Lecture 7:

Optimizations

Modern Parallel Computing

John Owens

EEC 289Q, UC Davis, Winter 2018

based on lecture by Kerry Seitz

Credits

- **Thanks to John Stratton and Anjul Patney for providing slides**

J. A. Stratton, C. Rodrigues, I.-J. Sung, L.-W. Chang, N. Anssari, G. Liu, W.-M. W. Hwu, and N. Obeid, "Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems," Computer, vol. 45, no. 8, pp. 26–32, 2012.

Announcements

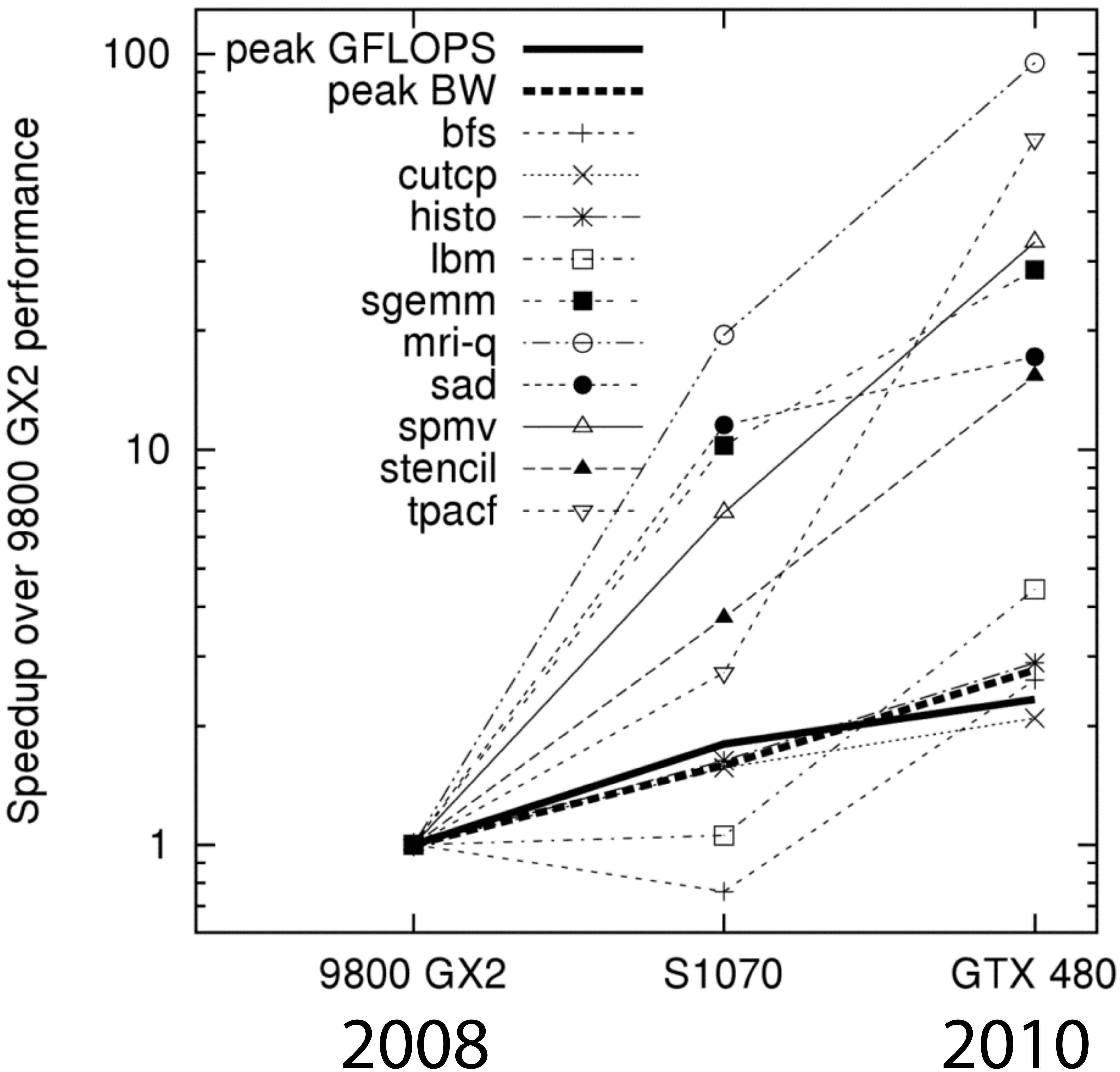
- HW2: Suggest you use cudaMallocManaged and linearize your input/output matrices
- Guest speakers next 3 Thursdays
  - Please come on time
  - Office hours will be perturbed, feel free to ask for alternate arrangements, will keep you posted

