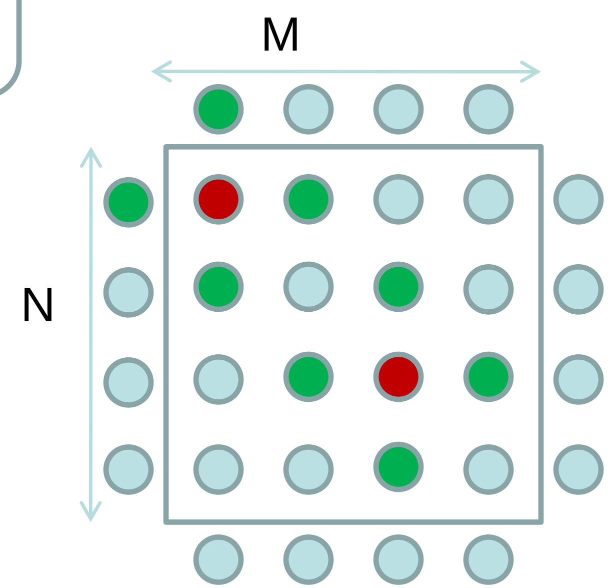


Push and Relabel Kernels (Shared Memory)

1. Load h(u) from the global memory to shared memory of the block.

Shared Memory Used:

- Each Block has MxN threads.
- Total Shared Memory Used:
    - (M+2)x(N+2)x(sizeof(element))



CUDA Block

Push Relabel Algorithm

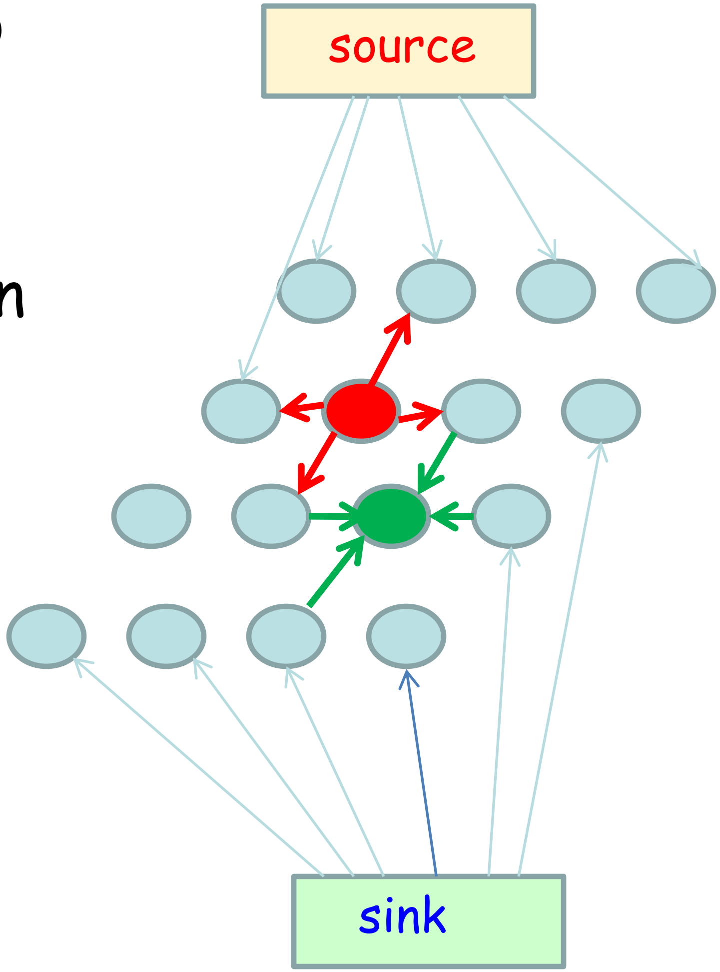
1. Push operation is divided into two phases: Push Phase and Pull Phase

2. Relabel is also local operation

3. Different Solution : Two kernels

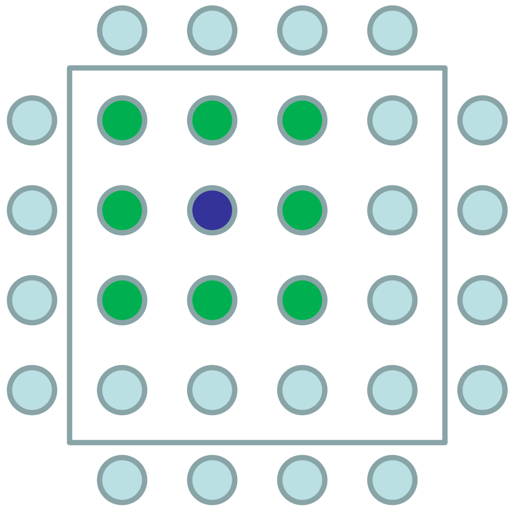
   1. Push Kernel

   2. Pull + Relabel Kernel



Pull + Relabel Kernel (node u)

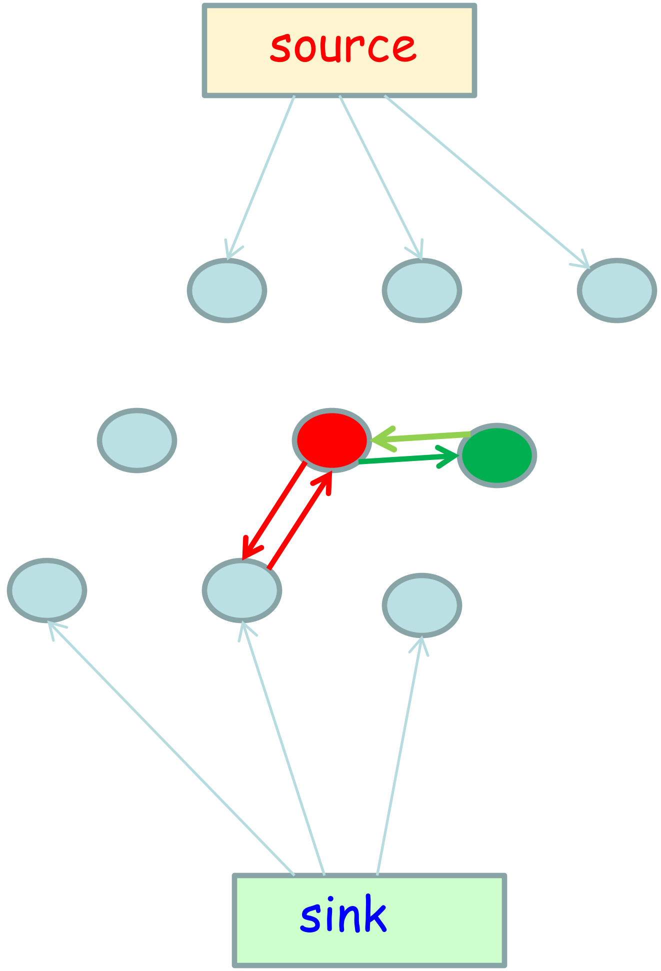
1. Load h(u) from the global memory to the shared memory.
2. Synchronize threads to ensure the completion of load.
3. Update the excess flow e(u) and residual capacities of edges (u,v) in the residual graph with the flows from the global memory array F.
4. Synchronize to ensure completion of updation of edge-weights and excess flow.
5. Compute the minimum heights of neighbors of node u.
6. Write the new height to global memory location h(u).



