Lecture 15
Data-Parallel Algorithms

EEC 171 Parallel Architectures

John Owens
UC Davis

Credits

-
- This lecture is primarily derived from tutorials at Supercomputing 2006, 2007, and 2009, and a talk at Dagstuhl in 2010, including material from Mark Harris, Nathan Bell, and Michael Garland (NVIDIA).

Summary: three key ideas

- Use many "slimmed down cores" to run in parallel

- Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
  - The CUDA/OpenCL model expresses programs as scalar programs and has implicit sharing managed by hardware

- Avoid latency stalls by interleaving execution of many groups of fragments
  - When one group stalls, work on another group

Programming Model Big Idea #1

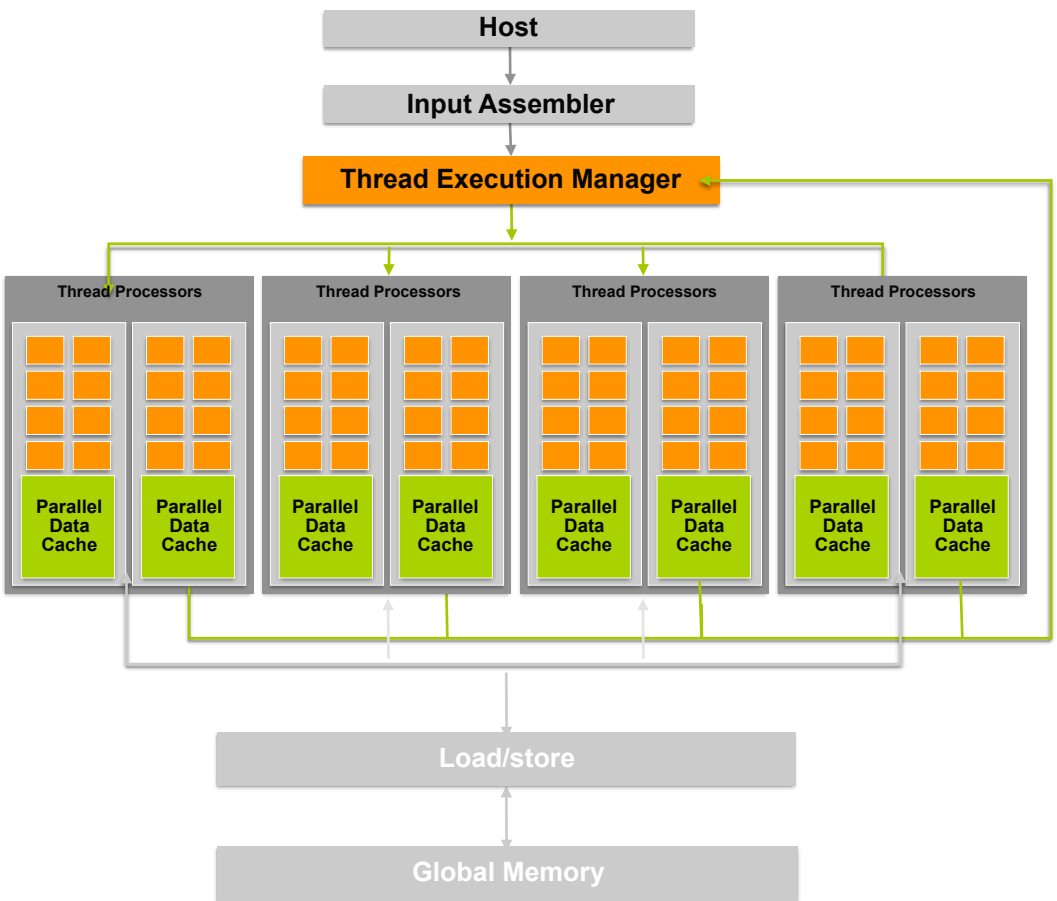
- One thread per data element.
- Doesn't this mean that large problems will have millions of threads?

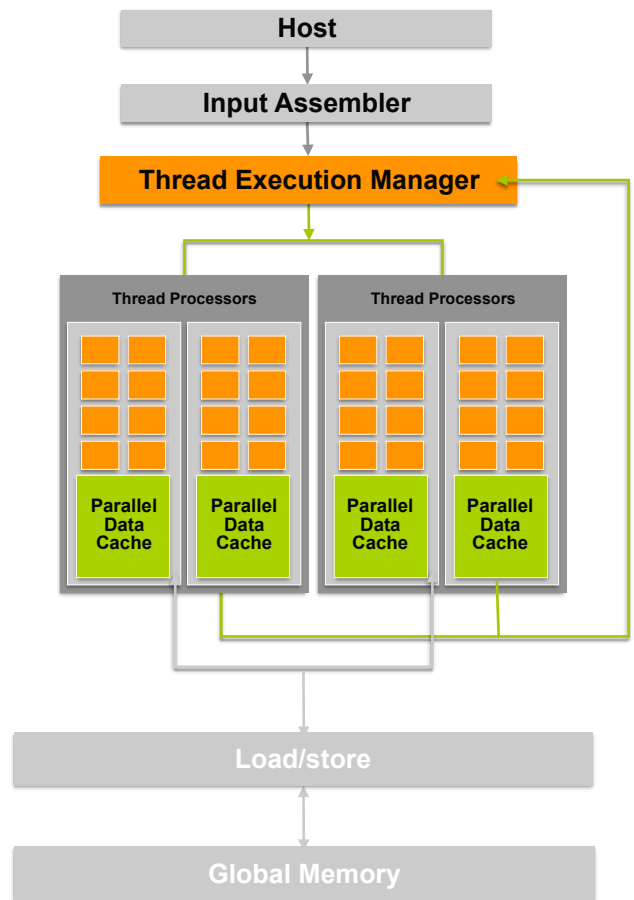
Programming Model Big Idea #2

- Write one program.
- That program runs on ALL threads in parallel.
- Software terminology here is "SPMD": single-program, multiple-data.
- Hardware terminology here is "SIMT": single-instruction, multiple-thread.
    - Roughly: SIMD means many threads run in lockstep; SIMT means that some divergence is allowed and handled by the hardware

Scaling the Architecture

- Same program
- Scalable performance





There's lots of dimensions to scale this processor with more resources. What are some of those dimensions? If you had twice as many transistors, what could you do with them?

What *should* you do with more resources? In what dimension do you think NVIDIA will scale future GPUs?

Programming Model Big Idea #3

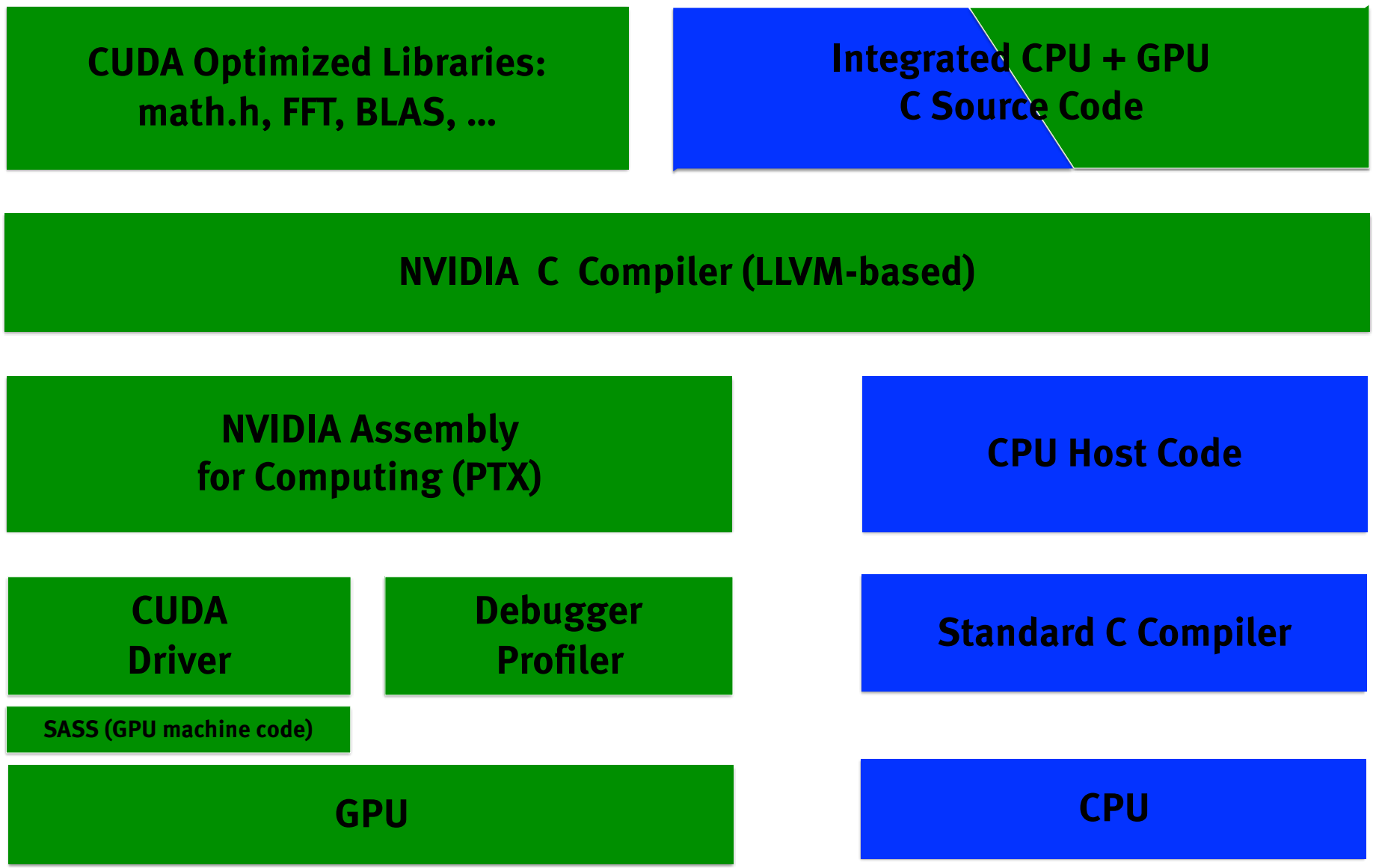
- *Scalable execution*
  - Program must be insensitive to the number of cores
  - Write one program for any number of SM cores
  - Program runs on any size GPU without recompiling

- Hierarchical execution model
  - Decompose problem into sequential steps (kernels)
  - Decompose kernel into computing parallel blocks
  - Decompose block into computing parallel threads

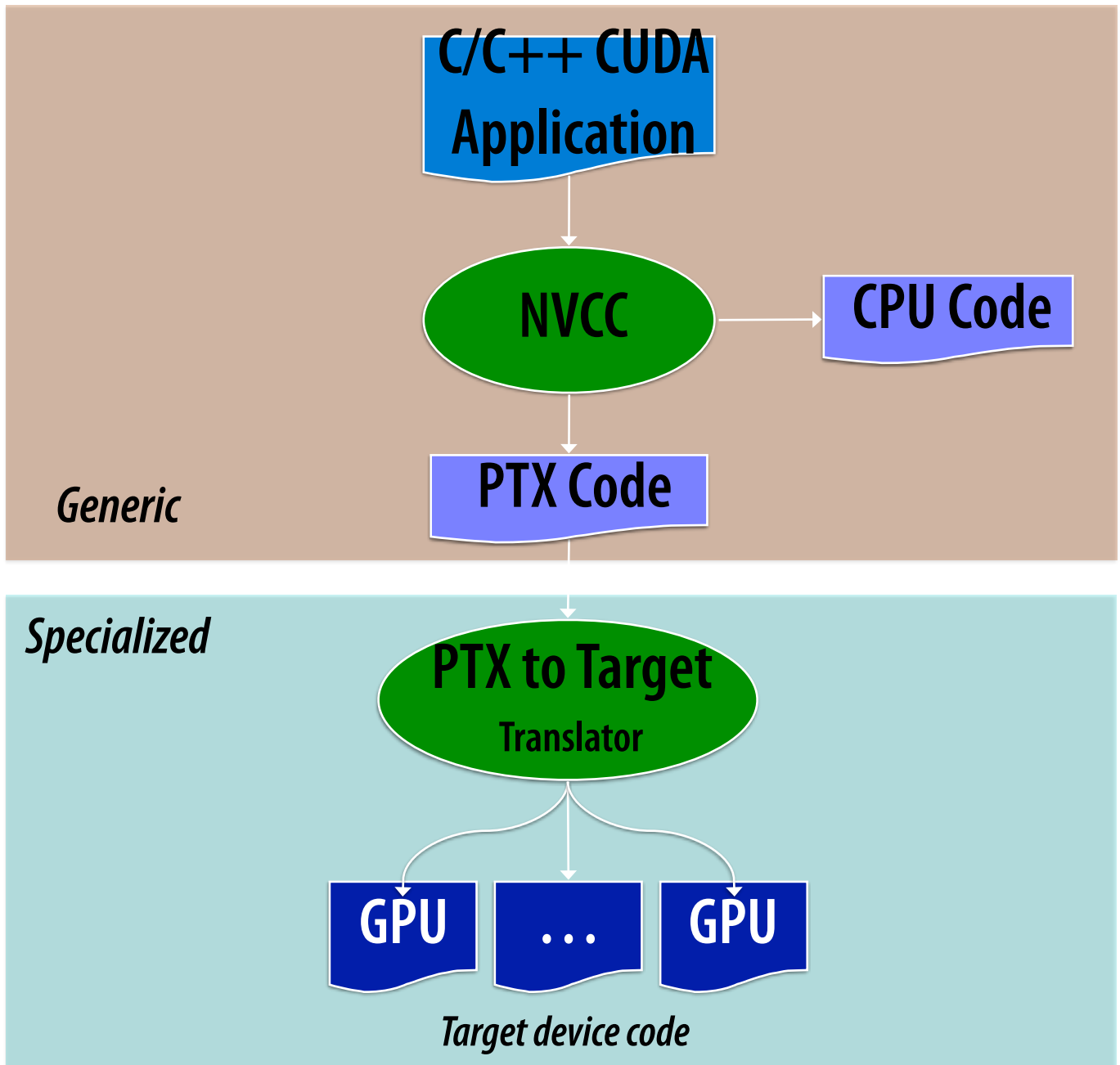
This is very important!

- Hardware distributes independent blocks to SMs as available

CUDA Software Development Kit



Compiling CUDA for GPUs



Basic Efficiency Rules

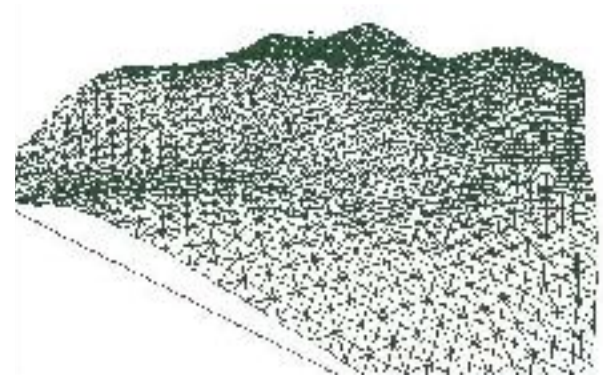
- Develop algorithms with a data parallel mindset
- Minimize divergence of execution within blocks
- Maximize locality of global memory accesses
- Exploit per-block shared memory as scratchpad
- Expose enough parallelism

Data-Parallel Algorithms

- Efficient algorithms require efficient building blocks
- Five data-parallel building blocks
  - Map
  - Gather & Scatter
  - Reduce
  - Scan
  - Sort
- Advanced data structures:
  - Sparse matrices
  - Hash tables
  - Task queues

Sample Motivating Application

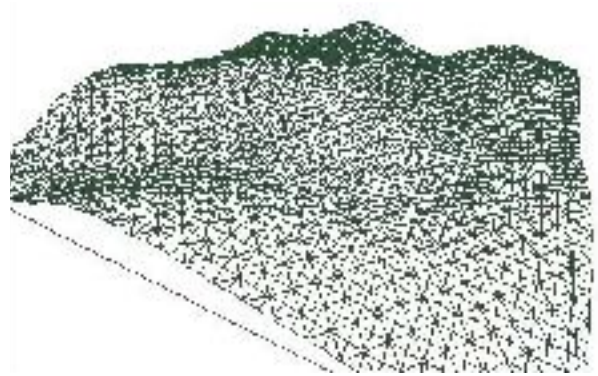
- How bumpy is a surface that we represent as a grid of samples?



- Algorithm:
  - Loop over all elements
  - At each element, compare the value of that element to the average of its neighbors (“difference”). Square that difference.
  - Now sum up all those differences.
    - But we don’t want to sum all the diffs that are o.
    - So only sum up the non-zero differences.
  - This is a fake application—don’t take it too seriously.

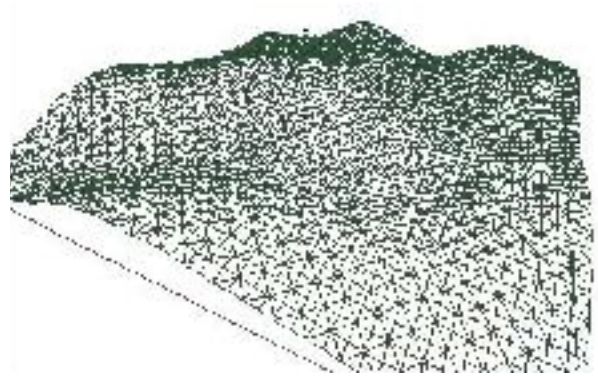
Picture courtesy http://www.artifice.com

Sample Motivating Application



```
for all samples:

    neighbors[x,y] =
       0.25 * (  value[x-1,y]+
                 value[x+1,y]+
                 value[x,y+1]+
                 value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

Sample Motivating Application

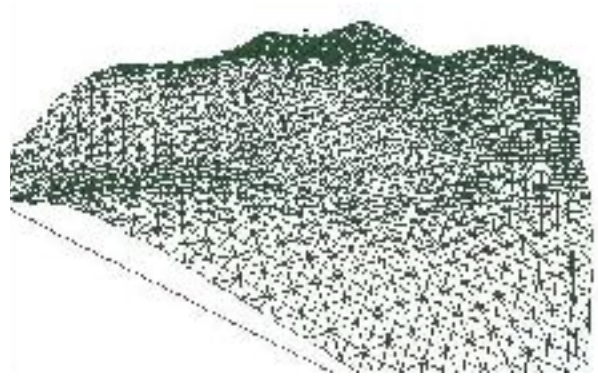


```
for all samples:

    neighbors[x,y] =
       0.25 * (  value[x-1,y]+
                 value[x+1,y]+
                 value[x,y+1]+
                 value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

The Map Operation

- Given:
  - Array or stream of data elements A
  - Function $f(x)$
- map(A, f) = applies $f(x)$ to all $a_i \in A$
- How does this map to a data-parallel processor?

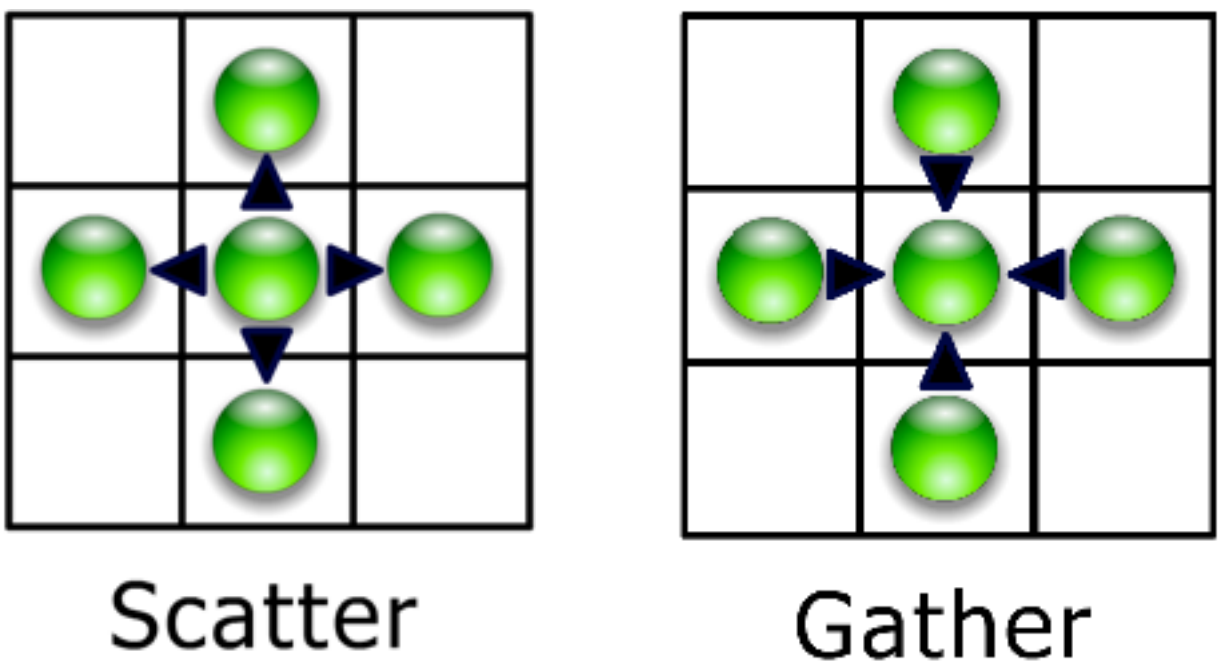
Sample Motivating Application



```
for all samples:

    neighbors[x,y] =
       0.25 * (  value[x-1,y]+
                 value[x+1,y]+
                 value[x,y+1]+
                 value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

Scatter vs. Gather

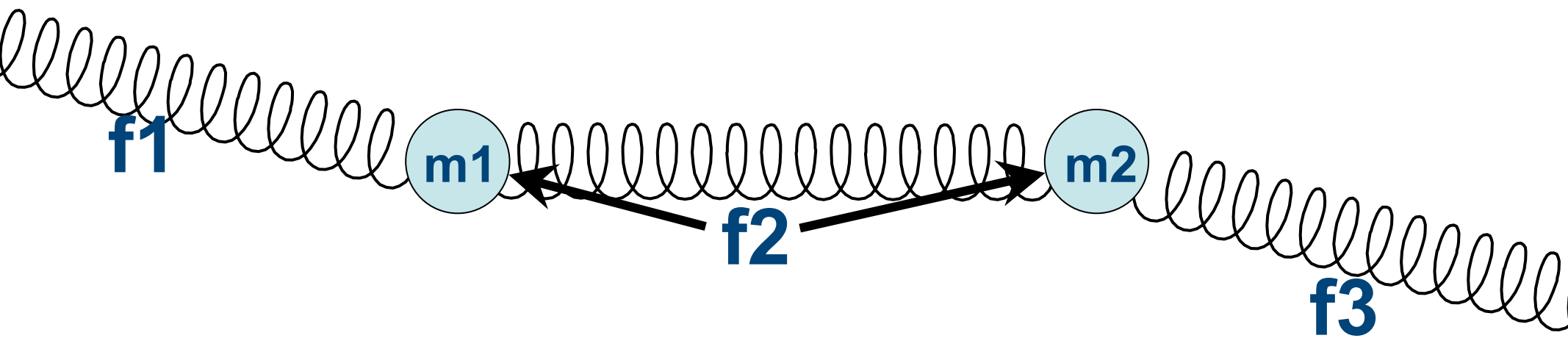
- Gather: `p = a[i]`
- Scatter: `a[i] = p`
- How does this map to a data-parallel processor?



Scatter Techniques

- Convert to Gather

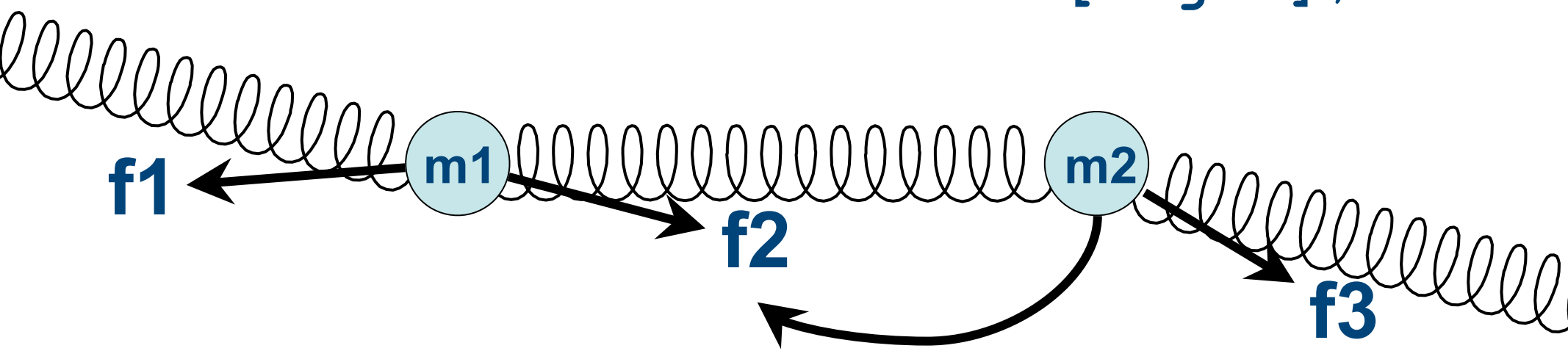
```
for each spring
    f = computed force
    mass_force[left]  += f;
    mass_force[right] -= f;
```



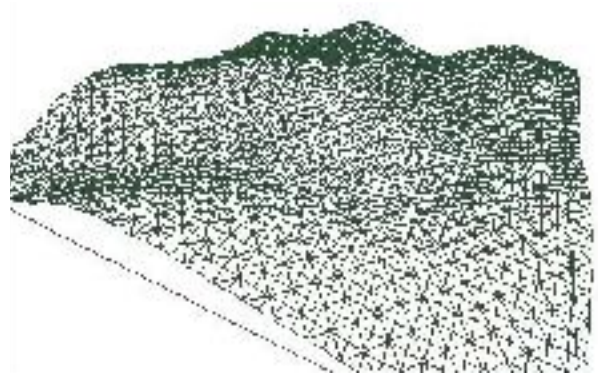
Scatter Techniques

- Convert to Gather