Basic Efficiency Rules

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

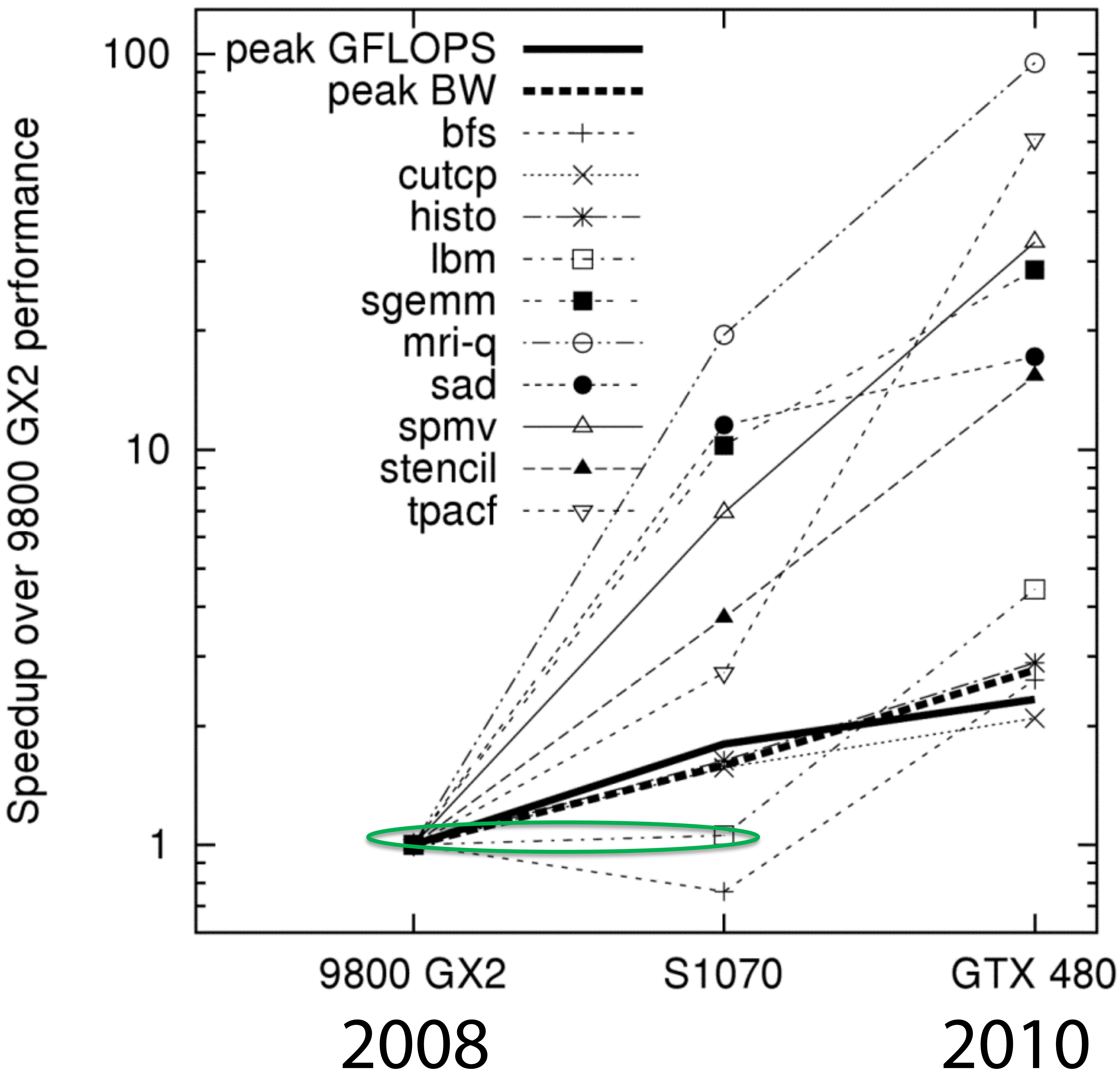
How much faster do applications really get each hardware generation?