Height required by 9 nodes

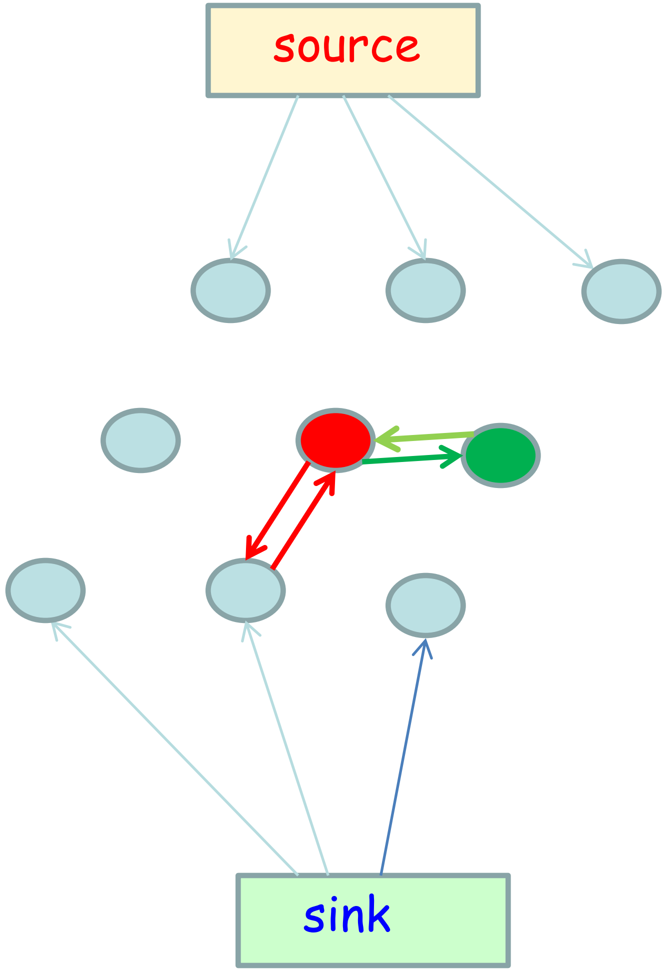
On Hardware with Atomic Capabilities

1. Push and Pull operations can performed without any RAW problem.
2. Relabel is also local operation
3. Third Solution on Hardware with Atomic Capabilities: Two kernels
    1. Push + Pull Kernel
    2. Relabel Kernel



Push + Pull Kernel (node u)

1. Load h(u) from the global memory to the shared memory.
2. Synchronize threads to ensure the completion of load.
3. Push flows to eligible neighbors **atomically** without violating the preflow condition.
4. Update the edge-weights of (u,v) and (v,u) **atomically** in the residual graph.
5. Update the excess flow of e(u) and e(v) atomically in the residual graph.



Results

Image	Size	Time (CPU) (millisecond)	Time (Non- Atomic)	Time (Atomic)	Time (Stochastic)
Sponge	640x480	142	28	16	11
Flower	608x456	188	33	26	16
Person	608x456	140	31	27	20
Synthetic	1Kx1K	655	19	10	7

Vibhav Vineet and P J Narayanan. “CudaCuts”. IEEE CVPR Workshop on Computer Vision on the GPUs. Alaska, June 2008.

Fast and Scalable List Ranking on the GPU

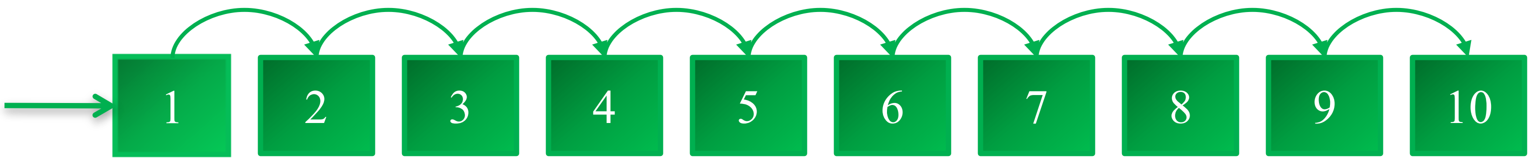
M. Suhail Rehman, Kishore Kothapalli, P. J. Narayanan

Center for Security, Theory, and Algorithmic Research
Center for Visual Information Technology
International Institute of Information Technology, Hyderabad

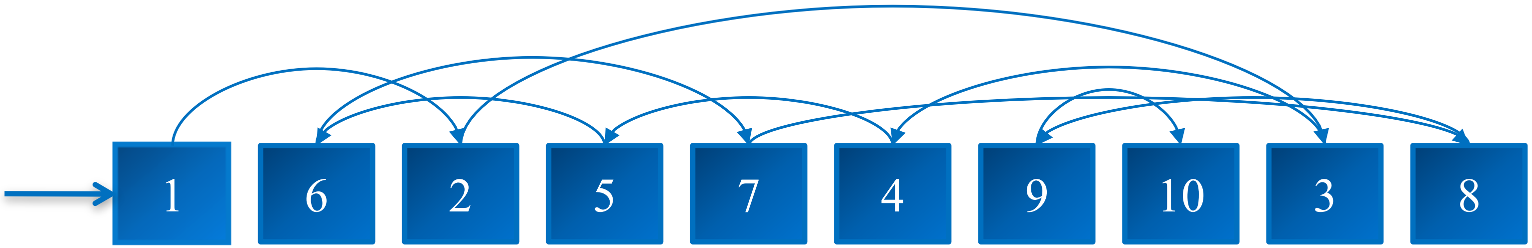
The List Ranking Problem

- Given a list of N elements, rank each element based on the distance of that element with the end of the list.
- A sequential algorithm is trivial and runs on O(n)
- Many parallel algorithms exist for various models.