```
for each spring
    f = computed force
for each mass
    mass_force = f[left] -
                 f[right];
```



Sample Motivating Application



```
for all samples:

    neighbors[x,y] =
       0.25 * (  value[x-1,y]+
                 value[x+1,y]+
                 value[x,y+1]+
                 value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

Parallel Reductions

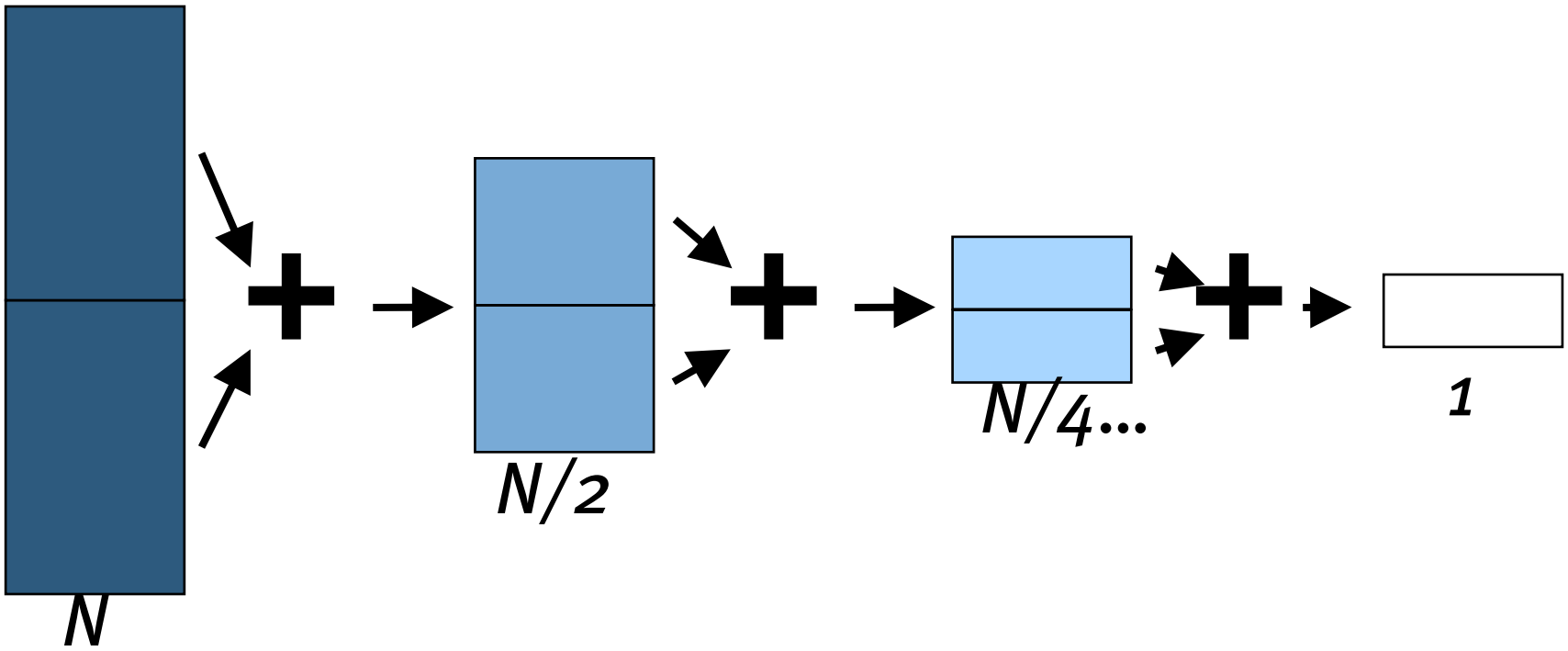
- Given:
  - Binary associative operator $\oplus$ with identity I
  - Ordered set s = [ao, a1, …, a_{n-1}] of n elements
- reduce($\oplus$, s) returns ao $\oplus$ a1 $\oplus$ … $\oplus$ a_{n-1}
- Example:
  reduce(+, [3 1 7 0 4 1 6 3]) = 25
- Reductions common in parallel algorithms
  - Common reduction operators are +, ×, min and max
  - Note floating point is only pseudo-associative

Efficiency

- Work efficiency:
  - Total amount of work done over all processors
- Step efficiency:
  - Number of steps it takes to do that work
- With parallel processors, sometimes you're willing to do more work to reduce the number of steps
- Even better if you can reduce the amount of steps and still do the same amount of work

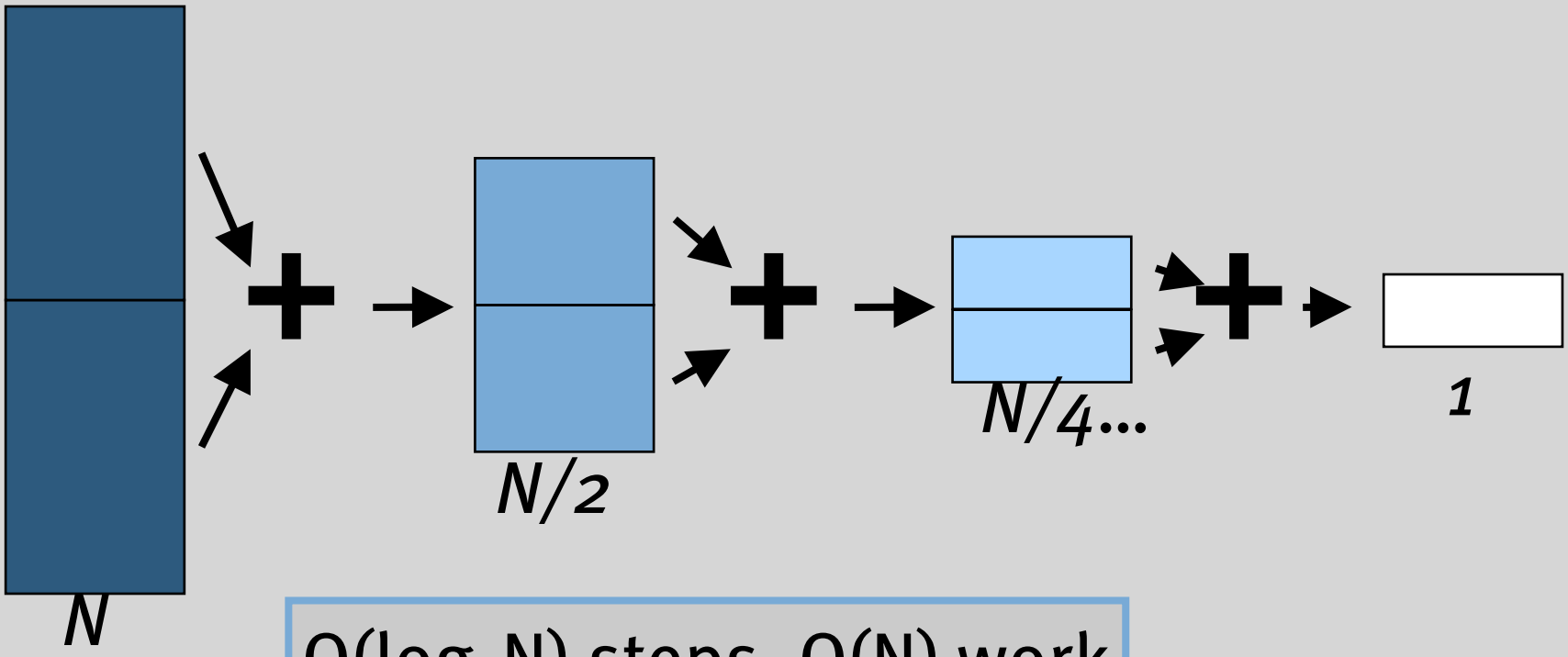
Parallel Reductions

- 1D parallel reduction:
  - add two halves of domain together repeatedly...
  - ... until we're left with a single row



Parallel Reductions

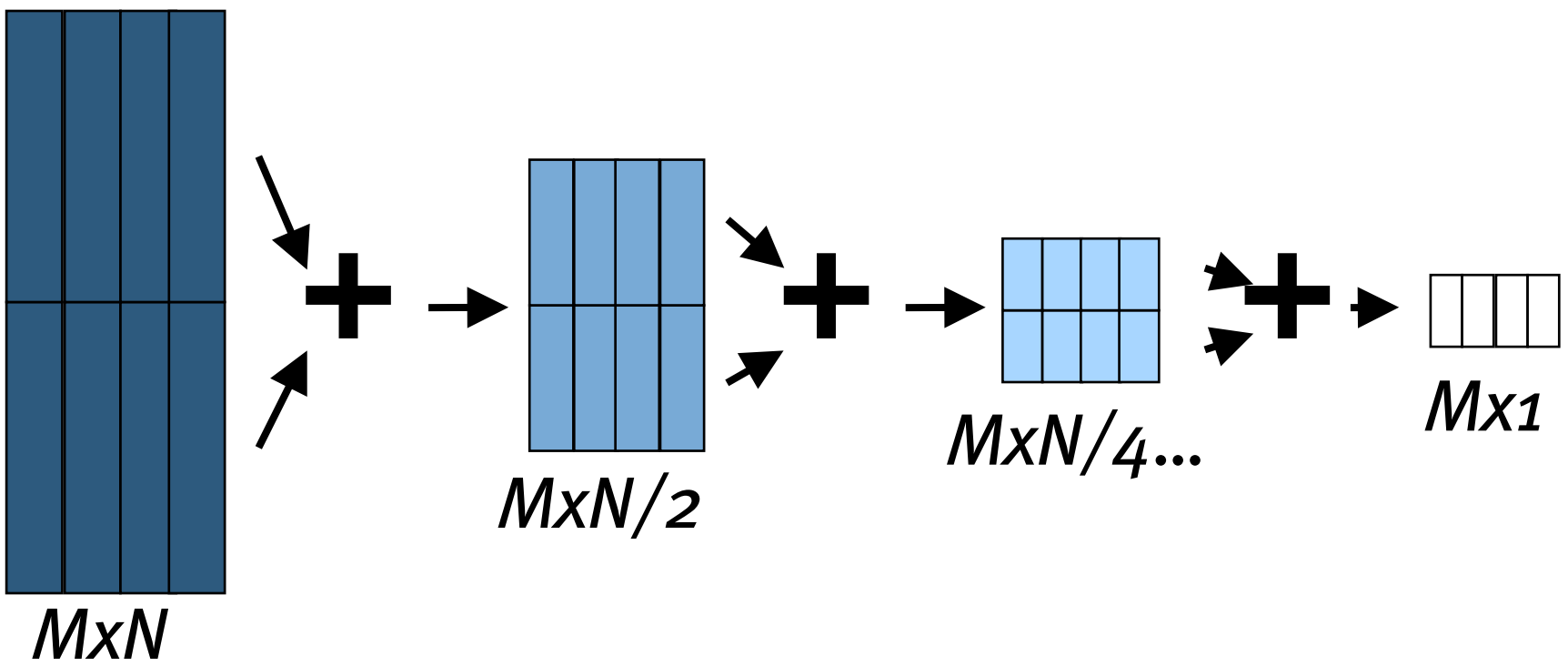
- 1D parallel reduction:
  - add two halves of domain together repeatedly...
  - ... until we're left with a single row



$O(\log_2 N)$ steps, $O(N)$ work

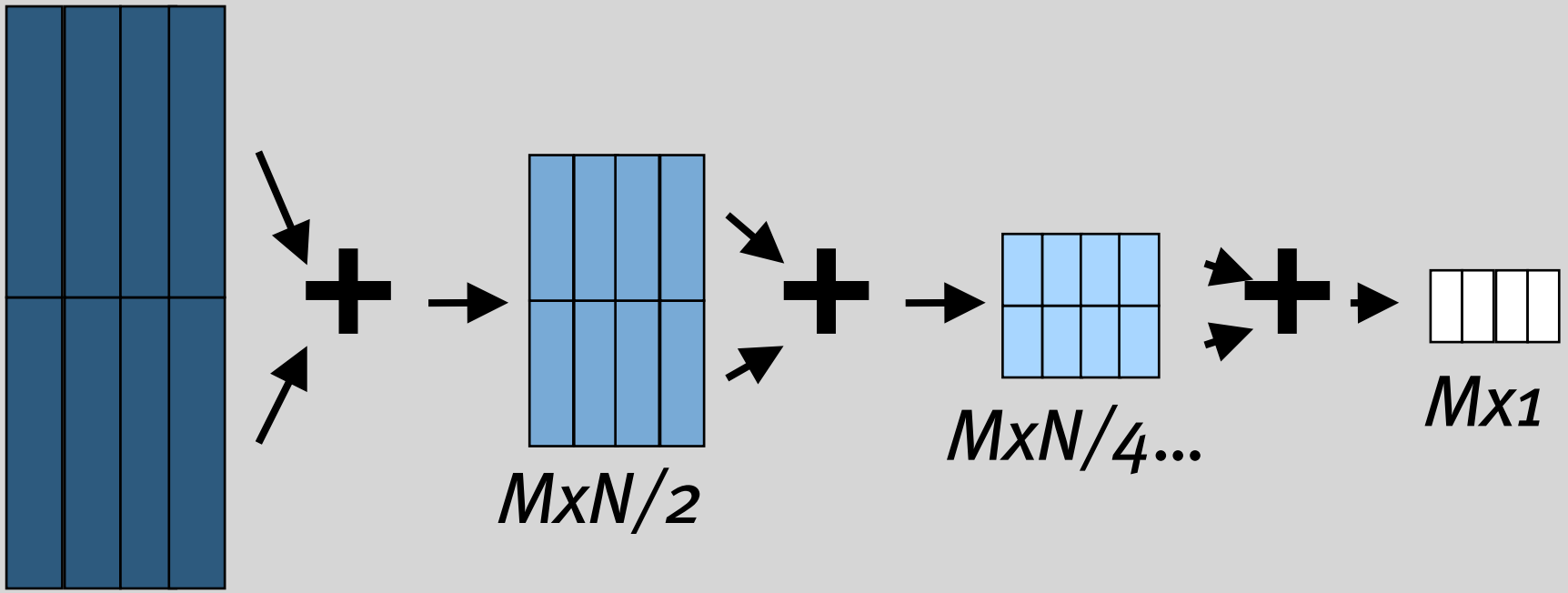
Multiple 1D Parallel Reductions

- Can run many reductions in parallel
- Use 2D grid and reduce one dimension



Multiple 1D Parallel Reductions

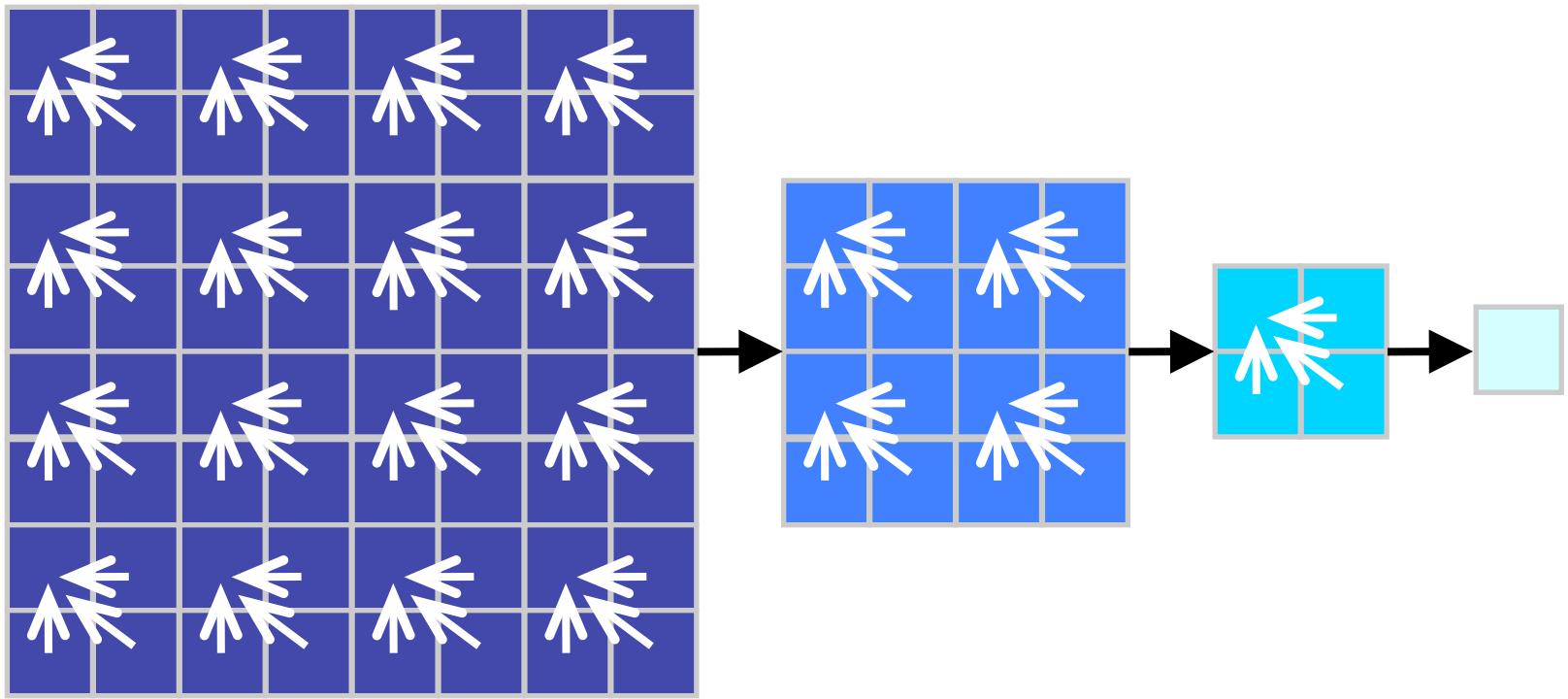
- Can run many reductions in parallel
- Use 2D grid and reduce one dimension



$O(\log_2 N)$ steps, $O(MN)$ work

2D reductions

- Like 1D reduction, only reduce in both directions simultaneously

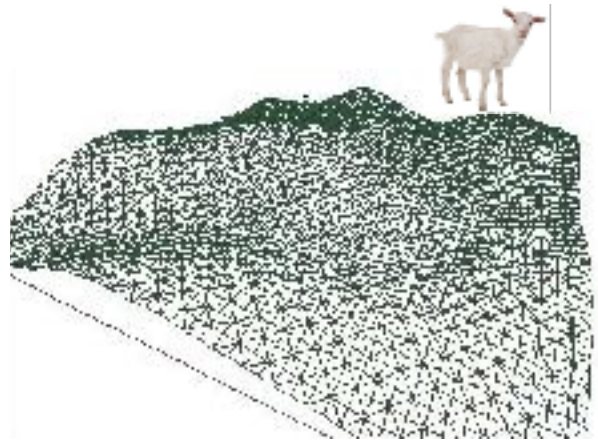


- Note: can add more than 2x2 elements per step
  - Trade per-pixel work for step complexity
  - Best perf depends on specific hardware (cache, etc.)

Parallel Reduction Complexity

- log(n) parallel steps, each step S does $n/2^S$ independent ops
  - Step Complexity is $O(\log n)$
- Performs $n/2 + n/4 + \ldots + 1 = n\text{-}1$ operations
  - Work Complexity is $O(n)$—it is work-efficient
    - i.e. does not perform more operations than a sequential algorithm
- With p threads physically in parallel (p processors), time complexity is $O(n/p + \log n)$
  - Compare to $O(n)$ for sequential reduction

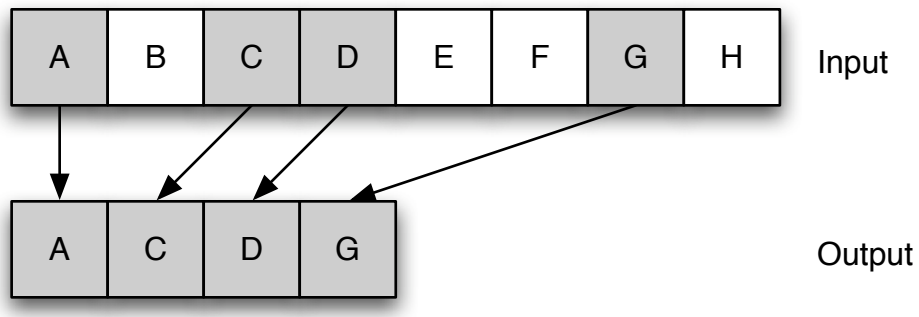
Sample Motivating Application



```
for all samples:

    neighbors[x,y] =
       0.25 * (  value[x-1,y]+
                 value[x+1,y]+
                 value[x,y+1]+
                 value[x,y-1] ) )

    diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

    result += diff

return result
```

Stream Compaction

- Input: stream of 1s and 0s
  [1 0 1 1 0 0 1 0]
- Operation:“sum up all elements before you”
- Output: scatter addresses for “1” elements
  [0 1 1 2 3 3 3 4]
- Note scatter addresses for red elements are packed!