Types of Linked Lists



Ordered List



Unordered List

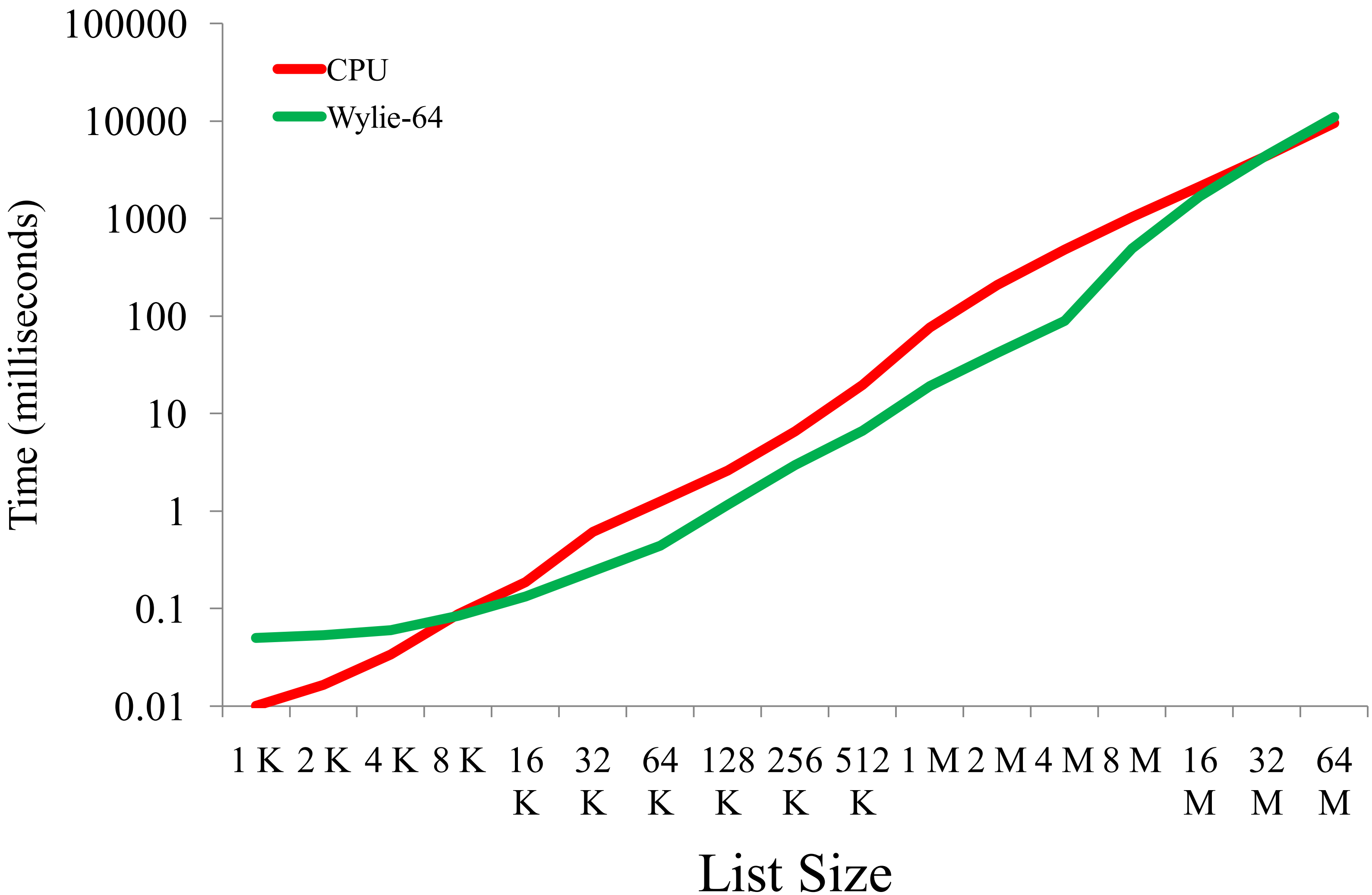
Baseline Implementation

- Wyllie's Algorithm uses Pointer Jumping
- Initialize Ranks to 1
- For each element in Array, set it's rank to rank + rank of Successor
- Reset the Successor value to the successor of it's successor (effectively jumping over and contracting the list)

GPU-Specific Optimizations

- Load the data elements when needed
- Bitwise operations to pack and unpack data
- Block-level thread synchronization to force threads to write in a coalesced manner
- Current best implementation of Pointer Jumping on the GPU

Results



Time (milliseconds)
100000
10000
1000
100
10
1
0.1
0.01
CPU
Wylie-64
1 K 2 K 4 K 8 K 16 K 32 K 64 K 128 K 256 K 512 K 1 M 2 M 4 M 8 M 16 M 32 M 64 M
List Size

Helman JáJá Algorithm

- Wyllie's algorithm is work suboptimal at ***O (n log n)***
- Helman JáJá is based on sparse ruling set approach from Reid-Miller
- Originally devised for Symmetric multiprocessor systems with low processor count.
- Algorithm of choice for all recent work in this field
- Worst Case runtime is O(log n + n/p) and O(n) work.

Helman-JáJá (Contd.)

- Helman JáJá algorithm originally devised for SMP with low processor count
- Splits a list into smaller sublists, computes local rank of each sublist and stores it into a smaller, new list.
- Perform prefix sum on the new list
- Recombine the global prefix sum of the new list with the local ranks of the original list.

Successor Array	2	4	8	1	9	3	7	-	5	6

Step 1. Select **Splitters** at equal intervals

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	0	0	0	0	0	0	0	0	0

Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	0	1	0	1	1	0	0	0	1

Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	0	1	2	0	1	2	0	0	1

Step 2. **Traverse** the List until the next splitter is met and **increment** local ranks as we progress

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 3. **Stop** When all elements have been assigned a local rank

	1	2	3	4	5	6	7	8	9	10
Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of it's predecessor

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 4. **Create** a new list of splitters which contains a **prefix value** that is equal to the local rank of it's predecessor

New List Successor Array	2	-	1
Global Ranks	0	4	2

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 5. Scan the global ranks array **sequentially**

New List Successor Array	2	-	1
Global Ranks	0	4	2

→

Successor Array	2	-	1
After Ranking	0	6	2

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List

Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	0	0	0	0	0	0	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List

Successor Array	2	-	1
Global Ranks	0	4	2

After Ranking

2	-	1
0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	0	1	0	0	0	0	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	0	1	0	0	0	2	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List

Successor Array	2	-	1
Global Ranks	0	4	2

→

Successor Array	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	0	0	0	2	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List			
Successor Array	2	-	1
Global Ranks	0	4	2

After Ranking			
Successor Array	2	-	1
Global Ranks	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	0	0	2	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List

Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	0	3	2	0	0	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	0	3	2	0	1	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List

Successor Array	2	-	1
Global Ranks	0	4	2

→

2	-	1
0	6	2

After Ranking

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	2	0	1	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List			
Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	2	9	1	0

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List			
Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	2	9	1	7

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Step 6. Add the global ranks to the corresponding local ranks to get the final rank of the list.

New List			
Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

Local Ranks	0	3	1	2	0	1	2	3	0	1
Final Ranks	0	5	1	4	6	3	2	9	1	7

Modifying the algorithm for GPU

- Step 5 is a sequential ranking step.
- When we choose log n splitters, we reduce the list to n/log n, which is still large amount of sequential work
- By Amdahl's law, this is a bottleneck for parallel speedup. More so in the case of GPU.

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Make step 5 **recursive** to allow the GPU to continue processing the list in parallel

New List			
Successor Array	2	-	1
Global Ranks	0	4	2

Successor Array	2	-	1
After Ranking	0	6	2

Successor Array	2	4	8	1	9	3	7	-	5	6
Local Ranks	0	3	1	2	0	1	2	3	0	1

Make step 5 **recursive** to allow the GPU to continue processing the list in parallel

New List			
Successor Array	2	-	1
Global Ranks	0	4	2

→

	2	-	1
After Ranking	0	6	2

↓

Process this list again using the algorithm and reduce it further.

GPU Implementation

- Each phase is coded as separate GPU *kernel*
  - Since each step requires global synchronization.
- Splitter Selection
  - Each thread chooses a splitter
- Local Ranking
  - Each thread traverses its corresponding sublist and get the global ranks
- Recursive Step
- Recombination Step
  - Each thread adds the global and local ranks for each element

When do we stop?

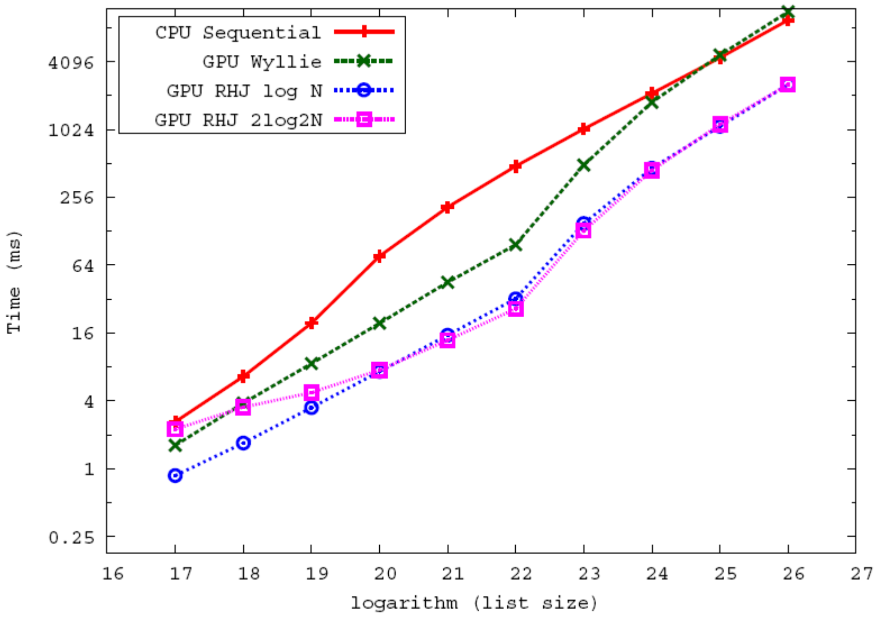
- Convergence can be met until list size is 1
- We also have the option to send a small list to CPU or Wyllie's algorithm so that it can be processed faster than on this algorithm.
- May save about 1% time

Choosing the right amount of splitters

- Notice that choosing splitters in a random list yields uneven sublists
- We can attempt to load balance the algorithm by varying the no. of splitters we choose.
- $n/\log n$ works for small lists, $n/2 \log^2 n$ works well for lists > 1 M.

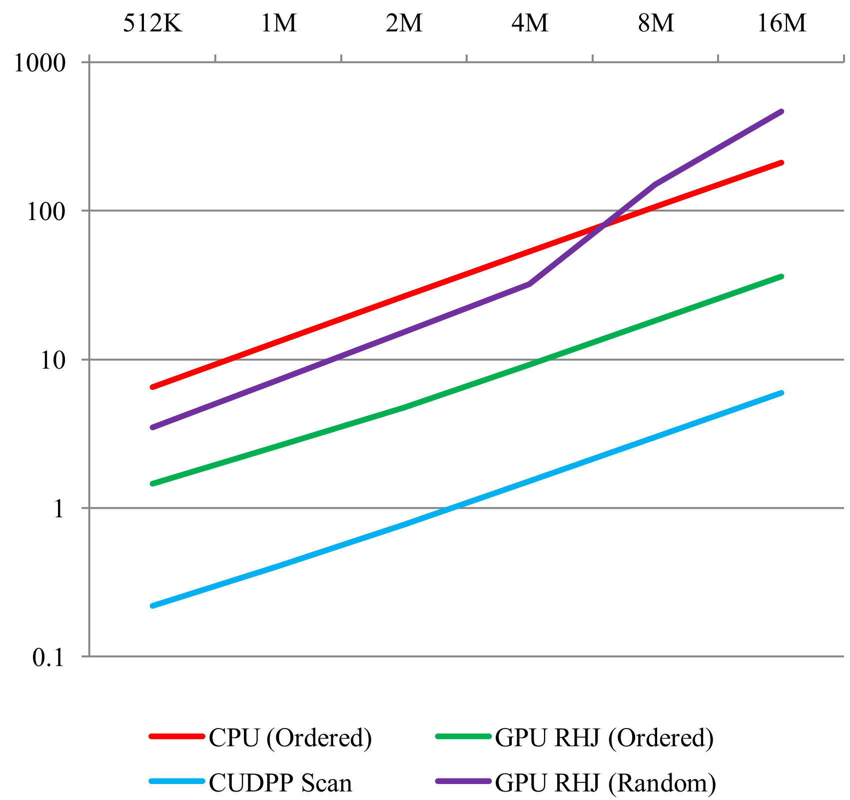
Results

- Significant Speedup over sequential algorithm on CPU ~ 10x
- Wylie's algorithm works best for small lists < 512 K
- GPU RHJ works well for large lists
- 2 log 2N works well for lists > 1M



Ordered Lists

- Perform significantly faster than random lists.
- Data locality is automatically taken advantage of by the global memory access hardware
- Compared with GPU ordered scan.



Other Irregular Applications

- Graph Algorithms:
  - Shortest path
  - Breadth-First Search
  - Spanning Tree, etc.
  - Etc
- Many others

General Graph Algorithms

1. General Graph Algorithms:
   - Breadth First Search
   - ST- Connectivity
   - Single Source Shortest Paths
   - All Pairs Shortest Path
   - Minimum Spanning Tree
   - Max Flow
2. Randomness in the graph posses great difficulty in utilizing the hardware resources.
3. Connectivity is unknown.
4. Graph Representation is not trivial.

Singular Value Decomposition

Work with Sheetal Lahabar

Appeared in IEEE IPDPS. Rome. June 2009.