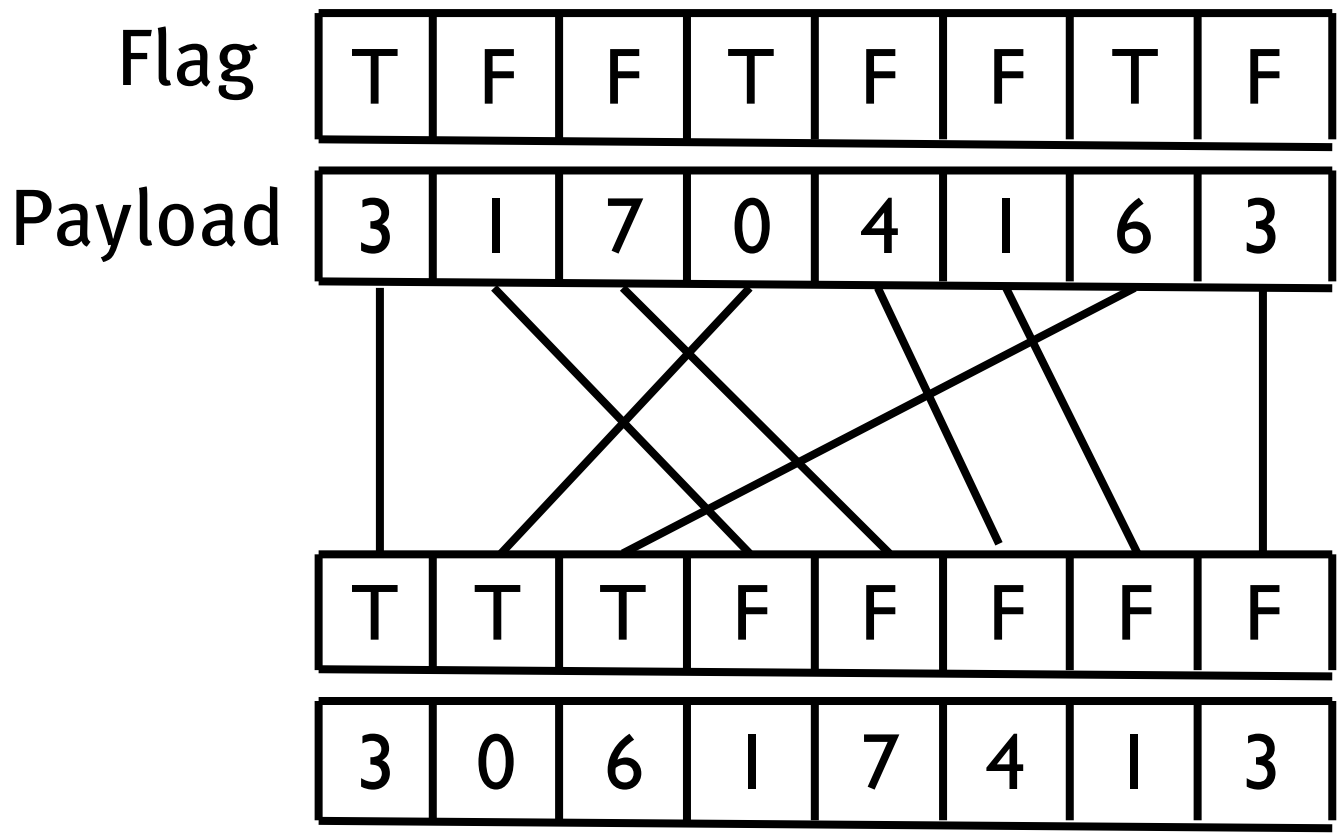


Common Situations in Parallel Computation

- Many parallel threads that need to partition data
  - Split
- Many parallel threads and variable output per thread
  - Compact / Expand / Allocate
- More complicated patterns than one-to-one or all-to-one
  - Instead all-to-all

Split Operation

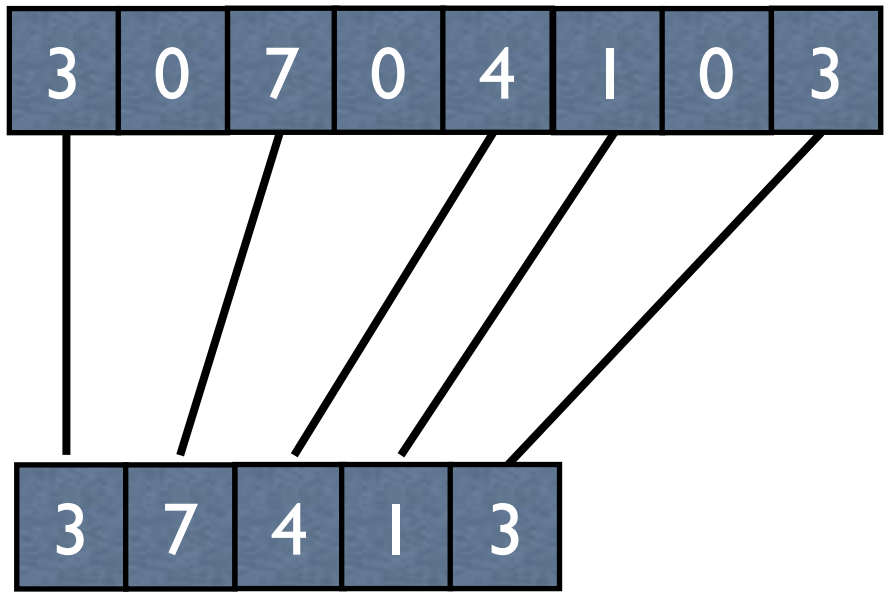
- Given an array of true and false elements (and payloads)



- Return an array with all true elements at the beginning
- Examples: sorting, building trees

Variable Output Per Thread: Compact

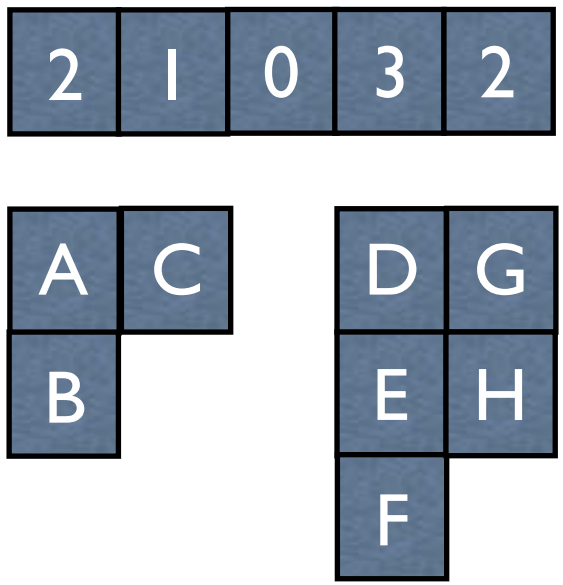
- Remove null elements



- Example: collision detection

Variable Output Per Thread

- Allocate Variable Storage Per Thread



- Examples: marching cubes, geometry generation

“Where do I write my output?”

- In all of these situations, each thread needs to answer that simple question
- The answer is:
- “That depends on how much the other threads need to write!”
  - In a serial processor, this is simple
- “Scan” is an efficient way to answer this question in parallel

Parallel Prefix Sum (Scan)

- Given an array $A = [a_0, a_1, \ldots, a_{n-1}]$
  and a binary associative operator $\oplus$ with identity I,

- $\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$

- Example: if $\oplus$ is addition, then scan on the set
  - [3 1 7 0 4 1 6 3]
- returns the set
  - [0 3 4 11 11 15 16 22]

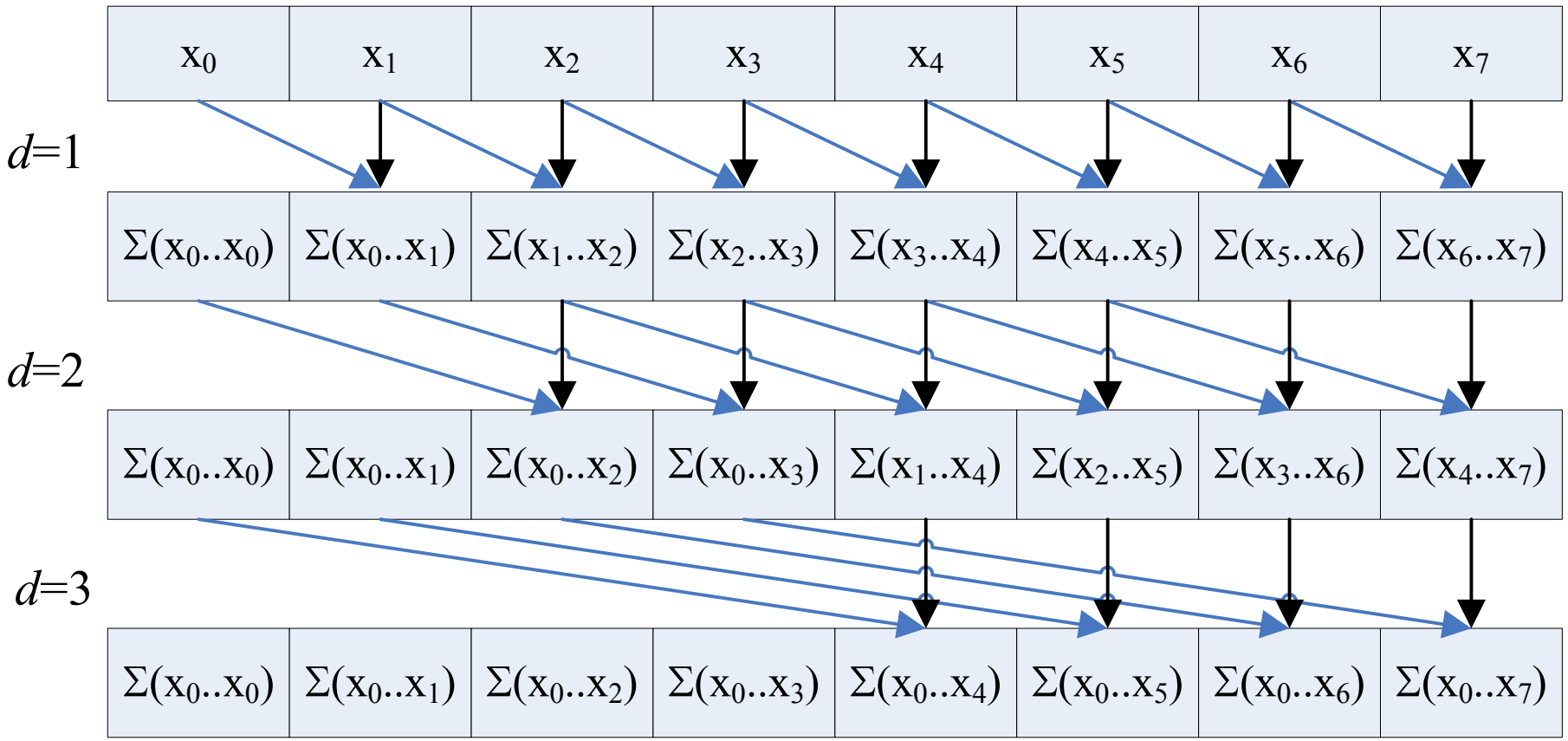
Segmented Scan

- Example: if $\oplus$ is addition, then scan on the set
  - [3 1 7 | 0 4 1 | 6 3]
- returns the set
  - [0 3 4 | 0 0 4 | 0 6]
- Same computational complexity as scan, but additionally have to keep track of segments (we use head flags to mark which elements are segment heads)
- Useful for *nested data parallelism* (quicksort)

Quicksort

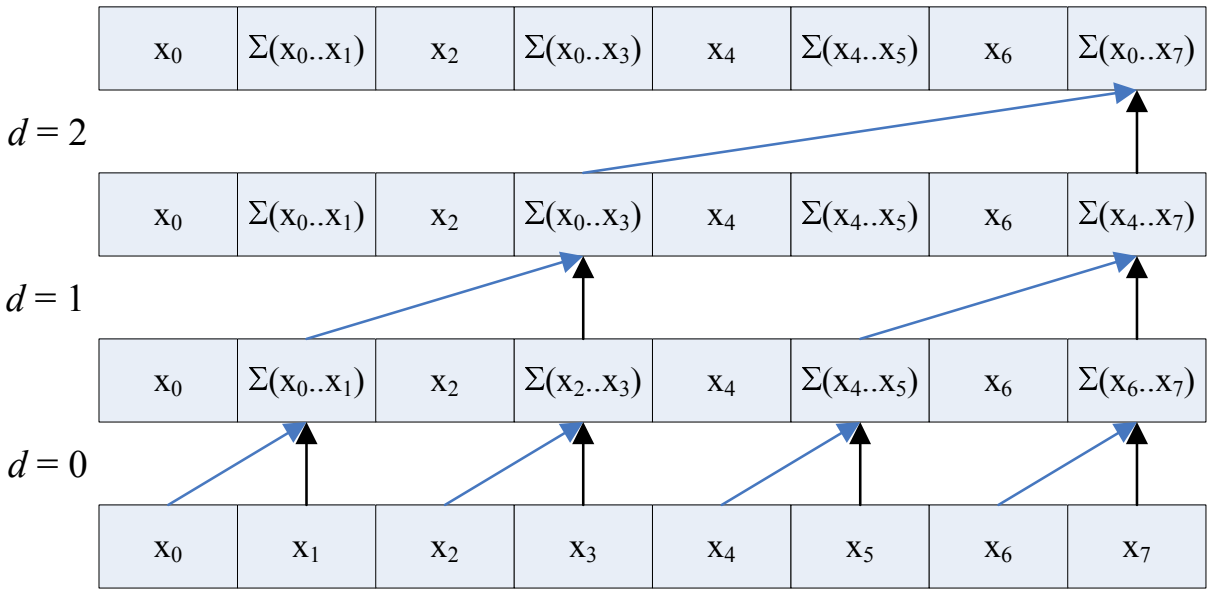
```
[5 3 7 4 6]   # initial input
[5 5 5 5 5]   # distribute pivot across segment
[f f t f t]   # input > pivot?
[5 3 4][7 6]  # split-and-segment
[5 5 5][7 7]  # distribute pivot across segment
[t f f][t f]  # input >= pivot?
[3 4 5][6 7]  # split-and-segment, done!
```

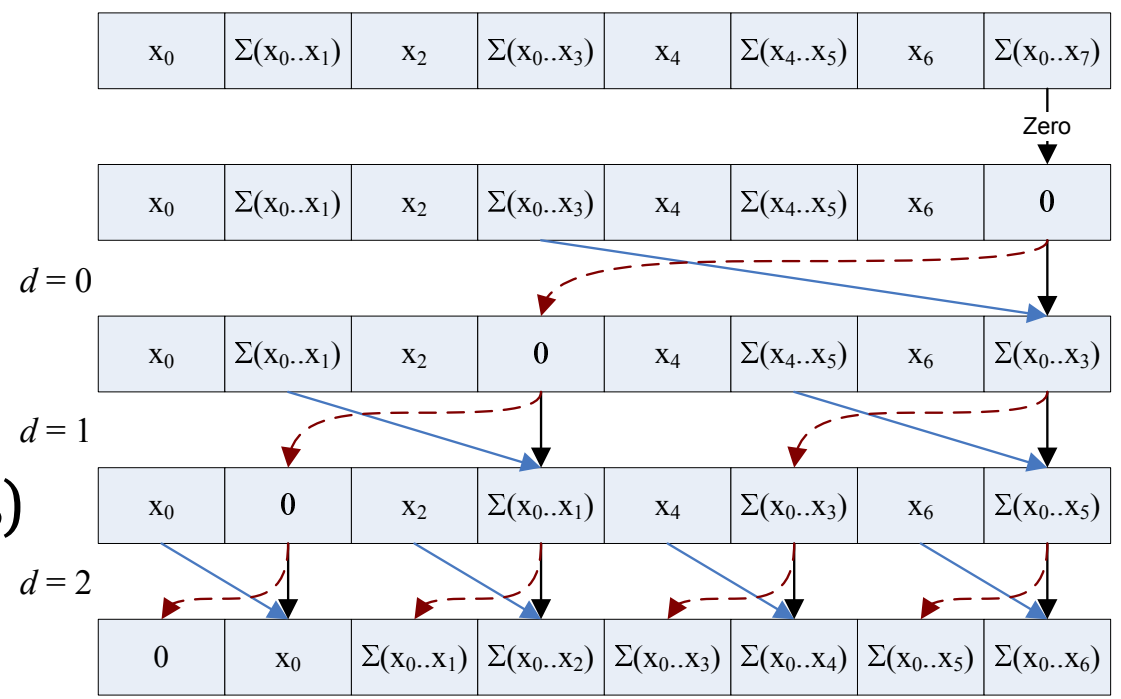
$O(n \log n)$ Scan



- Step efficient (log n steps)
- Not work efficient (n log n work)

$O(n)$ Scan

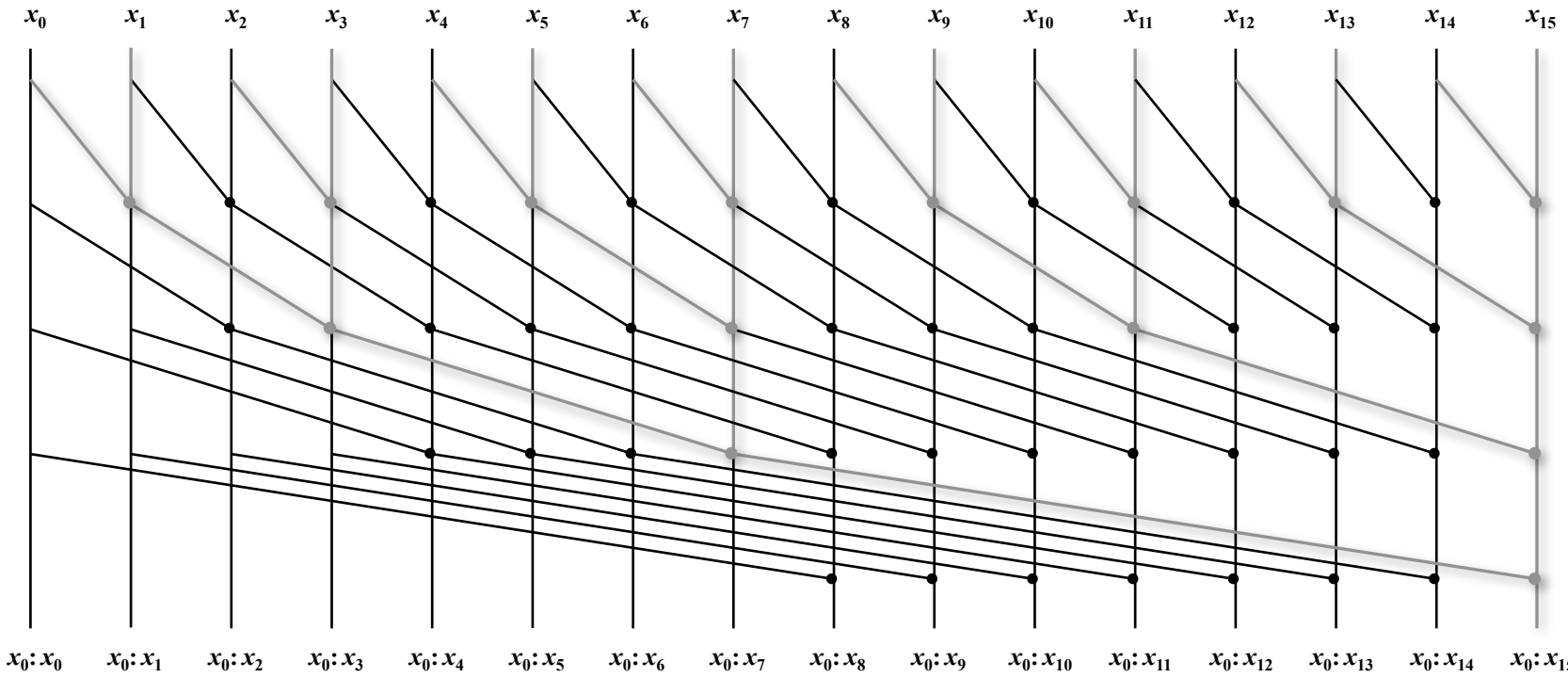




- Not step efficient (2 log n steps)
- Work efficient ($O(n)$ work)