Problem Statement

- SVD on GPU

SVD of matrix $A_{(mxn)}$ for $m>n$

$$\mathbf{A = U \Sigma V}$$

U and *V* are orthogonal and $\mathbf{\Sigma}$ is a diagonal matrix

Motivation

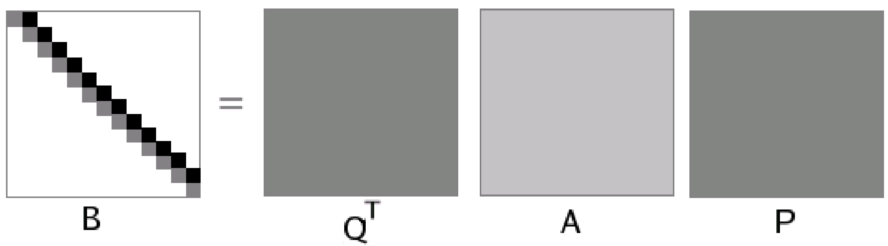
- SVD has many applications
- High computational complexity
- GPUs have high computing power
  - Teraflop performance
- Exploit the GPU for high performance

Methods

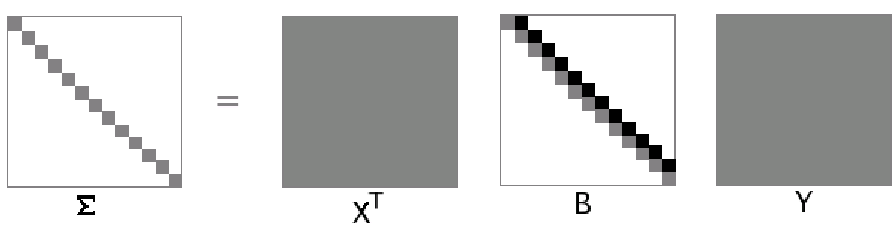
- SVD algorithms
  - Golub Reinsch
    (Bidiagonalization and Diagonalization)
  - Hestenes algorithm(Jacobi)
- Golub Reinsch method
  - Simple and compact
  - Maps well to the GPU
  - Popular in numerical libraries

Golub Reinsch algorithm

- Bidiagonalization:
  - Series of householder transformations



- Diagonalization:
  - Implicitly Shifted QR iterations

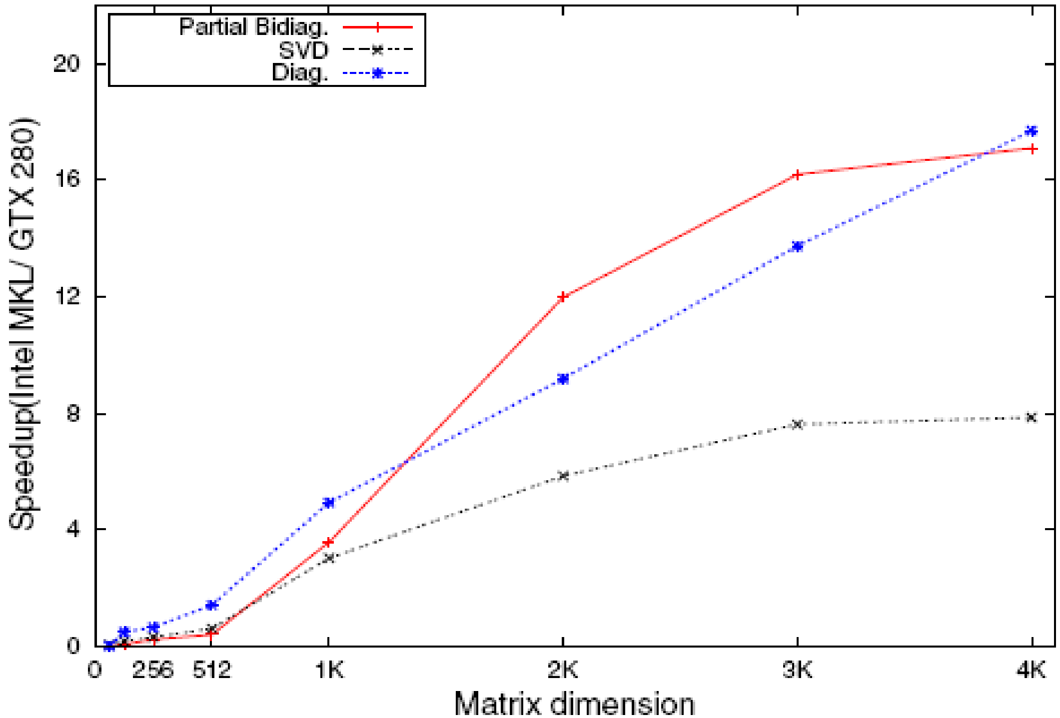


SVD

- Overall algorithm
  - $B = Q^{T}AP$
    Bidiagonalization of A to B
  - $\Sigma = X^{T}BY$
    Diagonalization of B to Σ
  - $U = QX,\ V^{T} = (PY)^{T}$
    Compute orthogonal matrices U and V^{T}
- Complexity: $O(mn^2)$ for $m>n$

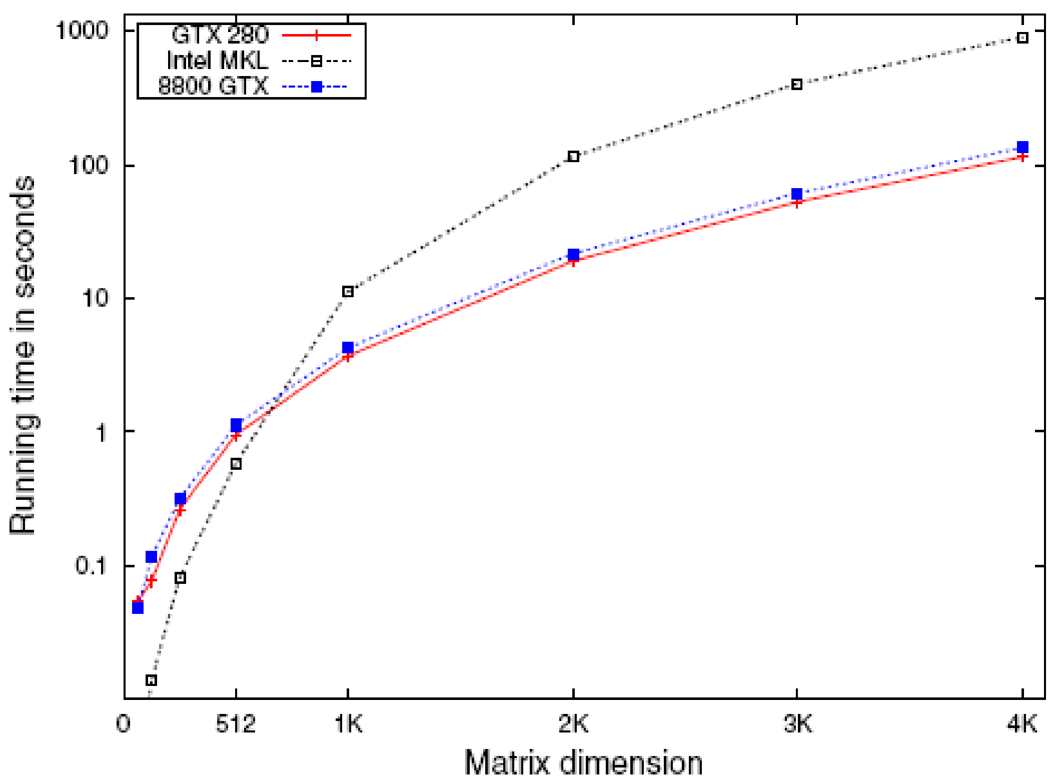
Results

- Intel 2.66 GHz Dual Core CPU used
- Speedup on NVIDIA GTX 280:
  - 3-8 over MKL LAPACK
  - 3-60 over MATLAB



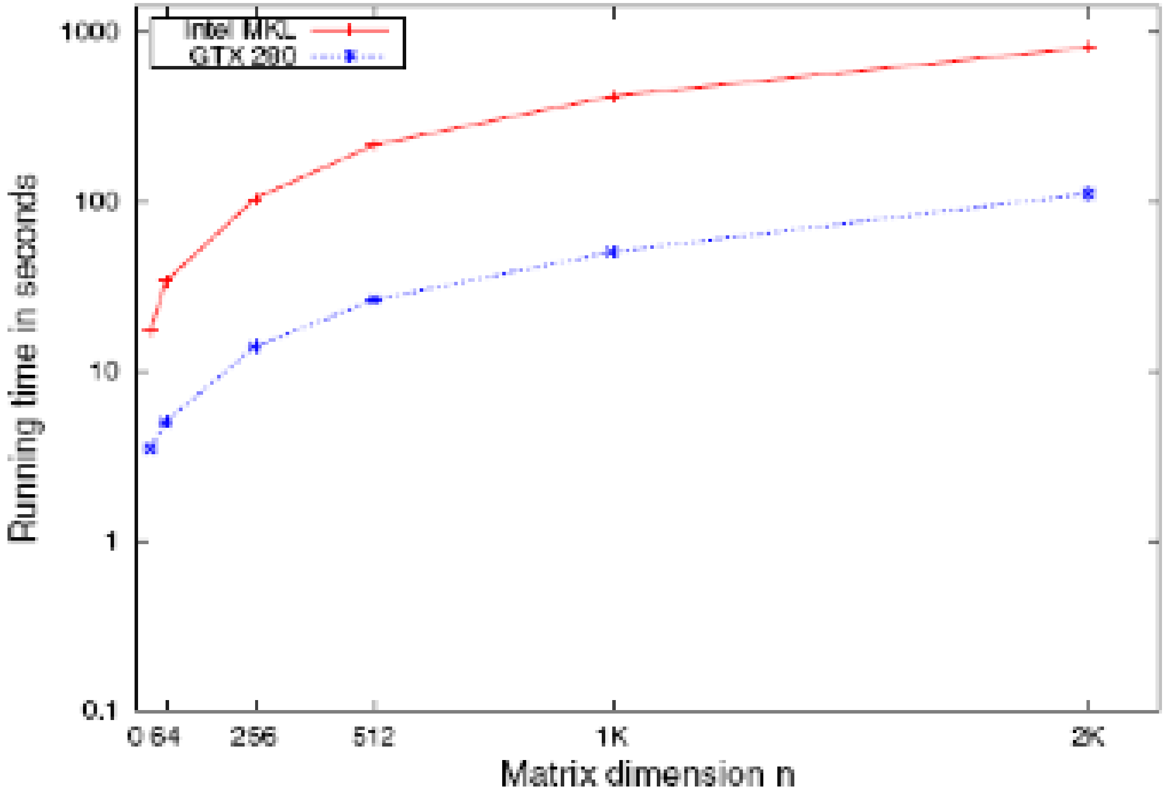
Contd…

- CPU outperforms for smaller matrices
- Speedup increases with matrix size



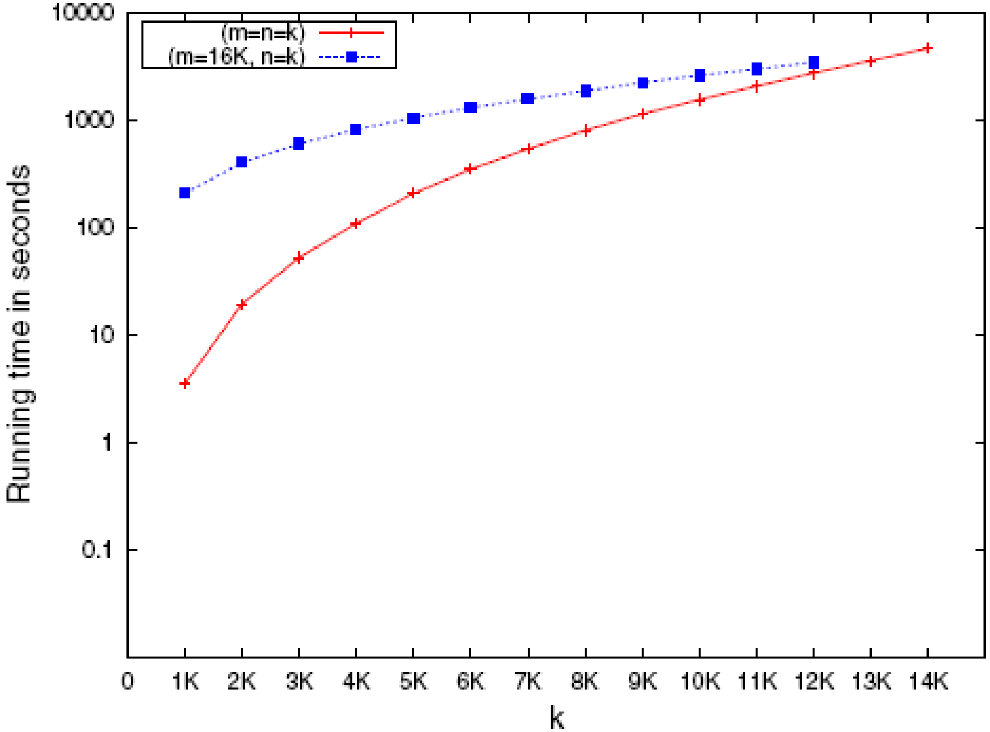
Contd…

- SVD timing for rectangular matrices (m=8K)
  - Speedup increases with varying dimension



Contd…

- SVD of upto 14K x 14K on Tesla S1070 takes 76 mins on GPU
- 10K x 10K SVD takes 4.5 hours on CPU, 25.6 minutes on GPU



Contd…

- Yamamoto achieved a speedup of 4 on CSX600 for very large matrices
- Bobda report the time for $10^6 \times 10^6$ matrix which takes 17 hours
- Bondhugula report only the partial bidiagonalization time

Timing for Partial Bidiagonalization

- Speedup:1.5-16.5 over Intel MKL
- CPU outperforms for small matrices
- Timing comparable to Bondhugula (11 secs on GTX 280 compared to 19 secs on 7900)

Time in secs

SIZE	Bidiag. GTX 280	Partial Bidiag. GTX 280	Partial Bidiag. Intel MKL
512 x 512	0.57	0.37	0.14
1K x 1K	2.40	1.06	3.81
2K x 2K	14.40	4.60	47.9
4K x 4K	92.70	21.8	361.8