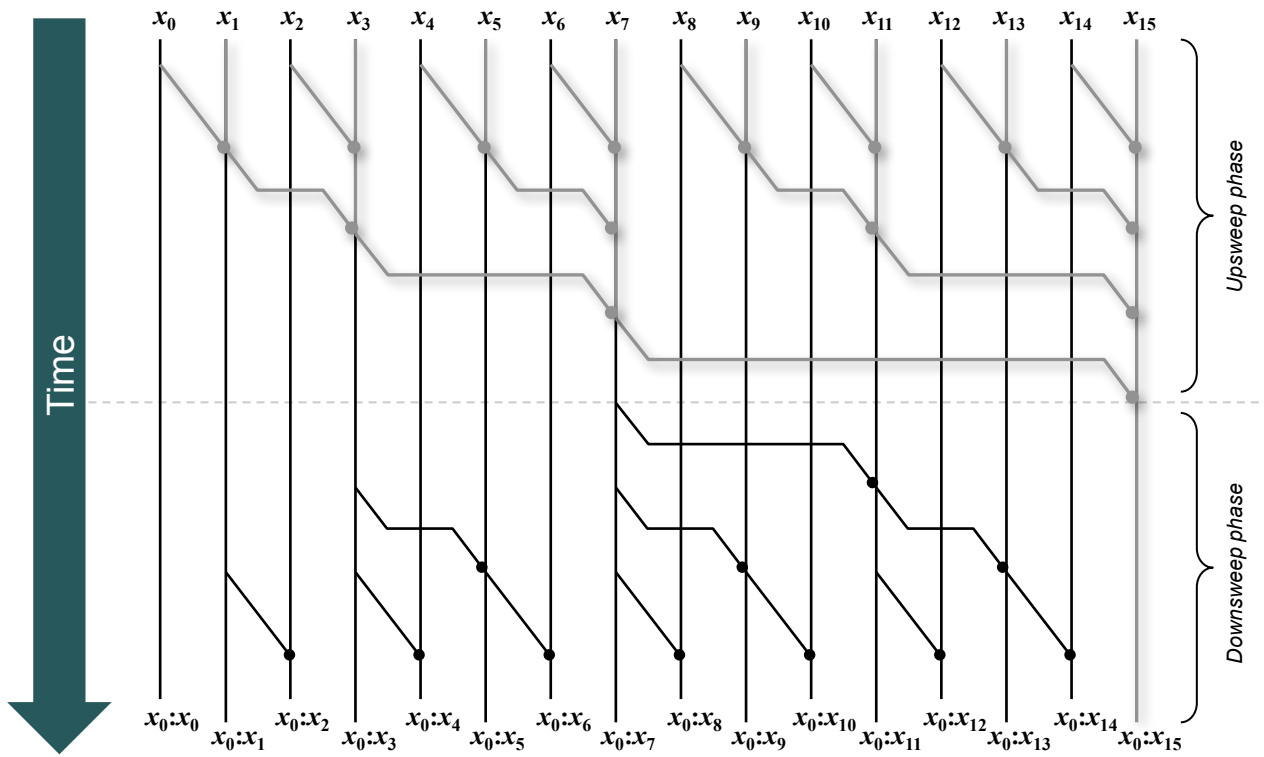
Kogge-Stone Scan

Circuit family

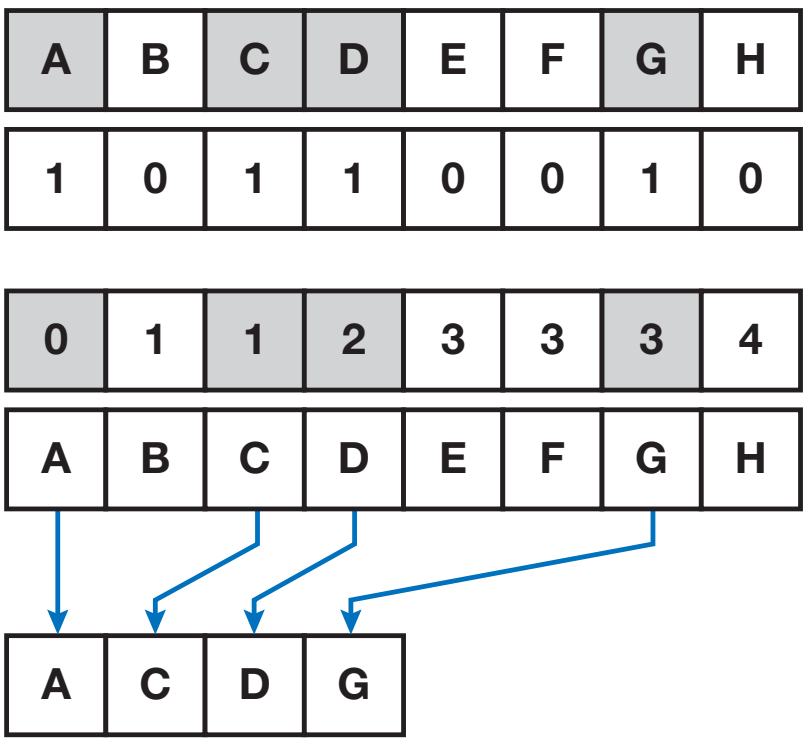


Brent Kung Scan

Circuit family



Application: Stream Compaction



Input: we want to preserve the gray elements

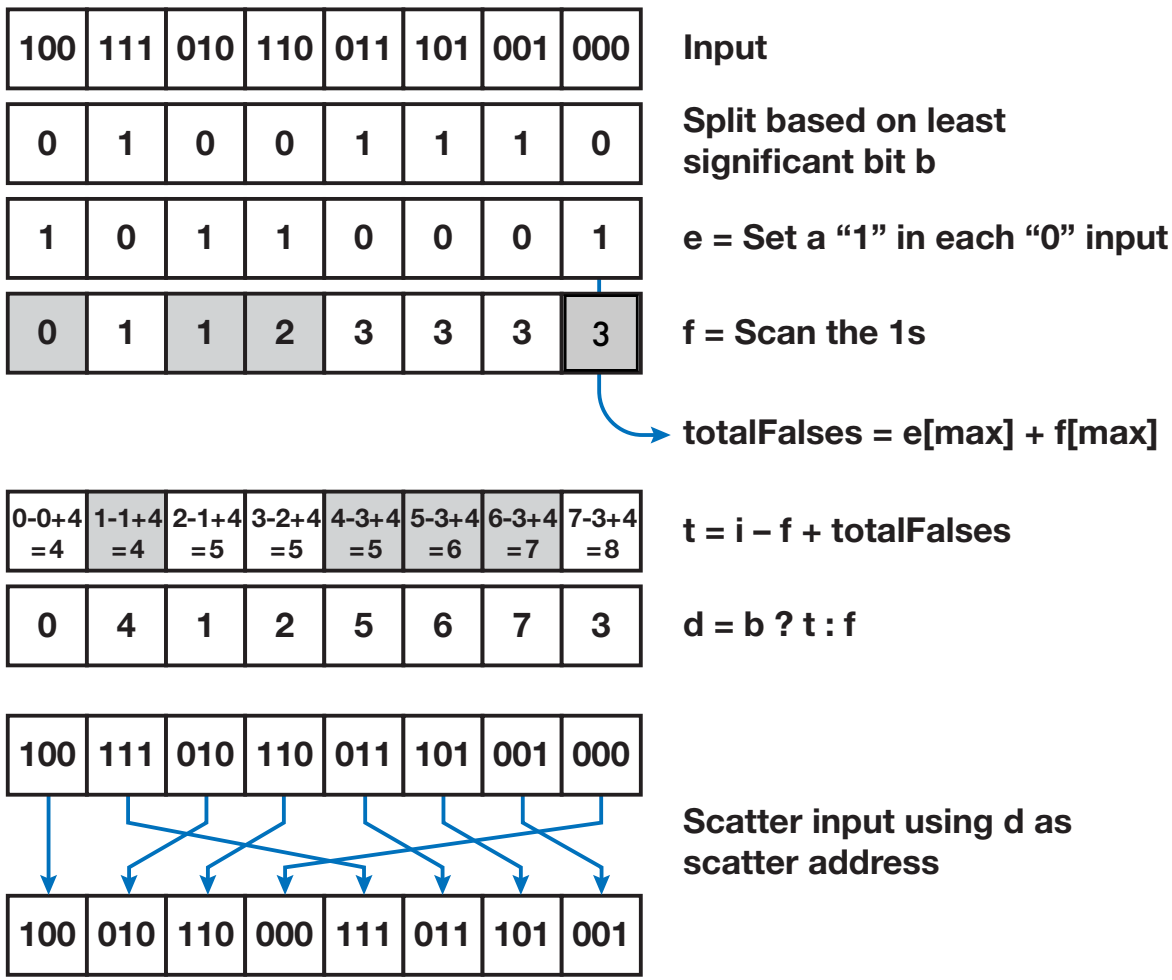
Set a "1" in each gray input

Scan

Scatter input to output, using scan result as scatter address

- 1M elements: ~0.6–1.3 ms
- 16M elements: ~8–20 ms
- Perf depends on # elements retained

Application: Radix Sort



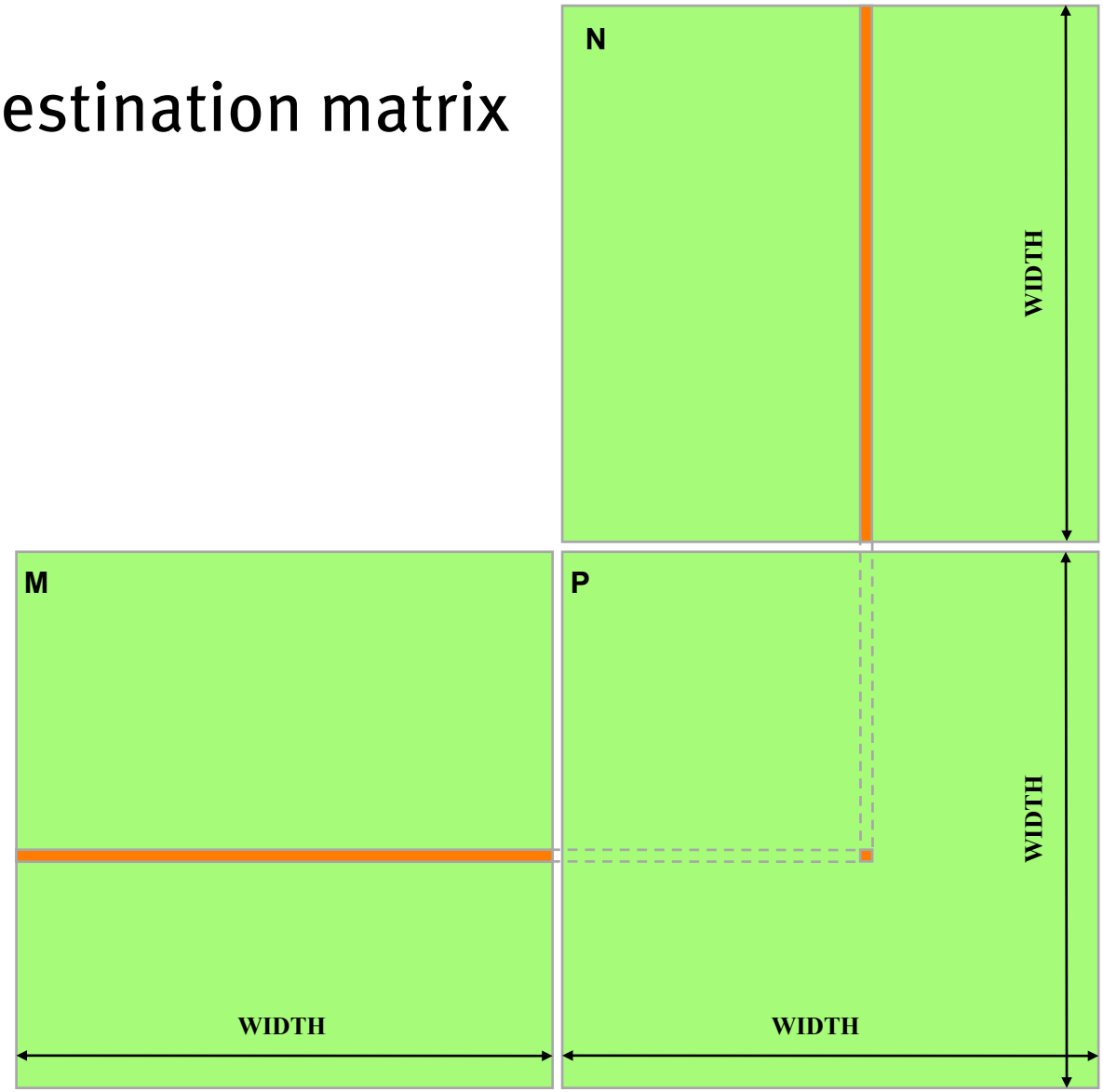
- Sort 16M 32-bit key-value pairs: ~120 ms
- Perform split operation on each bit using scan
- Can also sort each block and merge
  - Efficient merge on GPU an active area of research

GPU Design Principles

- Data layouts that:
  - Minimize memory traffic
  - Maximize coalesced memory access
- Algorithms that:
  - Exhibit data parallelism
  - Keep the hardware busy
  - Minimize divergence

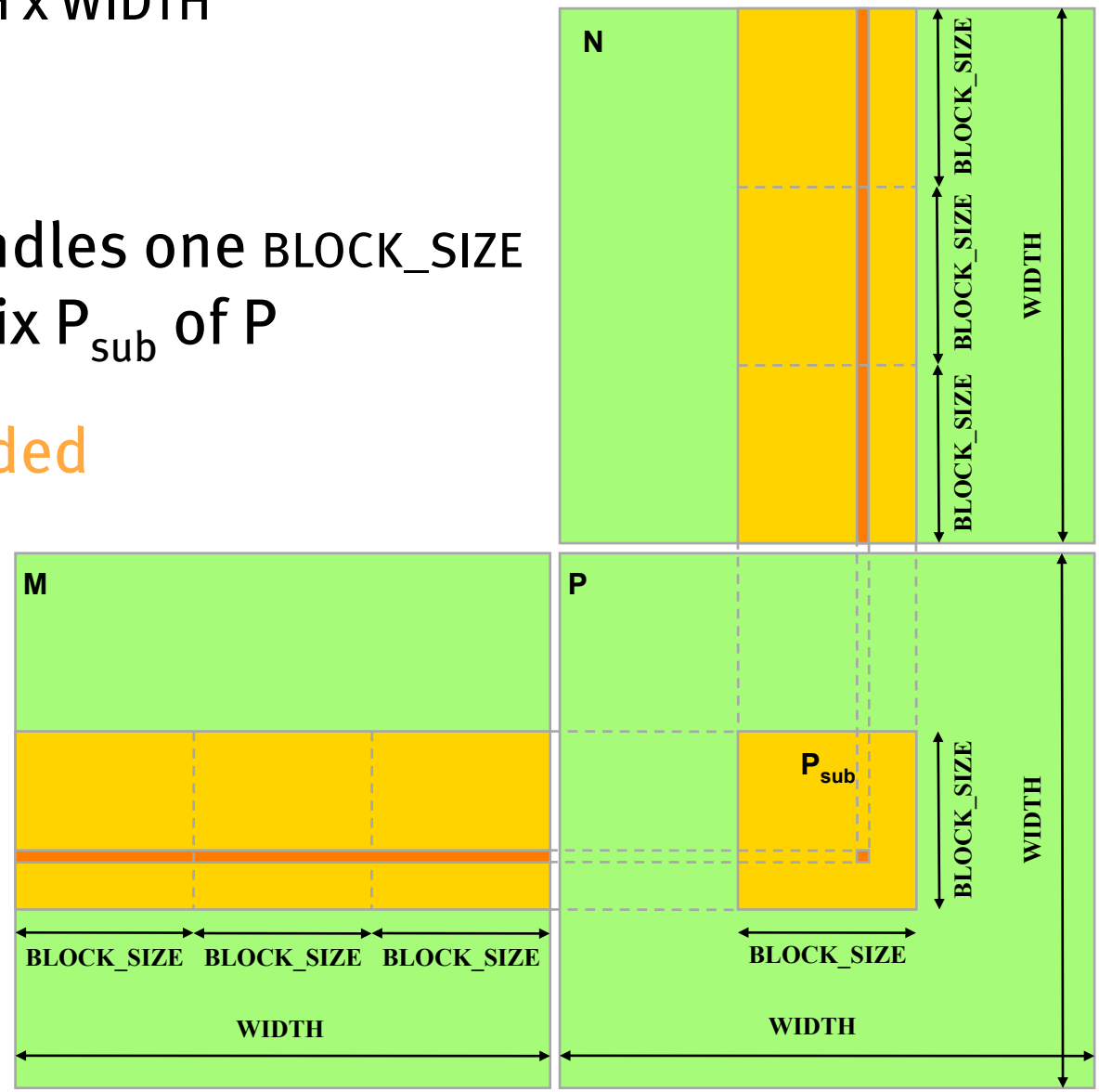
Dense Matrix Multiplication

- for all elements E in destination matrix P
  - $P_{r,c} = M_r \bullet N_c$



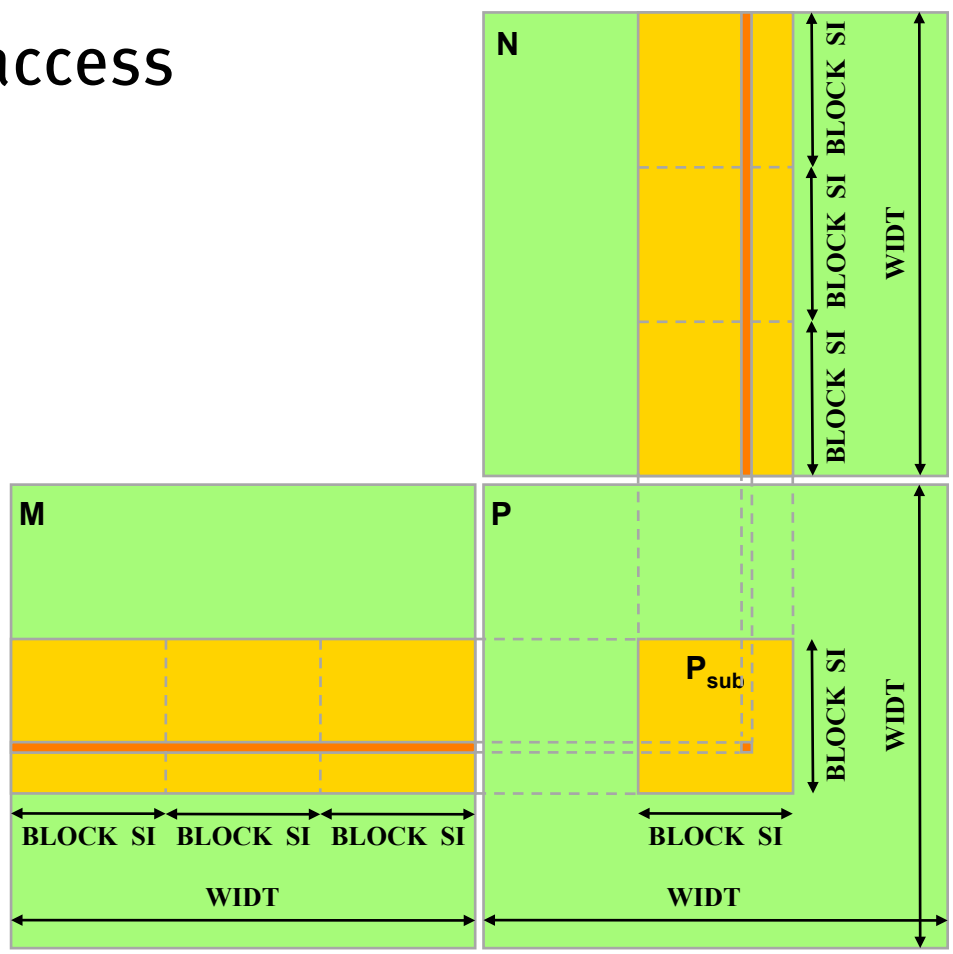
Dense Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- With blocking:
  - One thread block handles one BLOCK_SIZE x BLOCK_SIZE sub-matrix P_{sub} of P
  - M and N are only loaded WIDTH / BLOCK_SIZE times from global memory
- Great saving of memory bandwidth!



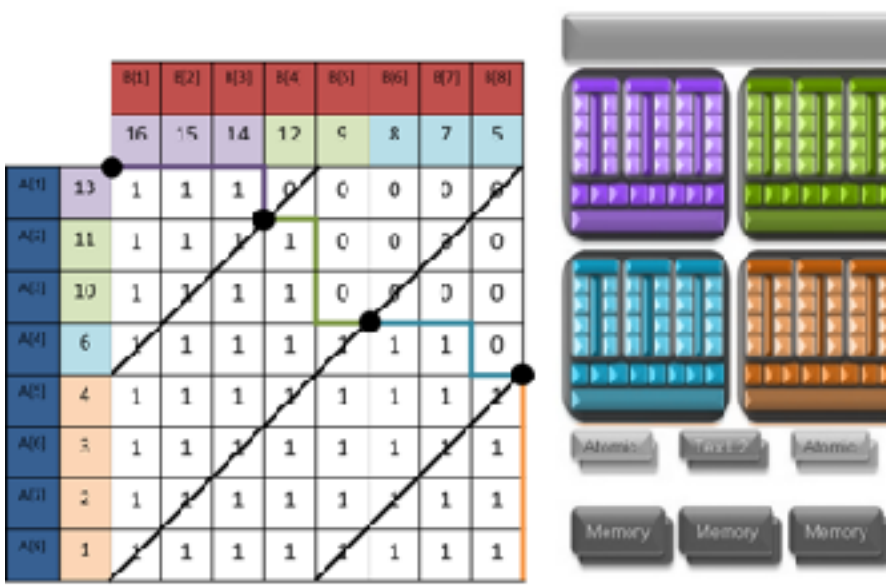
Dense Matrix Multiplication

- Data layouts that:
  - Minimize memory traffic
  - Maximize coalesced memory access
- Algorithms that:
  - Exhibit data parallelism
  - Keep the hardware busy
  - Minimize divergence



Merge-Path and Its Impact on Irregular Algorithms

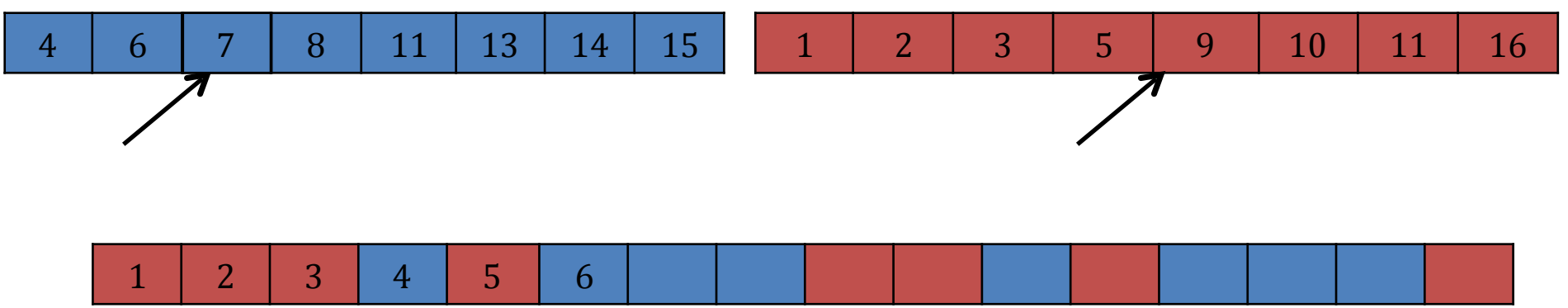
Oded Green



Lets start off by defining the “Merge” operation

- **Input:** Two sorted arrays A,B
- **Output:** Sorted array C
- |C| = |A| + |B|
- **Time:** O(n) = O(|C|)
- Simple

$$
\begin{aligned}
&\textit{if } (A[a_i] < B[b_i]) \textit{ then}\\
&\quad C[c_i++] \leftarrow A[a_i++]\\
&\textit{else}\\
&\quad C[c_i++] \leftarrow B[b_i++]
\end{aligned}
$$

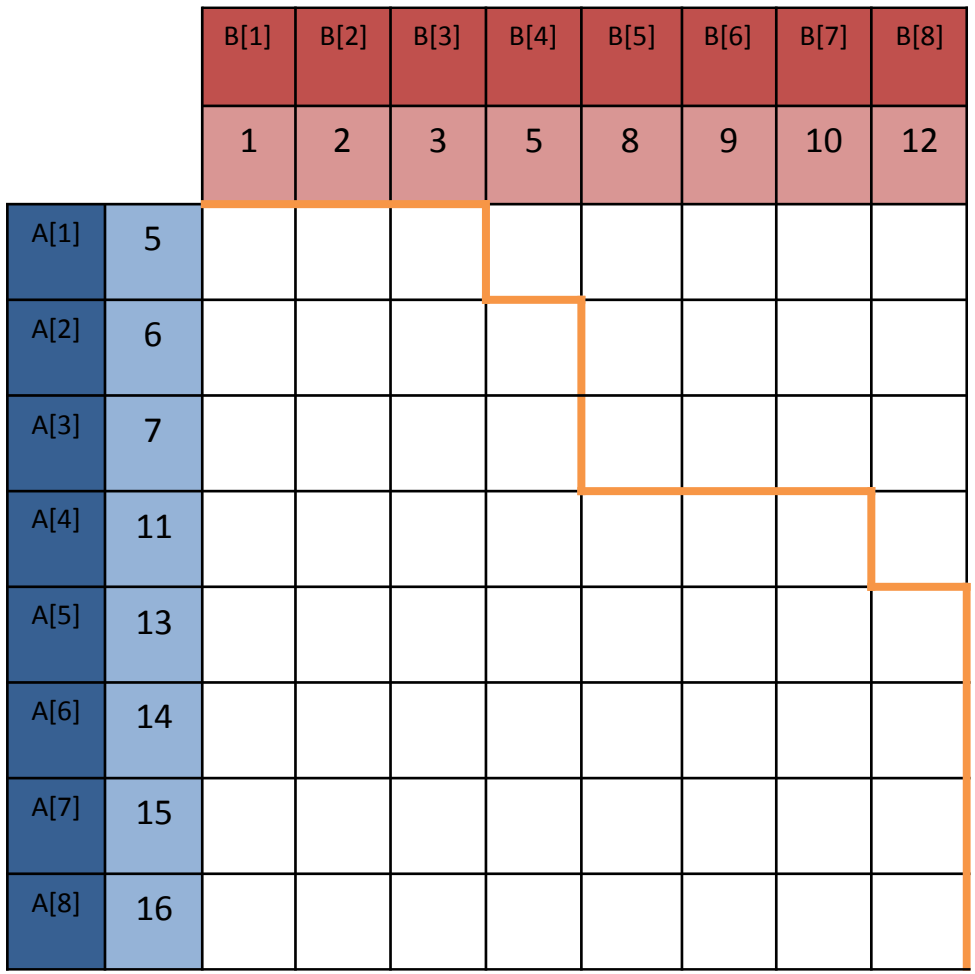


Parallel Merge Challenges

- Partitioning / Load balancing
  - Must be computationally cheap (less than O(n))
  - Must be parallel (high utilization)
- Low synchronization/communication
- Scalable
- Simple (preferably)