Timing for Diagonalization

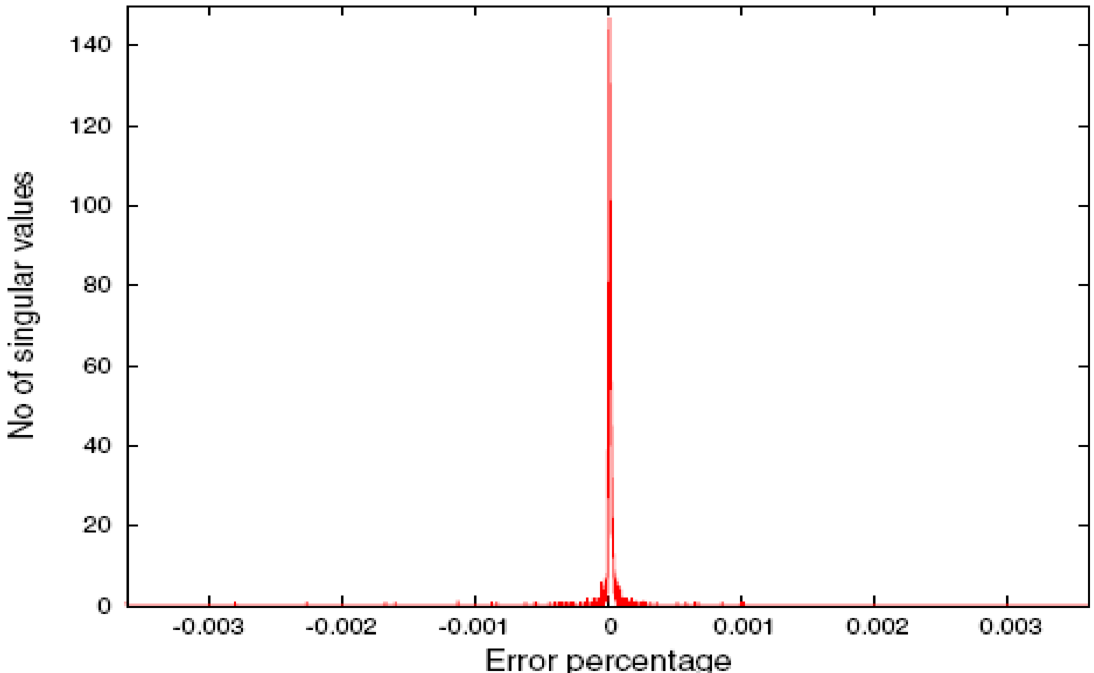
- Speedup:1.5-18 over Intel MKL
- Maximum Occupancy: 83%
- Data coalescing achieved
- Performance increases with matrix size
- Performs well even for small matrices

SIZE	Diag. GTX 280	Diag. Intel MKL
512 x 512	0.38	0.54
2K x 2K	5.14	49.1
4K x 4K	20	354
8K x 2K	8.2	100

Time in secs

Limitations

- Limited double precision support
- High performance penalty
- Discrepancy due to reduced precision



$m=3K, n=3K$

Contd…

- Max singular value discrepancy = 0.013%

  Average discrepancy < 0.00005%
- Average discrepancy < 0.001% for U and V^T
- Limited by device memory

Regular Algorithms on CUDA

IIIT Hyderabad

Mapping an Image on CUDA

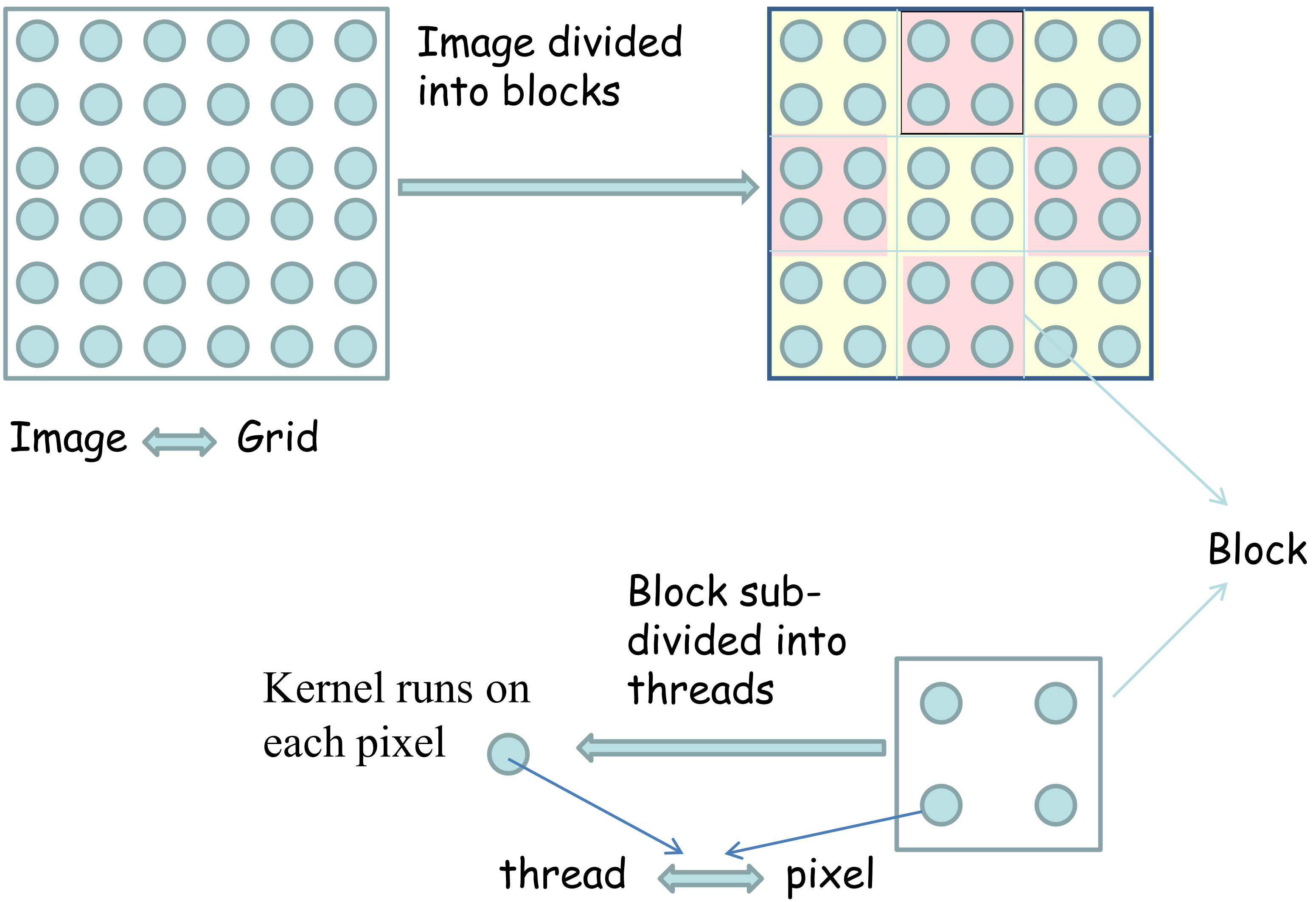
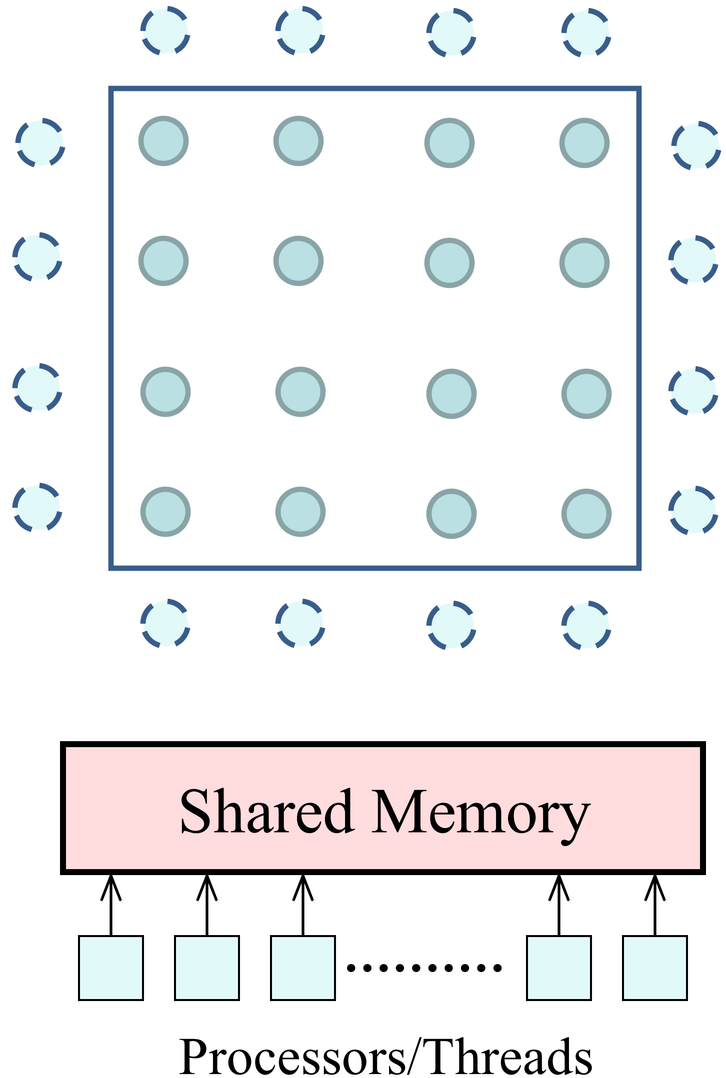