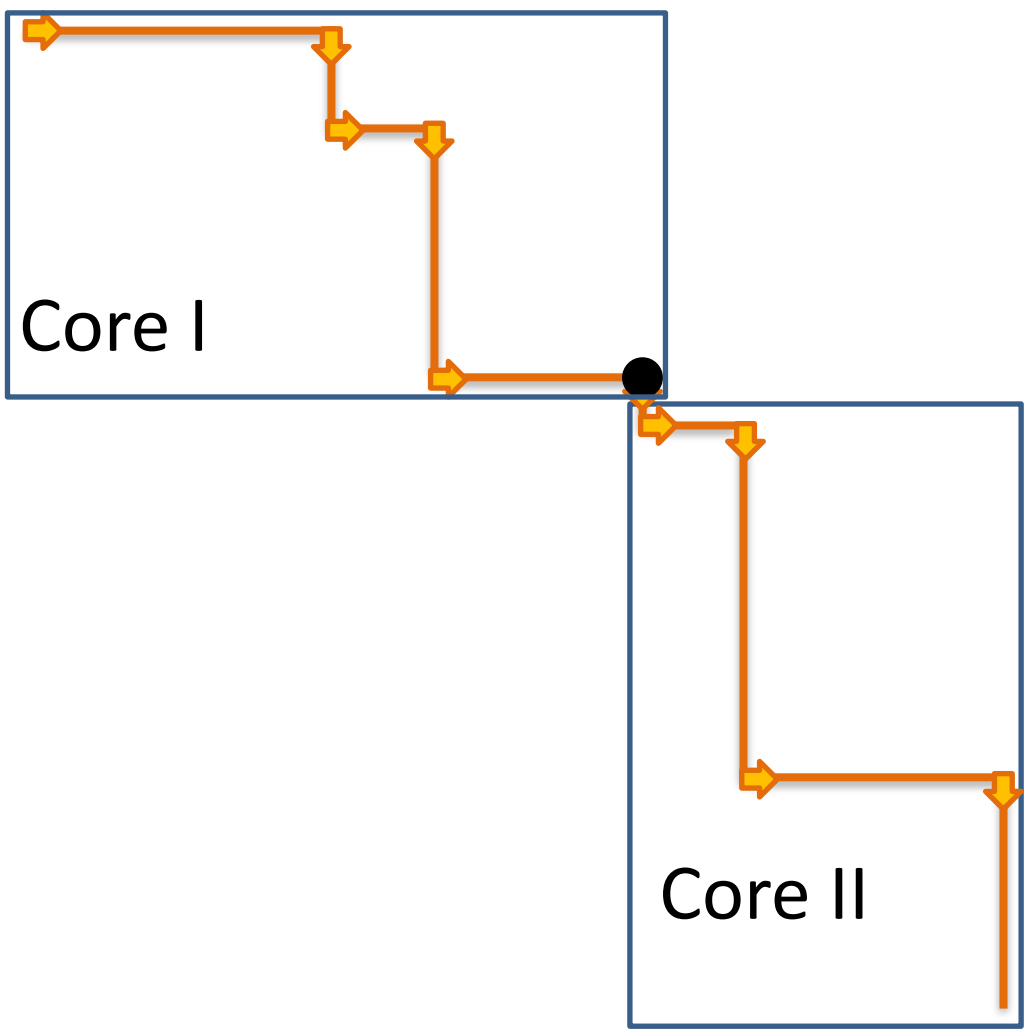
Merge Path

- if (A[i]>B[j])
  - Select A[i]
  - Next i
  - **Move down**
- else
  - Select B[j]
  - Next j
  - **Move right**



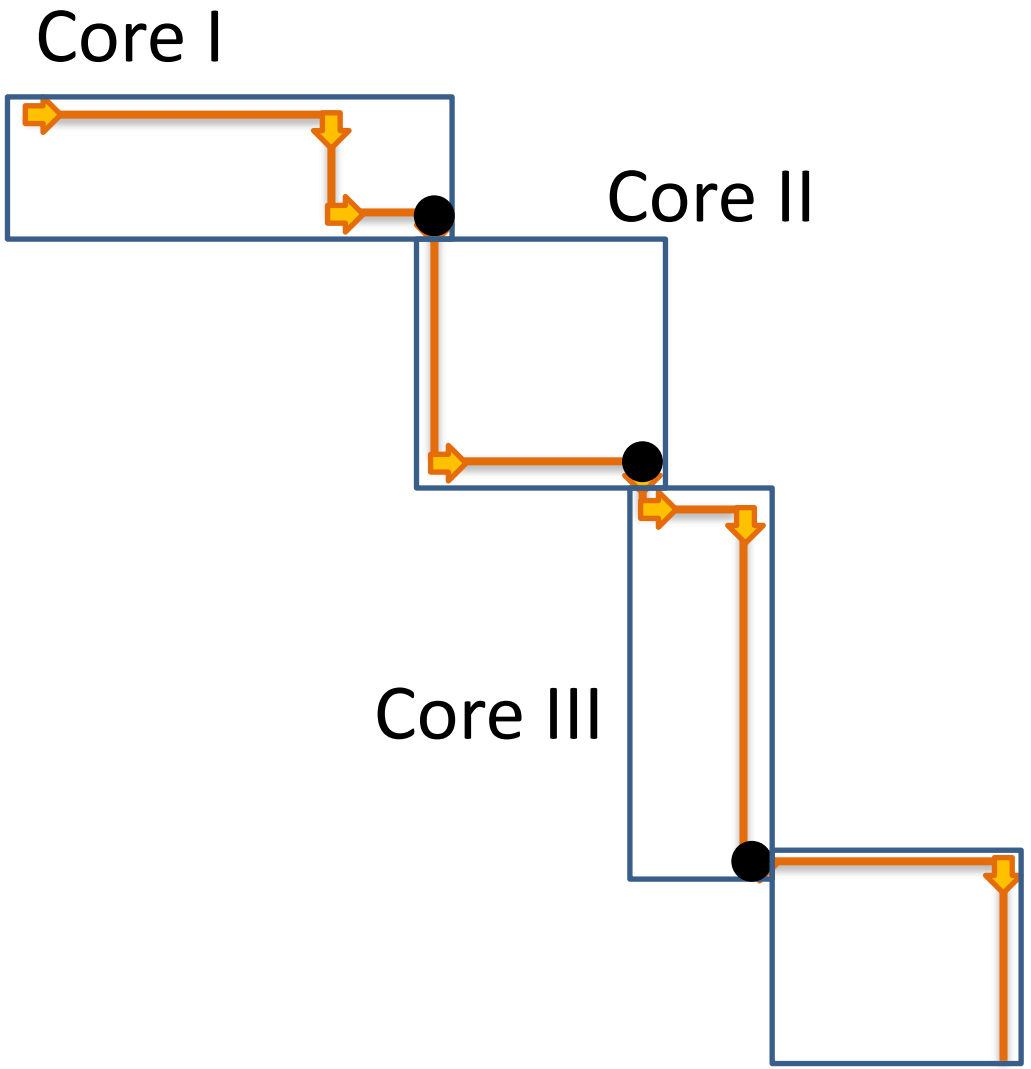
[*Odeh, Green, Birk*; 2012; *MTAAP*]
[*Green*; 2012; *ICS*]

Intuition – 2 Cores



- Each core will merge a sub-path
- For perfect load balancing, divide path into equal length sub-paths

Intuition – 4 Cores

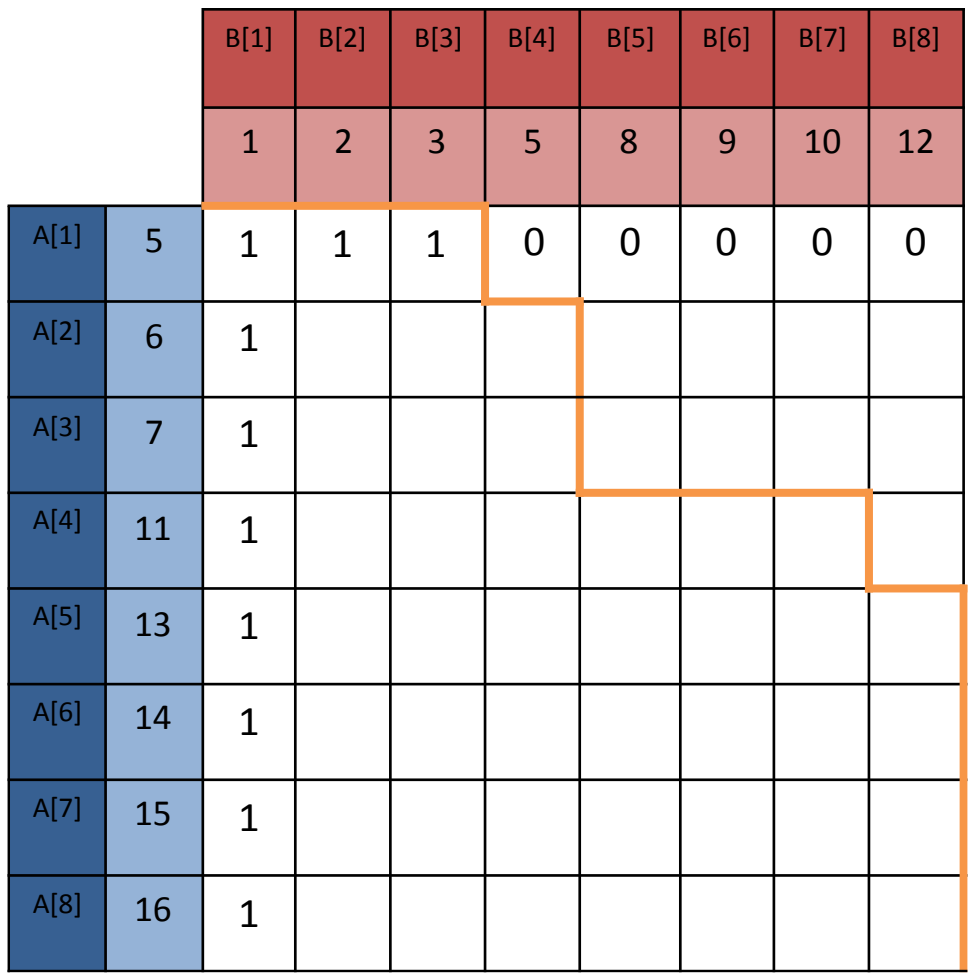


- Each core will merge a sub-path
- For perfect load balancing, divide path into equal length sub-paths

Merge Matrix

- Size |A|x|B|

- M[i,j]=1 if A[i]<B[j]
- M[i,j]=0 if A[i]≥B[j]



		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		1	2	3	5	8	9	10	12
A[1]	5	1	1	1	0	0	0	0	0
A[2]	6	1							
A[3]	7	1							
A[4]	11	1							
A[5]	13	1							
A[6]	14	1							
A[7]	15	1							
A[8]	16	1							

Merge Matrix

- O(|A||B|) space and time to compute
- Matrix is conceptual
- Values implicitly computed

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		1	2	3	5	8	9	10	12
A[1]	5	1	1	1	0	0	0	0	0
A[2]	6	1	1	1	1	0	0	0	0
A[3]	7	1	1	1	1	0	0	0	0
A[4]	11	1	1	1	1	1	1	1	0
A[5]	13	1	1	1	1	1	1	1	1
A[6]	14	1	1	1	1	1	1	1	1
A[7]	15	1	1	1	1	1	1	1	1
A[8]	16	1	1	1	1	1	1	1	1

Merge Matrix

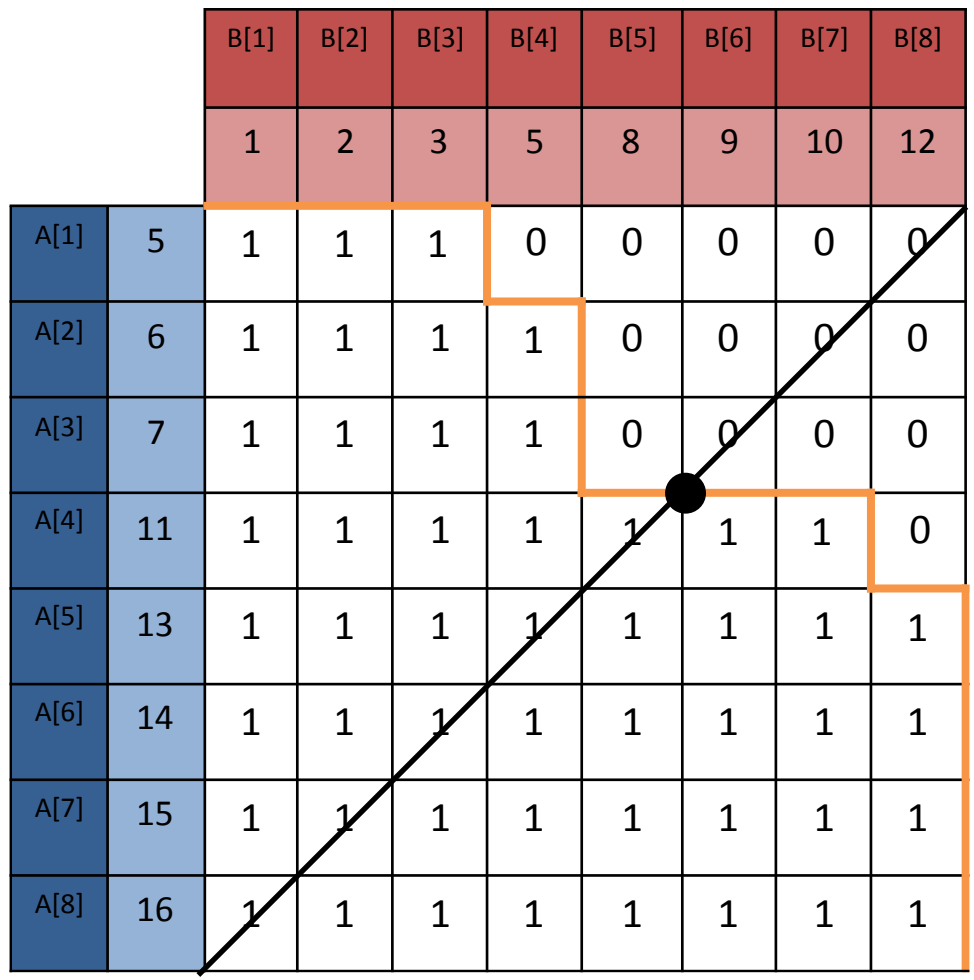
- Path goes between "1"s and "0"s.

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		1	2	3	5	8	9	10	12
A[1]	5	1	1	1	0	0	0	0	0
A[2]	6	1	1	1	1	0	0	0	0
A[3]	7	1	1	1	1	0	0	0	0
A[4]	11	1	1	1	1	1	1	1	0
A[5]	13	1	1	1	1	1	1	1	1
A[6]	14	1	1	1	1	1	1	1	1
A[7]	15	1	1	1	1	1	1	1	1
A[8]	16	1	1	1	1	1	1	1	1

Finding Partitioning Points

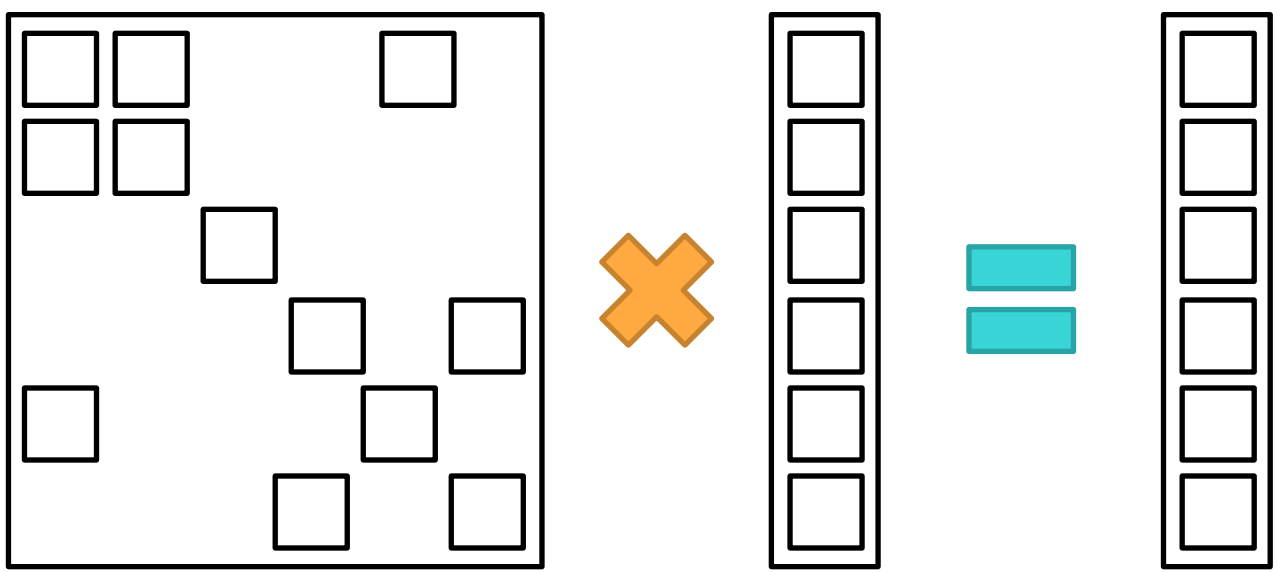
- Path
  - start = top-left
  - stop = bottom-right
- Cross Diagonals
  - start = top-right
  - stop = bottom-left

=>Intersection



Sparse Matrix-Vector Multiply: What's Hard?

- Dense approach is wasteful
- Unclear how to map work to parallel processors
- Irregular data access



Sparse Matrix Formats

(DIA) Diagonal

(ELL) ELLPACK

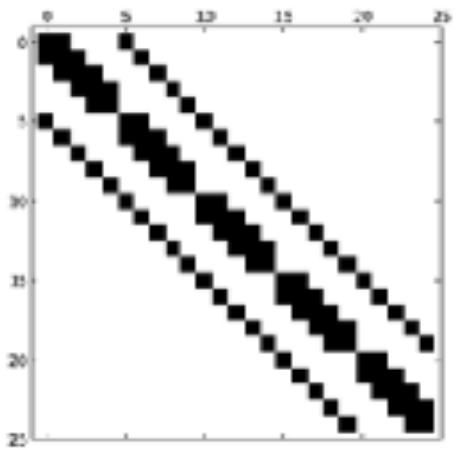
(CSR) Compressed Row

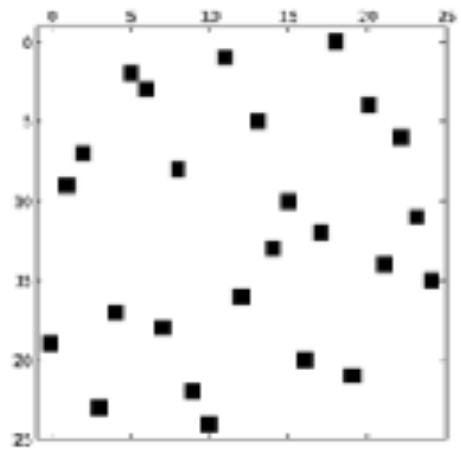
(HYB) Hybrid

(COO) Coordinate

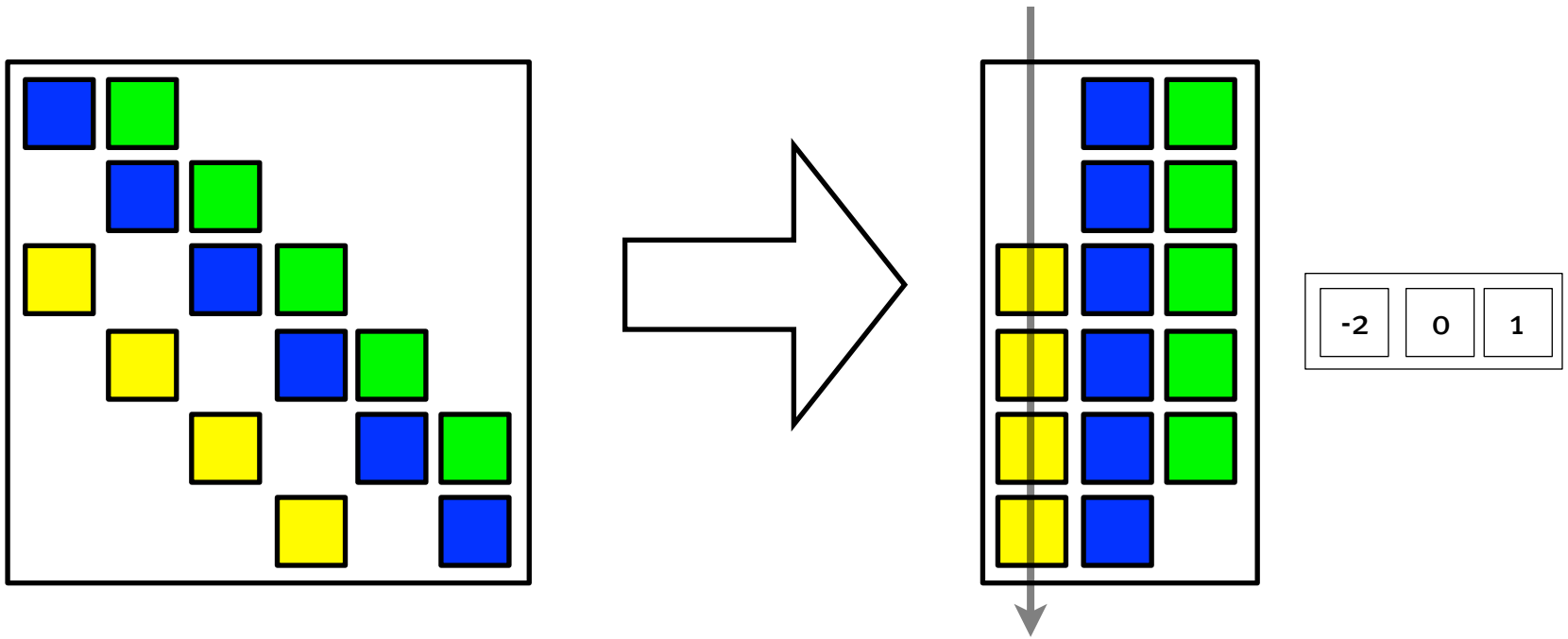
Structured

Unstructured



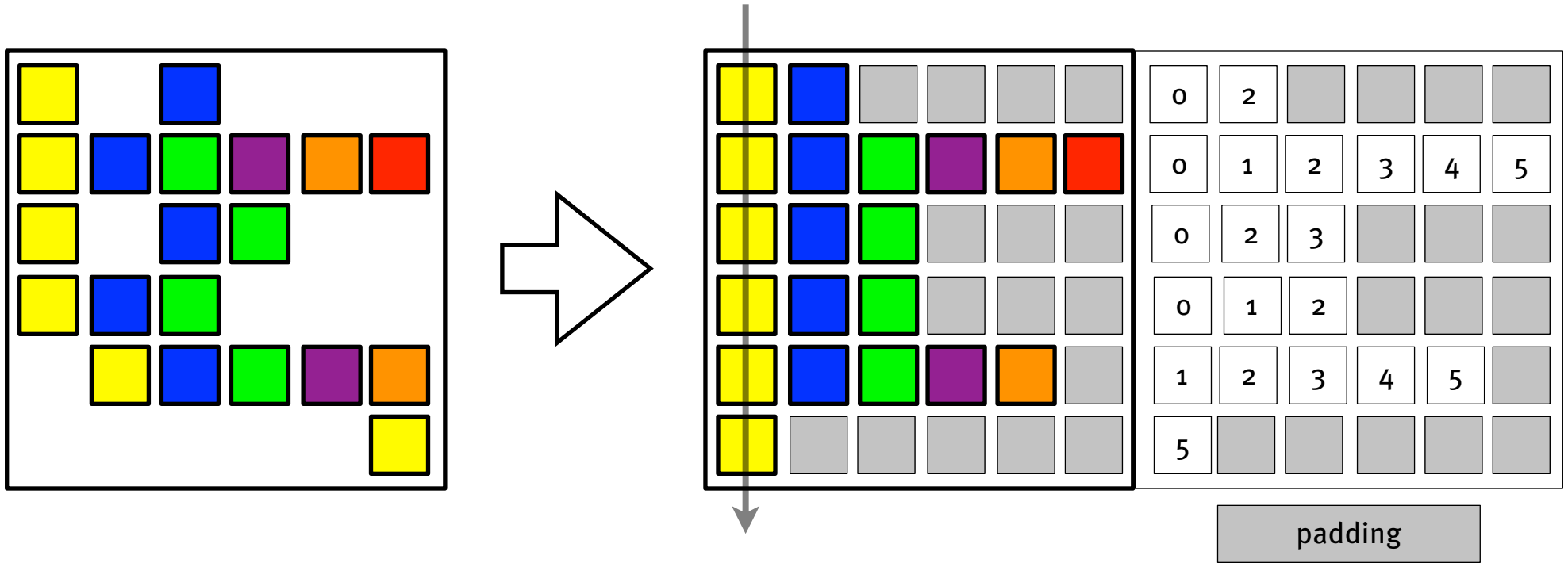


Diagonal Matrices



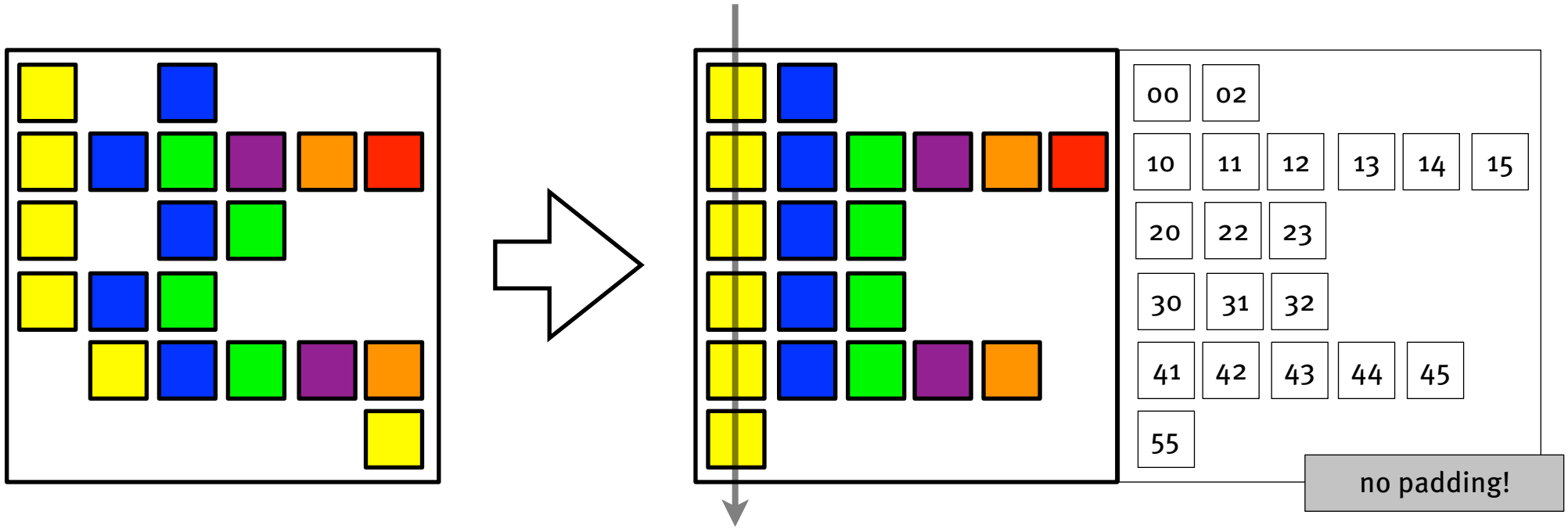
- Diagonals should be mostly populated
- Map one thread per row
  - Good parallel efficiency
  - Good memory behavior [column-major storage]

Irregular Matrices: ELL



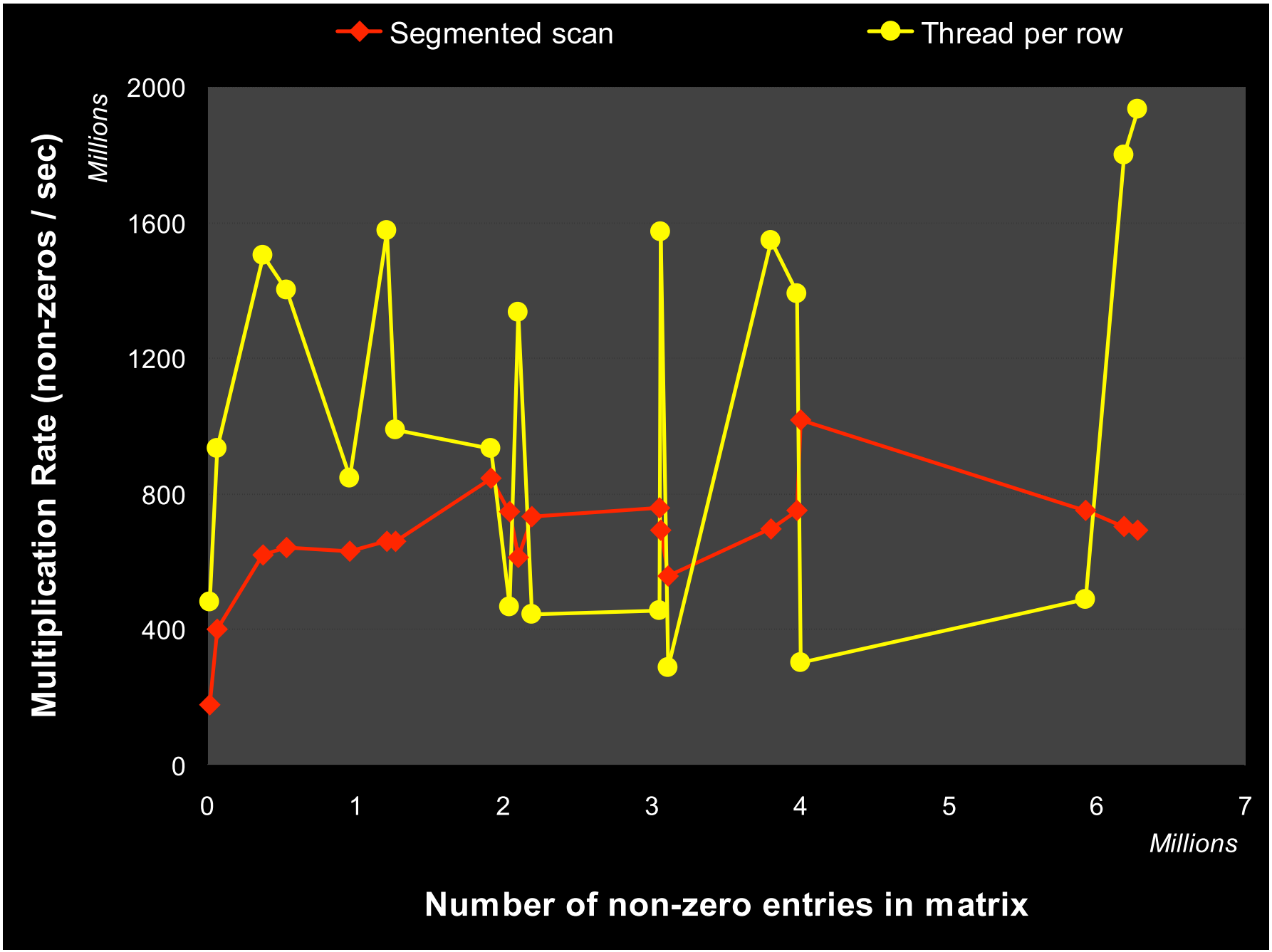
- Assign one thread per row again
- But now:
  - Load imbalance hurts parallel efficiency

Irregular Matrices: COO

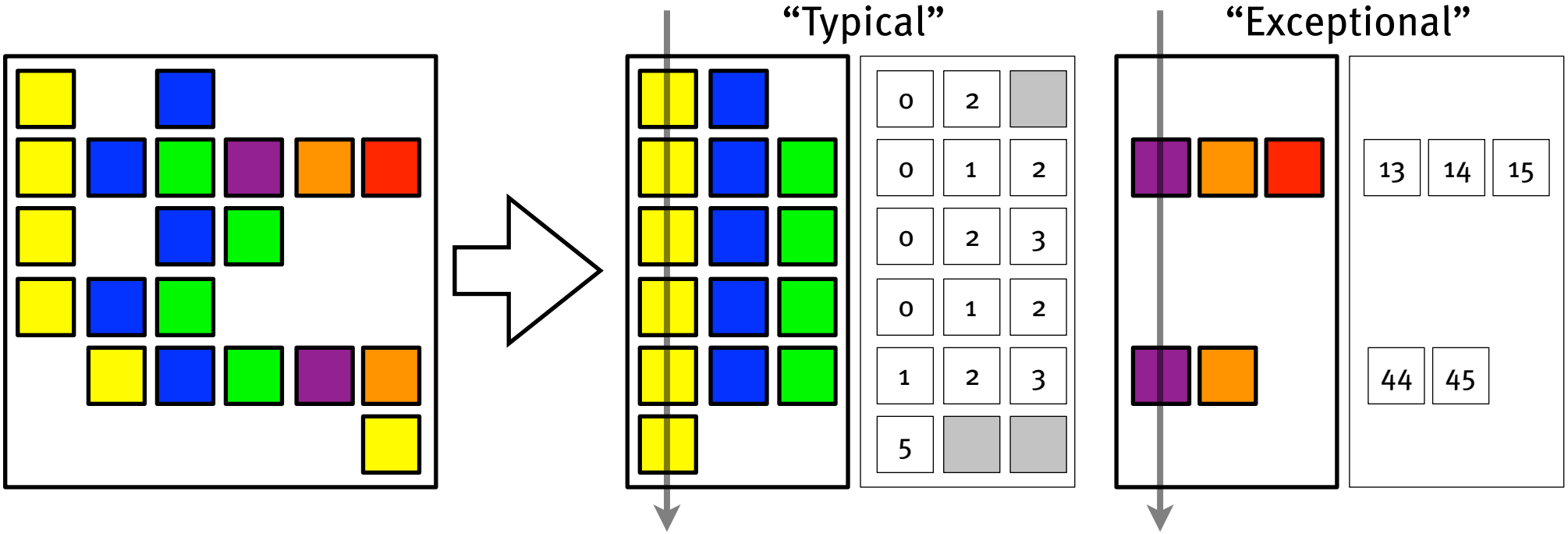


- General format; insensitive to sparsity pattern, but ~3x slower than ELL
- Assign one thread per element, combine results from all elements in a row to get output element
  - Req segmented reduction, communication btwn threads

Thread-per-{element,row}



Irregular Matrices: HYB

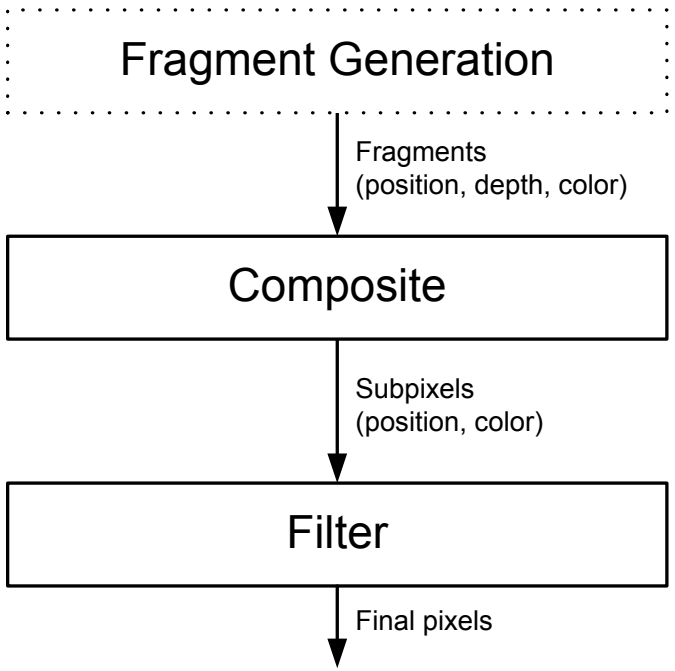


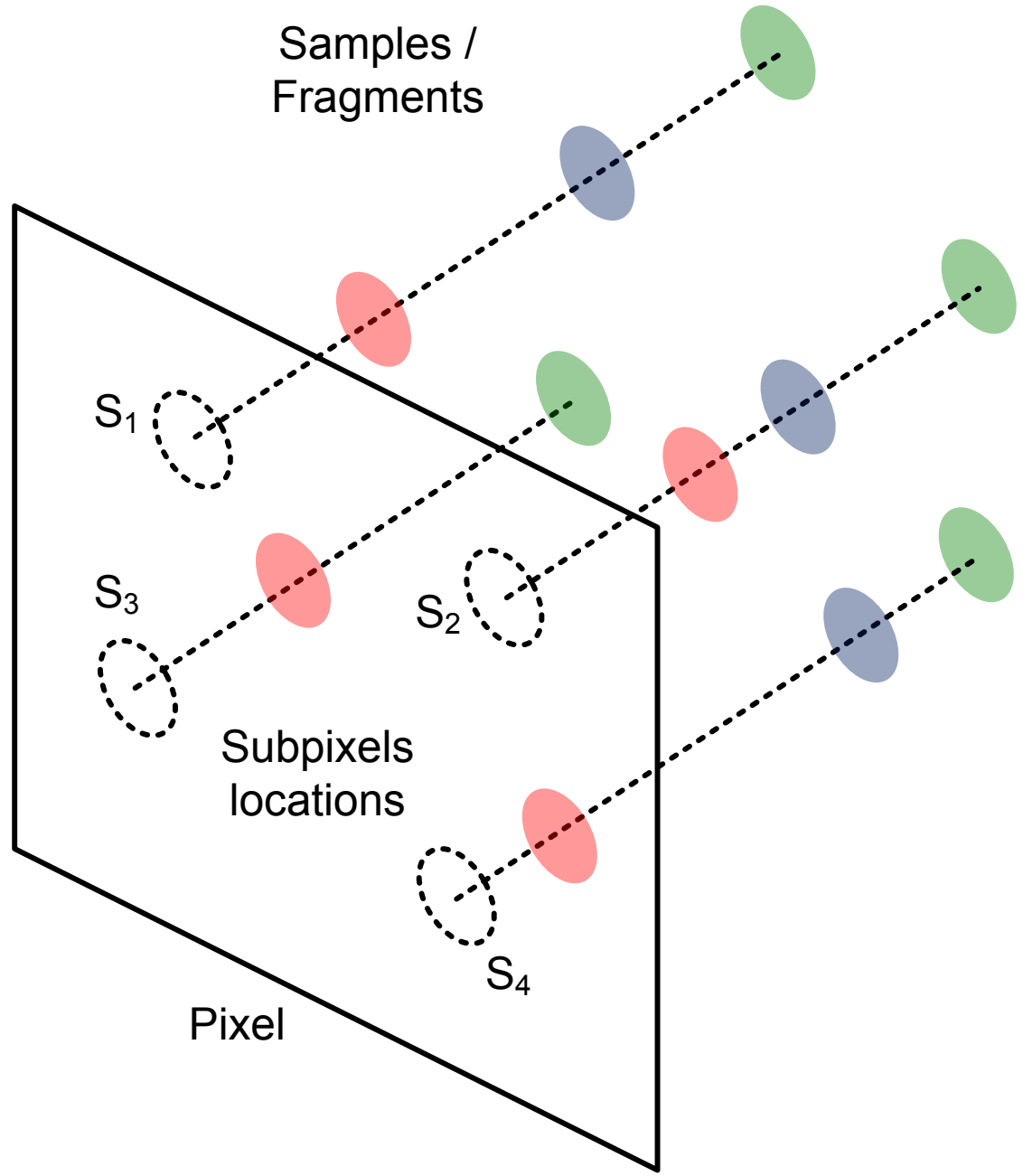
- Combine regularity of ELL + flexibility of COO

SpMV: Summary

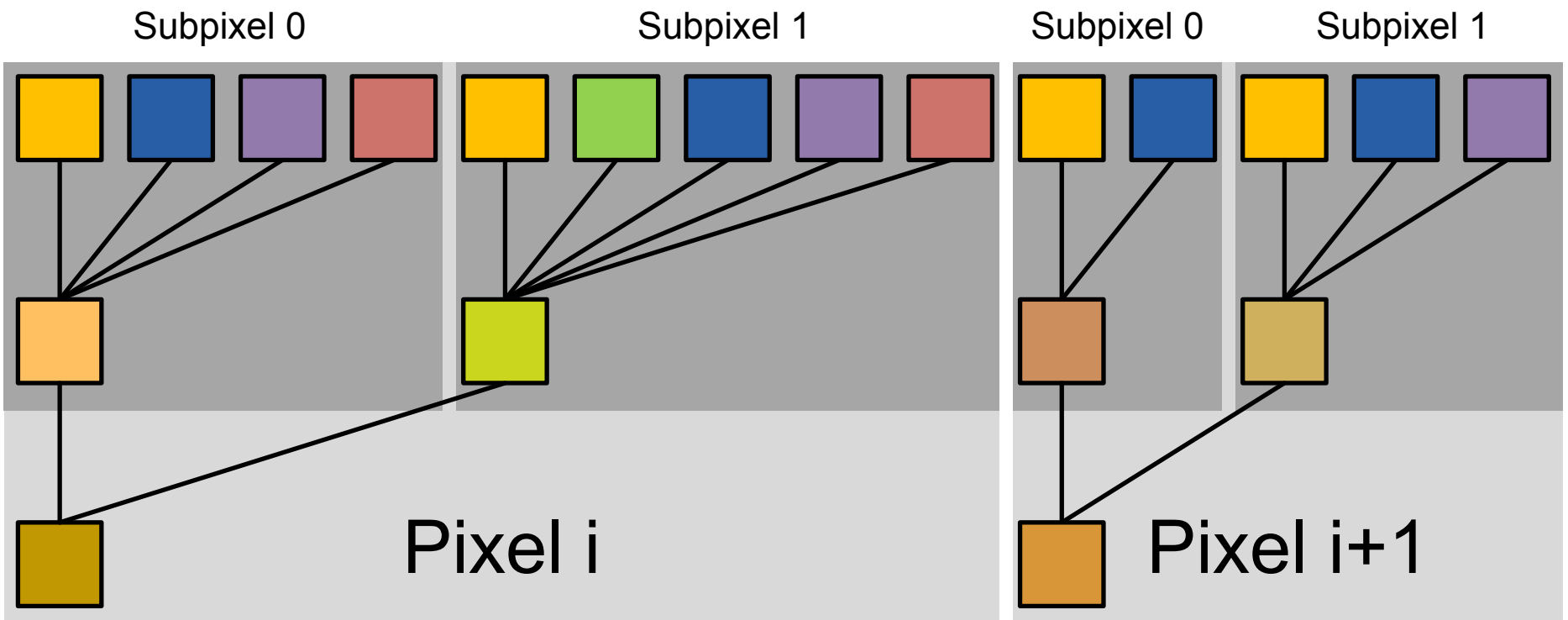
- Ample parallelism for large matrices
  - Structured matrices (dense, diagonal): straightforward
- Take-home message: Use data structure appropriate to your matrix
- Sparse matrices: Issue: Parallel efficiency
  - ELL format / one thread per row is efficient
- Sparse matrices: Issue: Load imbalance
  - COO format / one thread per element is insensitive to matrix structure
- Conclusion: Hybrid structure gives best of both worlds
  - Insight: Irregularity is manageable if you regularize the common case