Image Processing, Filtering

- Thread accesses its pixel data using thread to pixel mapping
  - Read is efficient: Coalesced
  - Process each pixel independently and write results
- 2D Filtering: Keep block values + neighbouring rows and cols in shared memory
  - Coalesced access to bring to SM
  - Synchronize threads of block to ensure loading
  - A thread computes its pixel's output value from shared memory
  - Write results coalesced



Mean filtering

```
float *shMem = (float *) &sharedMem[0]  // Pointer
// Computer image coordinates
x = blockIdx.x*blockDim.x + threadIdx.x
y = blockIdx.y*blockDim.y + threadIdx.y
// Compute a local coordinate within block
localIndex = threadIdx.x+threadIdx.y*blockDim.x
// Copy own portion to shared memory
shMem[localIndex] = globalImage[y*width + x]
__syncthreads() // Wait till all copying is done
// Compute the required output and copy back
g_odata[y*width + x] = meanGreyValue()
```

Mean Computation

```
float meanValue = 0.0
// Compute the average of the 9 pixels
for (int i=0; i<3; i++)
  for (int j=0; j<3; j++)
      indx = (threadIdx.x – i) + (threadIdx.y – j)*blockDim.x
      meanValue += shMem[indx]
meanValue /= 9.0
```

Note:

- Borders are not handled properly.
- Needs if-then-else to process borders specially
- Divergence: Different threads doing different actions
- Always suffers in performance on SIMD architectures
- Intra-warp divergence only for CUDA

Image Rotation

- Rotate by angle θ.
- $x' = x \cos\theta - y \sin\theta$
  $y' = x \sin\theta + y \cos\theta$
- Fractional coordinates!
- Think reverse and interpolate
- $x = x' \cos\theta + y' \sin\theta$
  $y = x' \sin\theta - y' \cos\theta$
- Can use texture memory to get interpolation

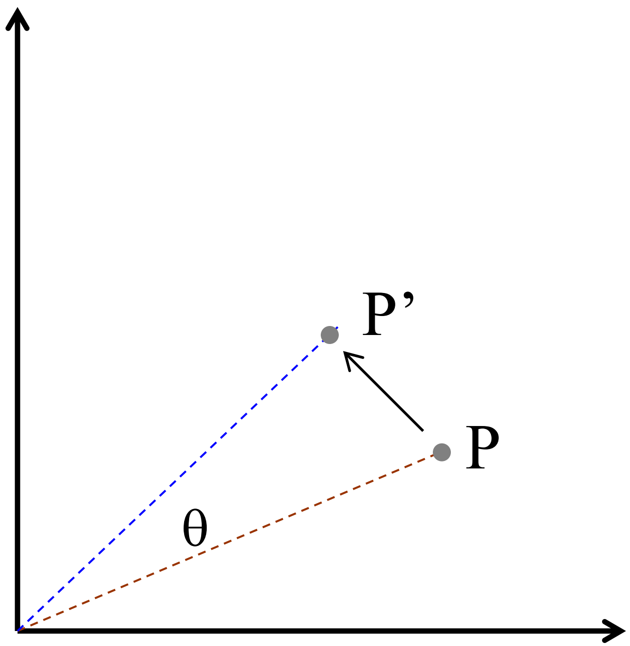


Image Rotation

```
// image/texture coordinates
    x = blockIdx.x*blockDim.x + threadIdx.x
    y = blockIdx.y*blockDim.y + threadIdx.y;
    u = x / (float) width
    v = y / (float) height;
// transform coordinates
    u -= 0.5f, v -= 0.5f;
    tu = u*cosf(theta) + v*sinf(theta) + 0.5f
    tv = v*cosf(theta) - u*sinf(theta) + 0.5f;
// read from texture and write to global memory
    g_odata[y*width + x] = tex2D(tex, tu, tv)

// Interpolation:        img[i,j] (1-b) (1-c) + img[i,j+1] (1-b) c +
//                       img[i+1,j] (1 - b) c + img[i+1,j+1] b c
```

Data-Parallel Computation

- Kernels operate on data elements
  - Little interaction between data elements
  - Simple model. **Think like data elements**. Know little!
- Also called
  - Stream computing
  - Throughput computing
- Application areas
  - Signal processing, Image processing
  - (Large) matrix operations
  - Scientific computing with large data
    - Molecues, fluid flow, ….

Thank you!