Composition

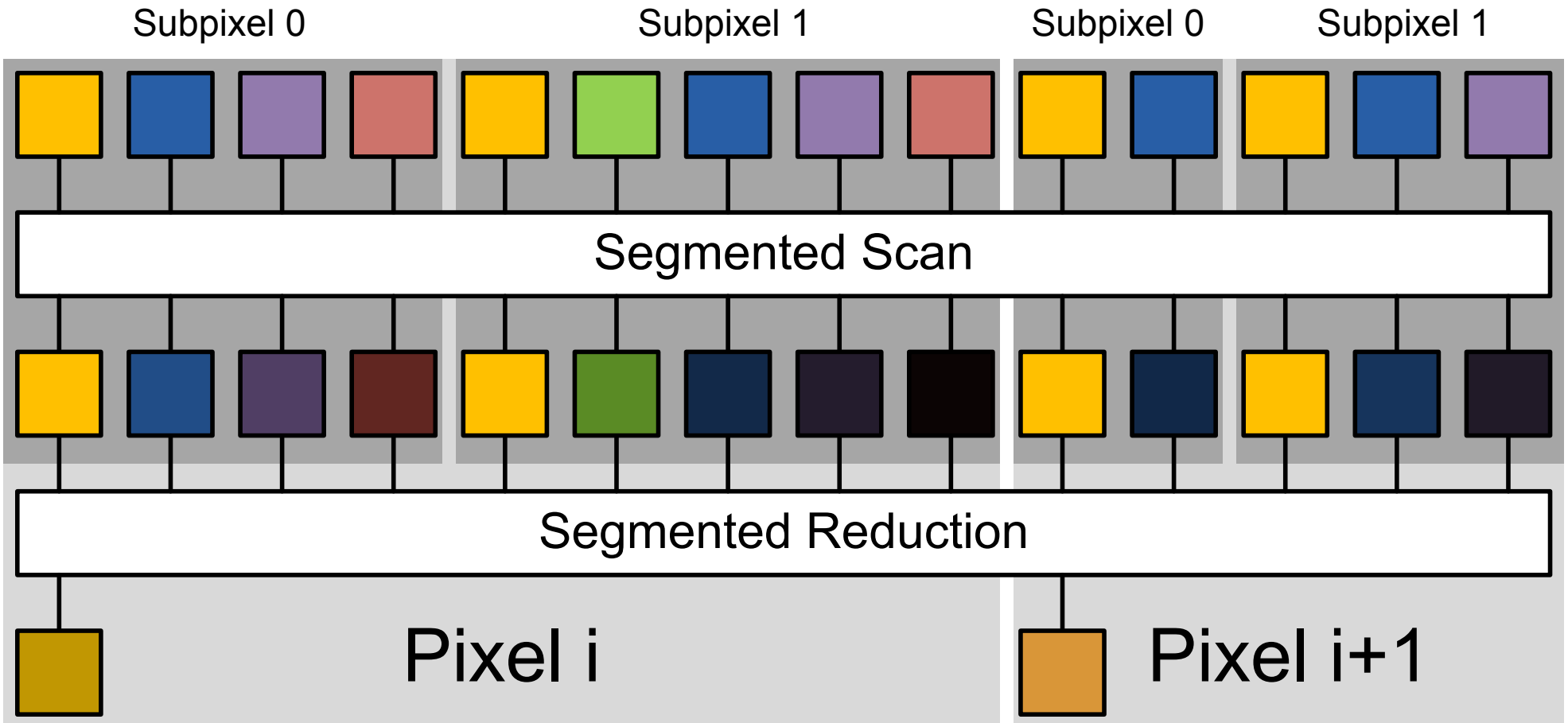


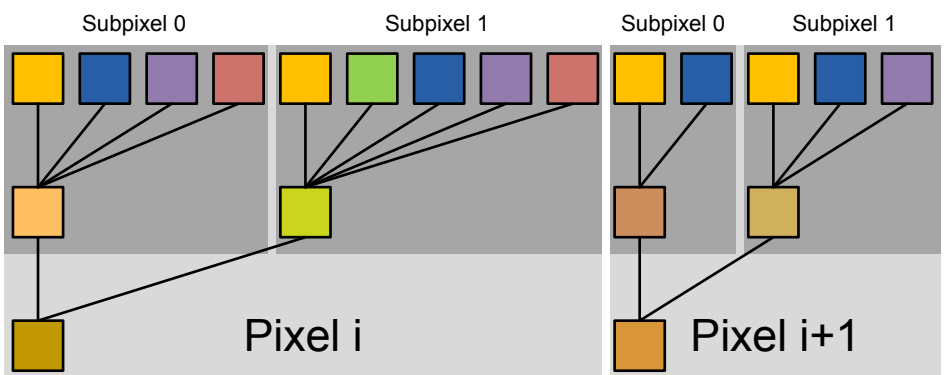


Pixel-Parallel Composition

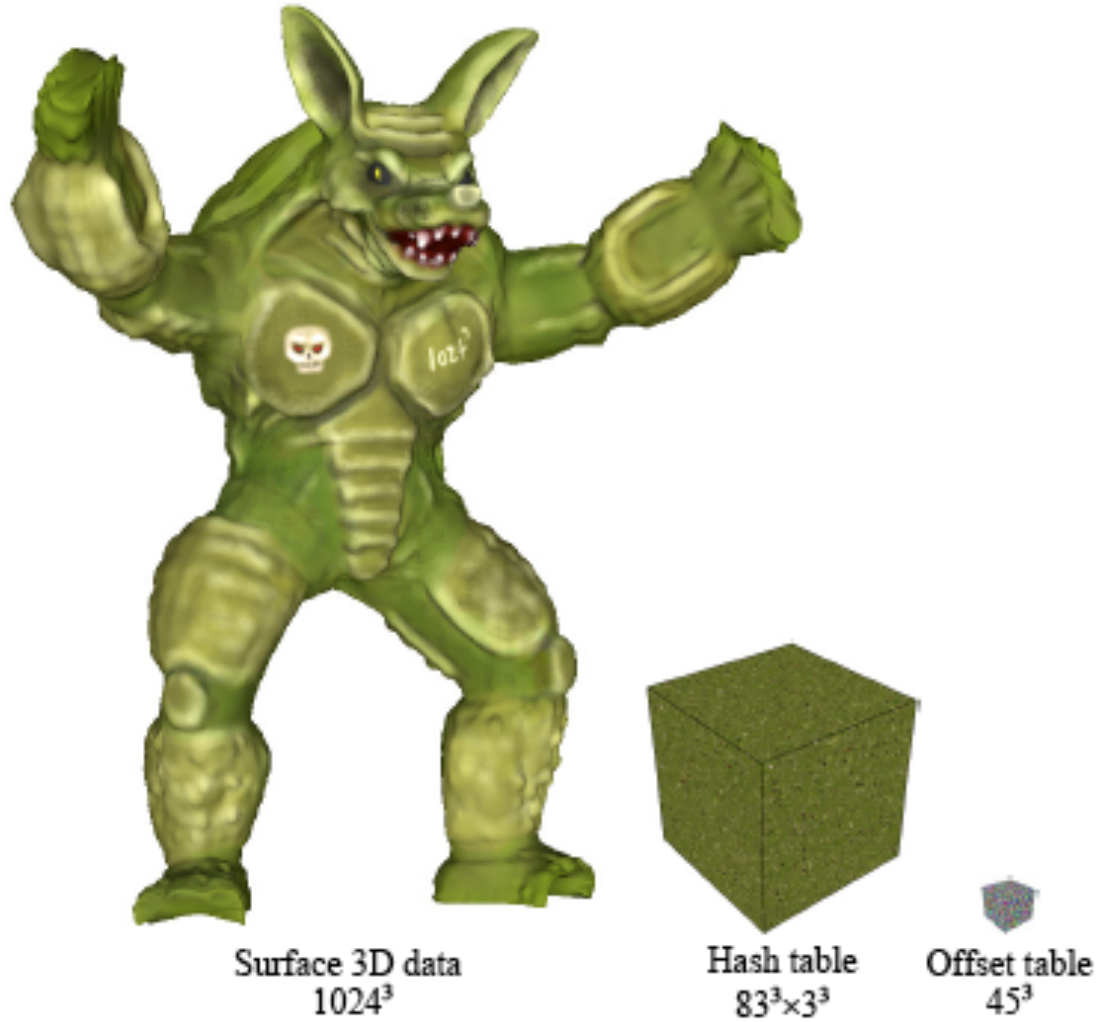


Sample-Parallel Composition



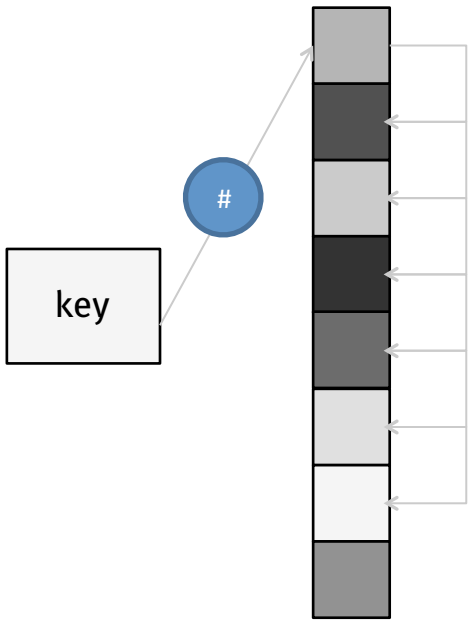


Hash Tables & Sparsity

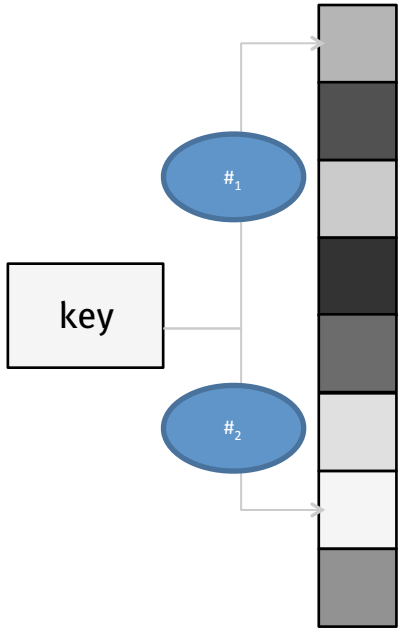


- Lefebvre and Hoppe, Siggraph 2006

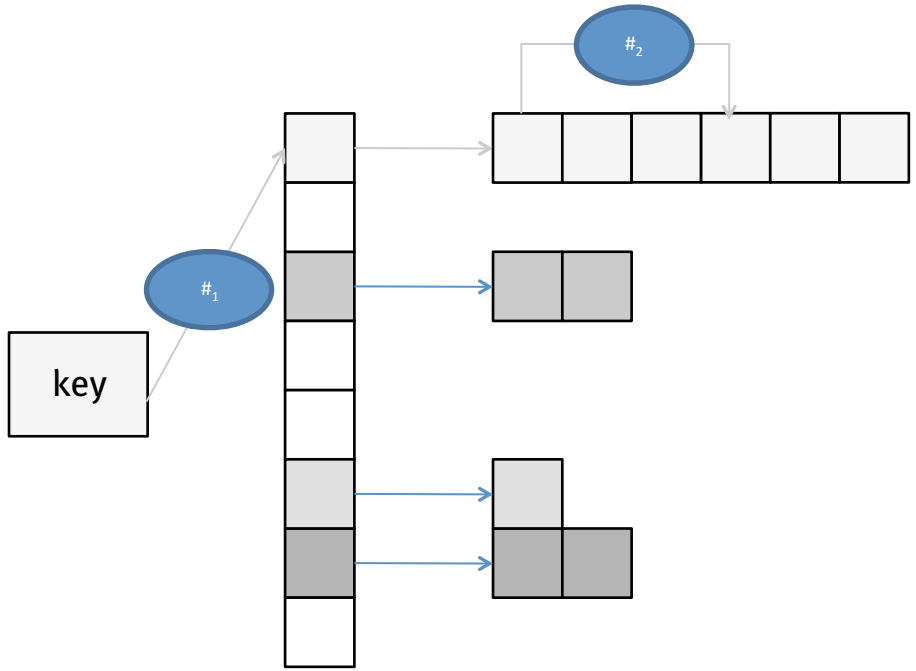
Scalar Hashing



Linear Probing

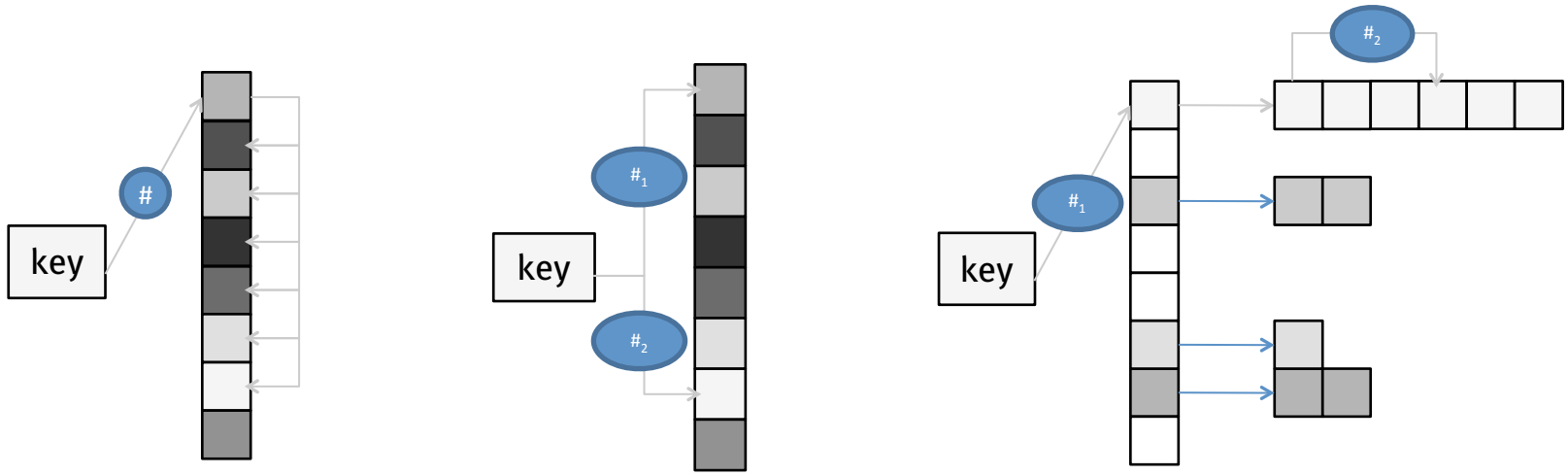


Double Probing



Chaining

Scalar Hashing: Parallel Problems



- Construction and Lookup
  - Variable time/work per entry
- Construction
  - Synchronization / shared access to data structure

Parallel Hashing: The Problem

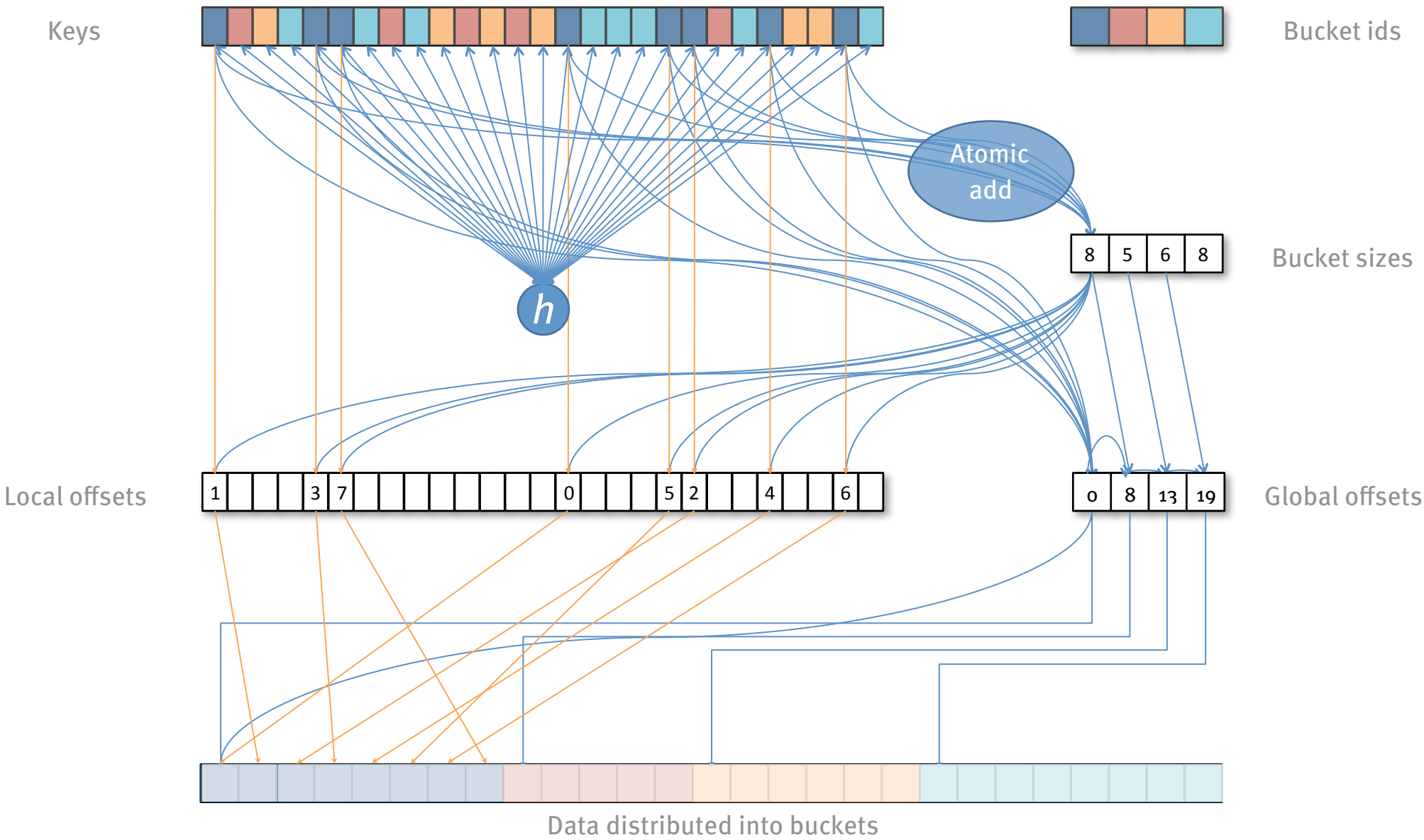
- Hash tables are good for sparse data.
- Input: Set of key-value pairs to place in the hash table
- Output: Data structure that allows:
  - Determining if key has been placed in hash table
  - Given the key, fetching its value
- Could also:
  - Sort key-value pairs by key (construction)
  - Binary-search sorted list (lookup)
- Recalculate at every change

Parallel Hashing: What We Want

- Fast construction time
- Fast access time
  - $O(1)$ for any element, $O(n)$ for n elements in parallel
- Reasonable memory usage

- Algorithms and data structures may sit at different places in this space
  - Perfect spatial hashing has good lookup times and reasonable memory usage but is very slow to construct

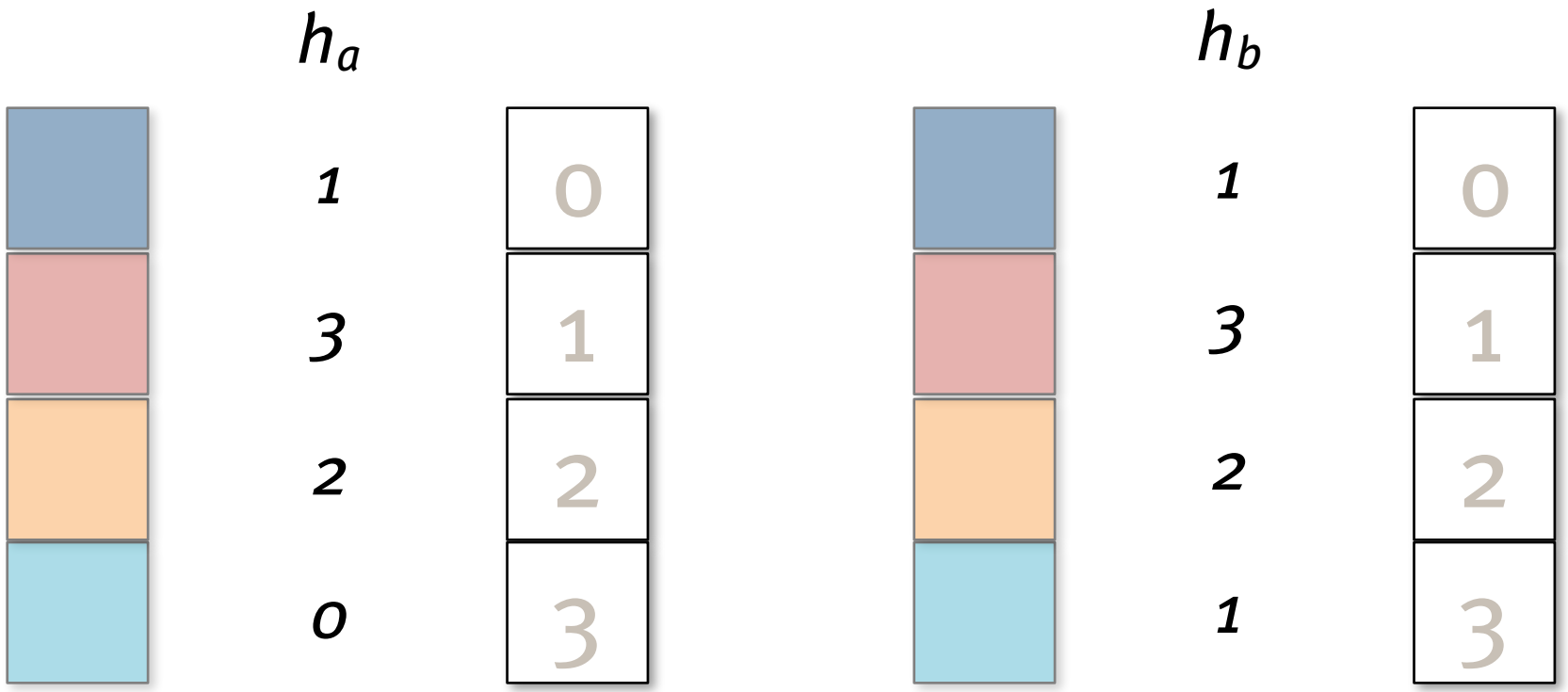
Level 1: Distribute into buckets



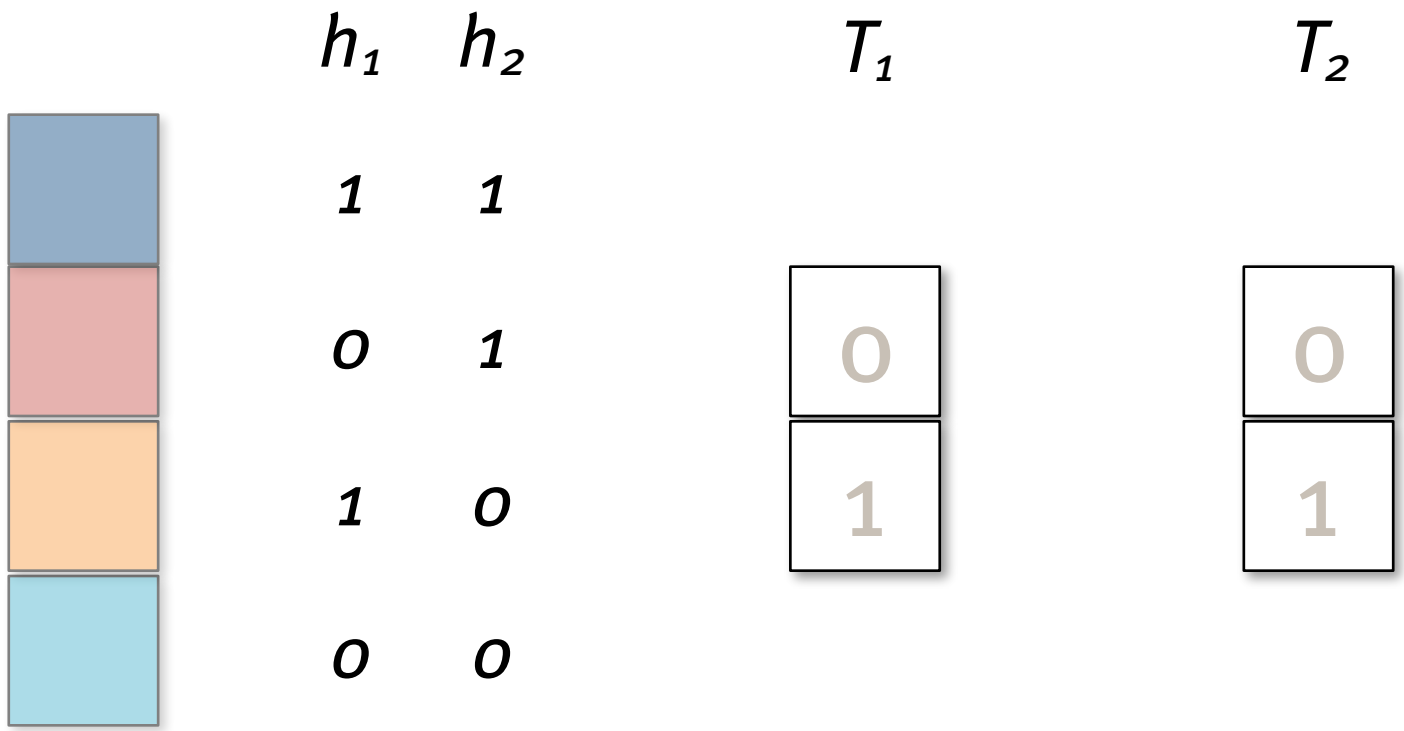
Parallel Hashing: Level 1

- Good for a coarse categorization
  - Possible performance issue: atomics
- Bad for a fine categorization
  - Space requirements for n elements to (probabilistically) guarantee no collisions are $O(n^2)$

Hashing in Parallel



Cuckoo Hashing Construction

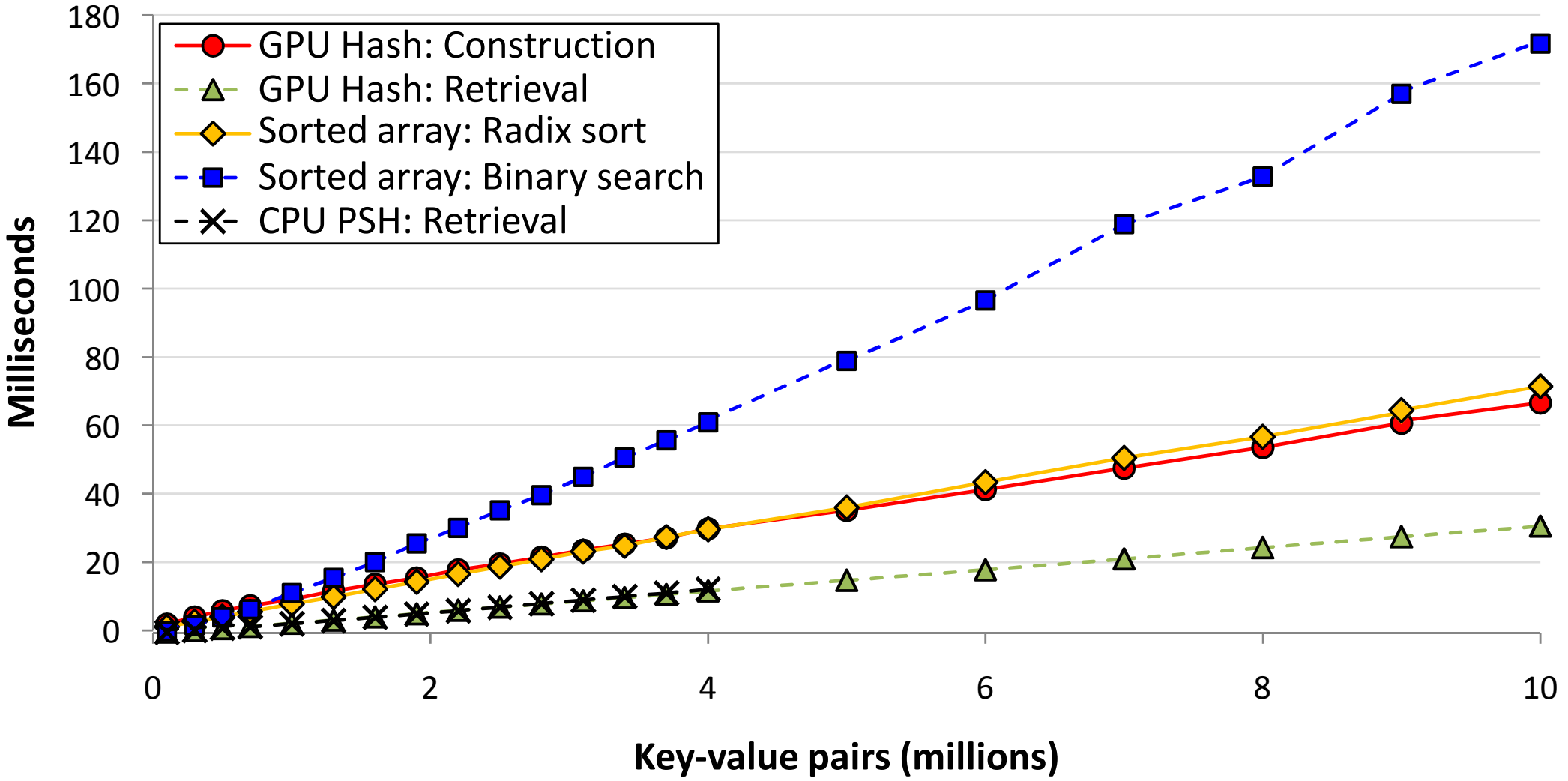


- Lookup procedure: in parallel, for each element:
  - Calculate h_1 & look in T_1;
  - Calculate h_2 & look in T_2; still $O(1)$ lookup

Cuckoo Construction Mechanics

- Level 1 created buckets of no more than 512 items
  - Average: 409; probability of overflow: $< 10^{-6}$
- Level 2: Assign each bucket to a thread block, construct cuckoo hash per bucket entirely within shared memory
  - Semantic: Multiple writes to same location must have one and only one winner
- Our implementation uses 3 tables of 192 elements each (load factor: 71%)
- What if it fails? New hash functions & start over.

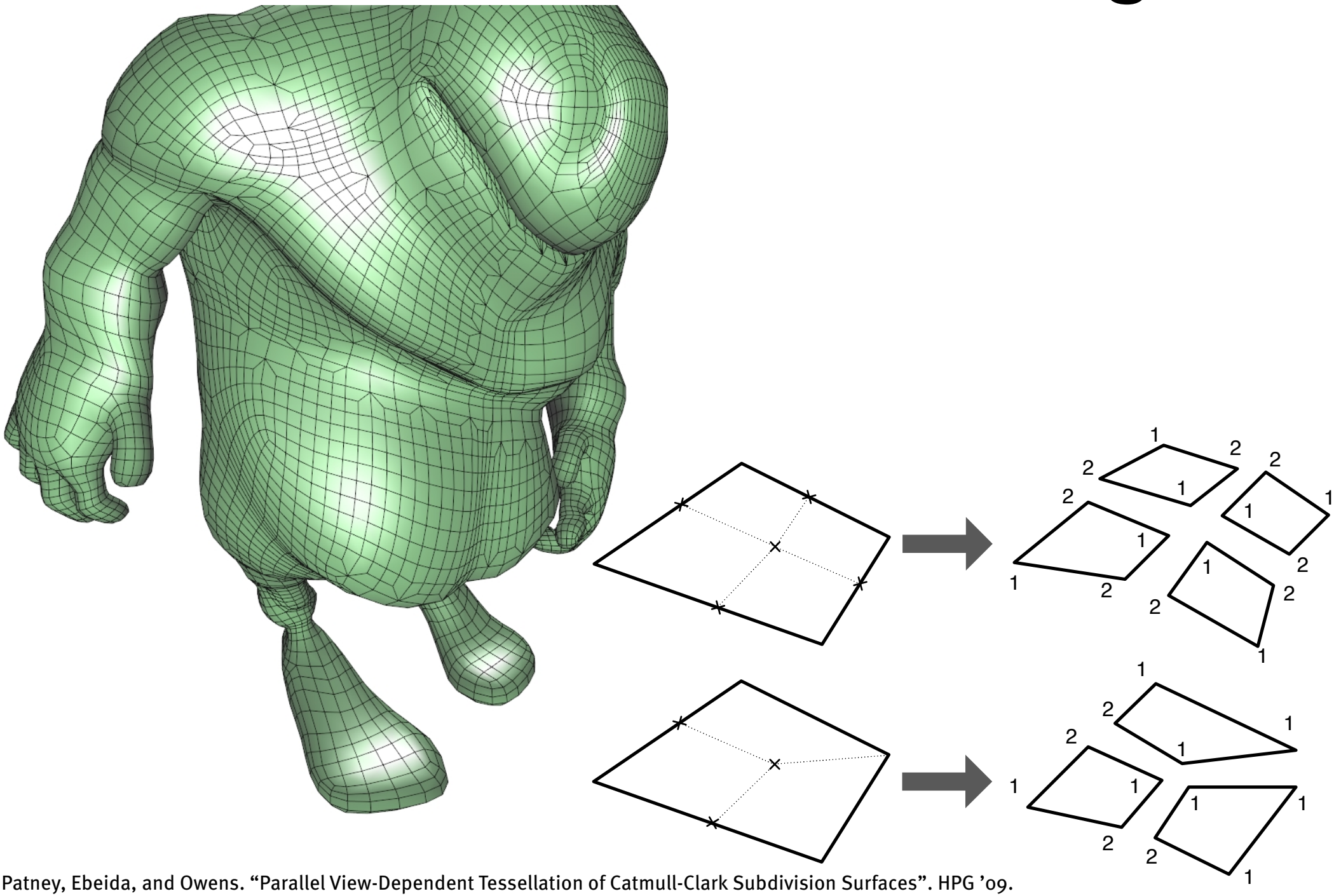
Timings on random voxel data



Hashing: Big Ideas

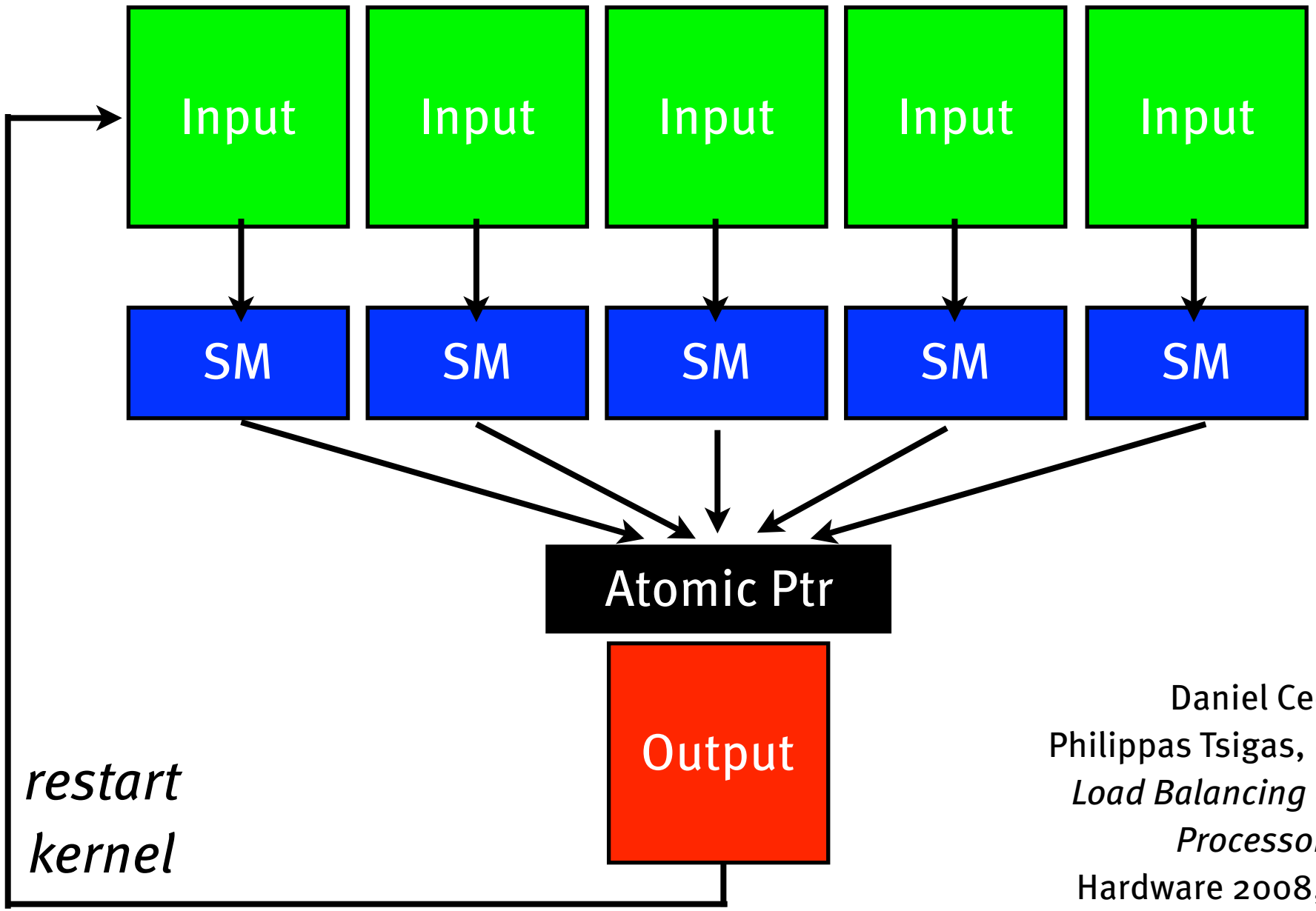
- Classic serial hashing techniques are a poor fit for a GPU.
  - Serialization, load balance
- Solving this problem required a different algorithm
  - Both hashing algorithms were new to the parallel literature
  - Hybrid algorithm was entirely new

Recursive Subdivision is Irregular



Patney, Ebeida, and Owens. “Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces”. HPG ’09.

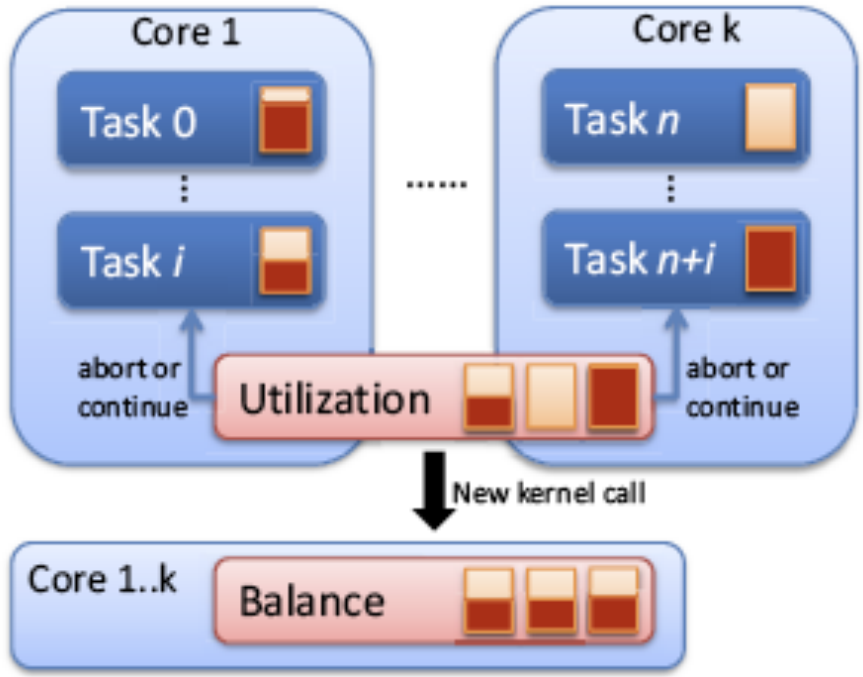
Static Task List



Daniel Cederman and Philippas Tsigas, *On Dynamic Load Balancing on Graphics Processors*. Graphics Hardware 2008, June 2008.

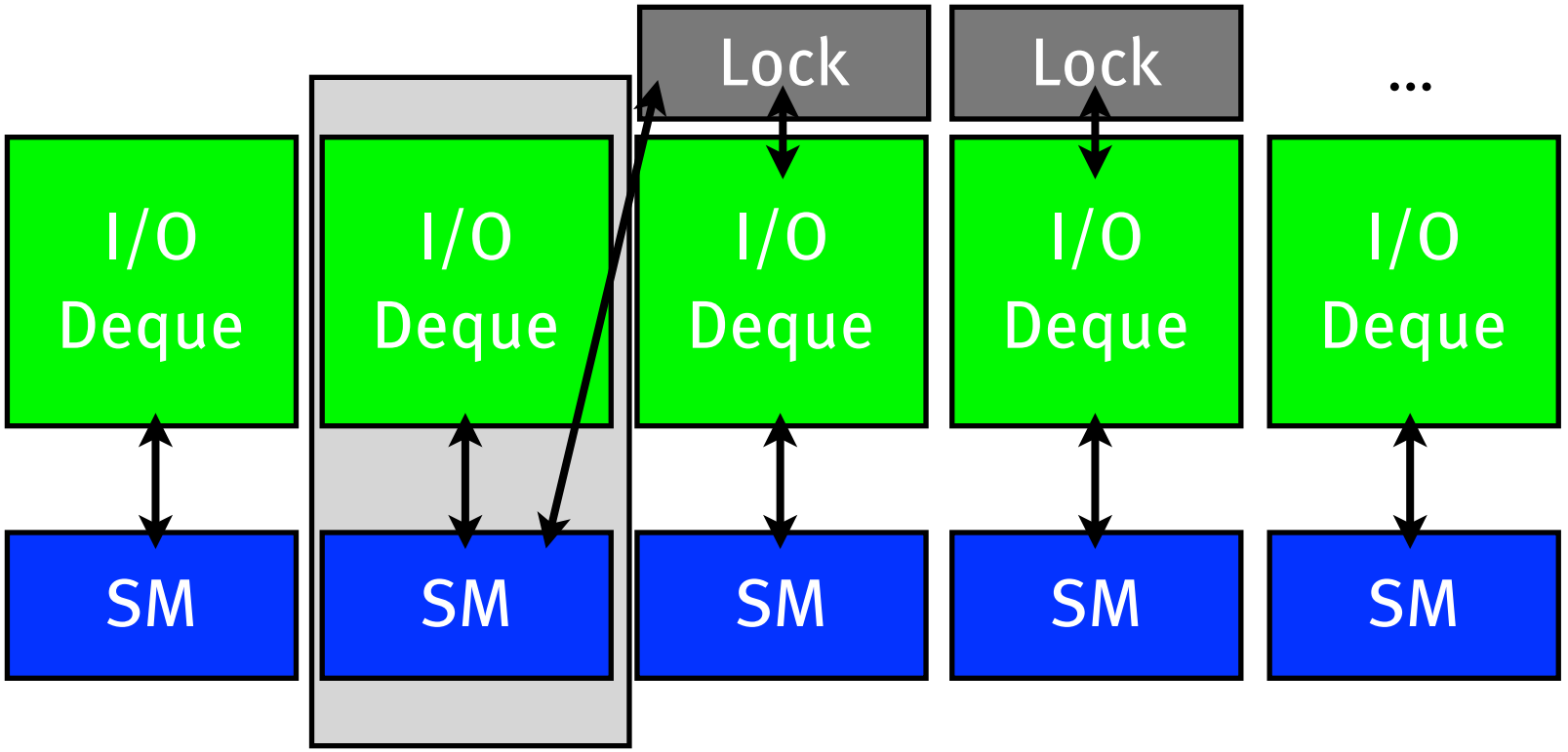
Private Work Queue Approach

- Allocate private work queue of tasks per core
  - Each core can add to or remove work from its local queue
- Cores mark self as idle if {queue exhausts storage, queue is empty}
- Cores periodically check global idle counter
- If global idle counter reaches threshold, rebalance work



gProximity: Fast Hierarchy Operations on GPU Architectures, Lauterbach, Mo, and Manocha, EG '10

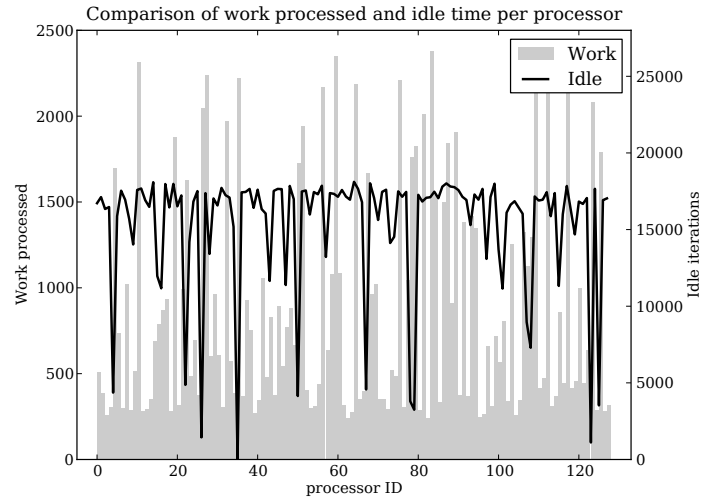
Work Stealing & Donating



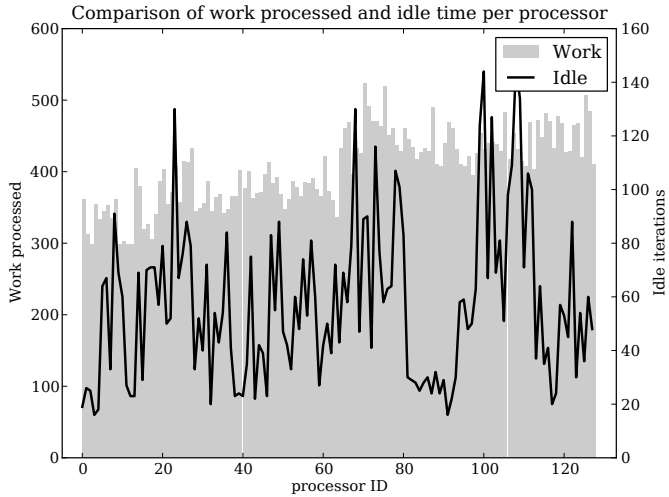
- Cederman and Tsigas: Stealing == best performance and scalability
- We added donating to minimize memory usage

Queuing Schemes

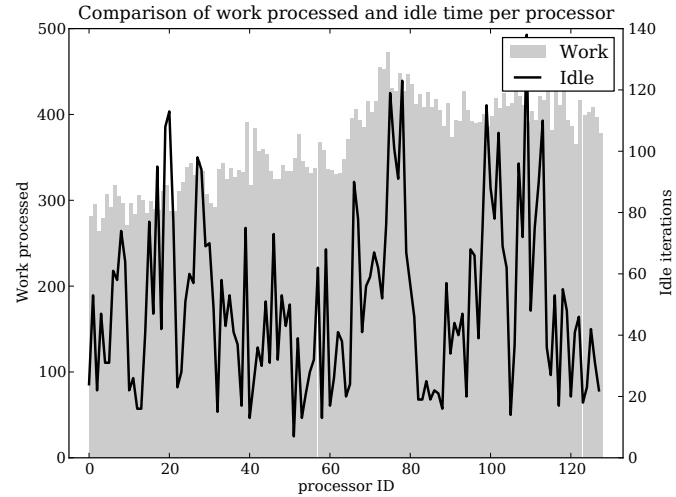
(a) Block Queuing



(b) Distributed Queuing



(c) Task Stealing



(d) Task Donating

DS Research Challenges

- String-based algorithms
- Building suffix trees (DNA sequence alignment)
- Graphs (vs. sparse matrix) and trees
- Dynamic programming
- Neighbor queries (kNN)
- Tuning
- True “parallel” data structures (not parallel versions of serial ones)?
- ***Incremental data structures***