Unoptimized Code Has Improved Drastically



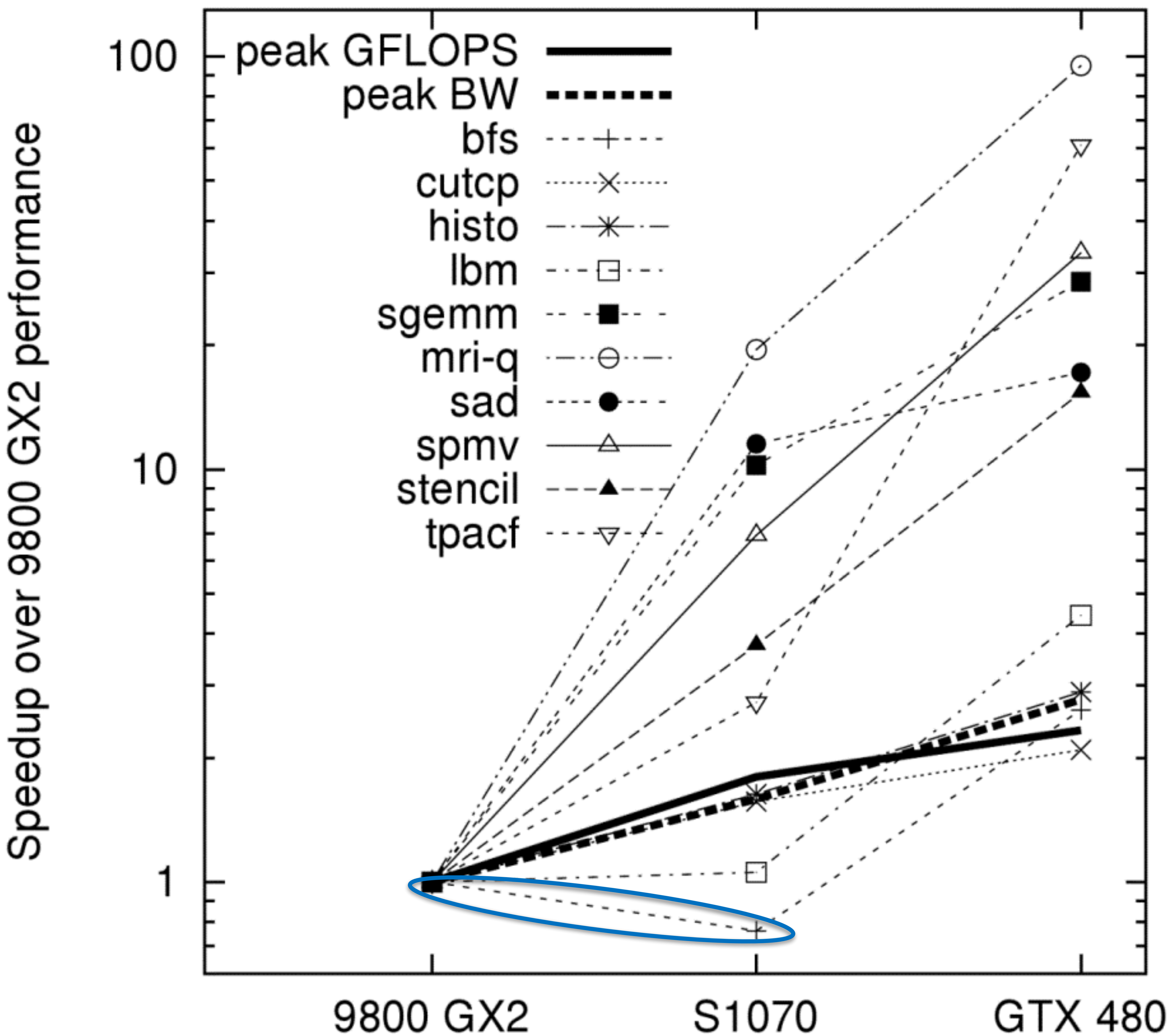
- Orders of magnitude speedup in many cases
- Hardware does not solve all problems

Unoptimized Code Has Improved Drastically



- Orders of magnitude speedup in many cases
- Hardware does not solve all problems
  - Coalescing (lbm)

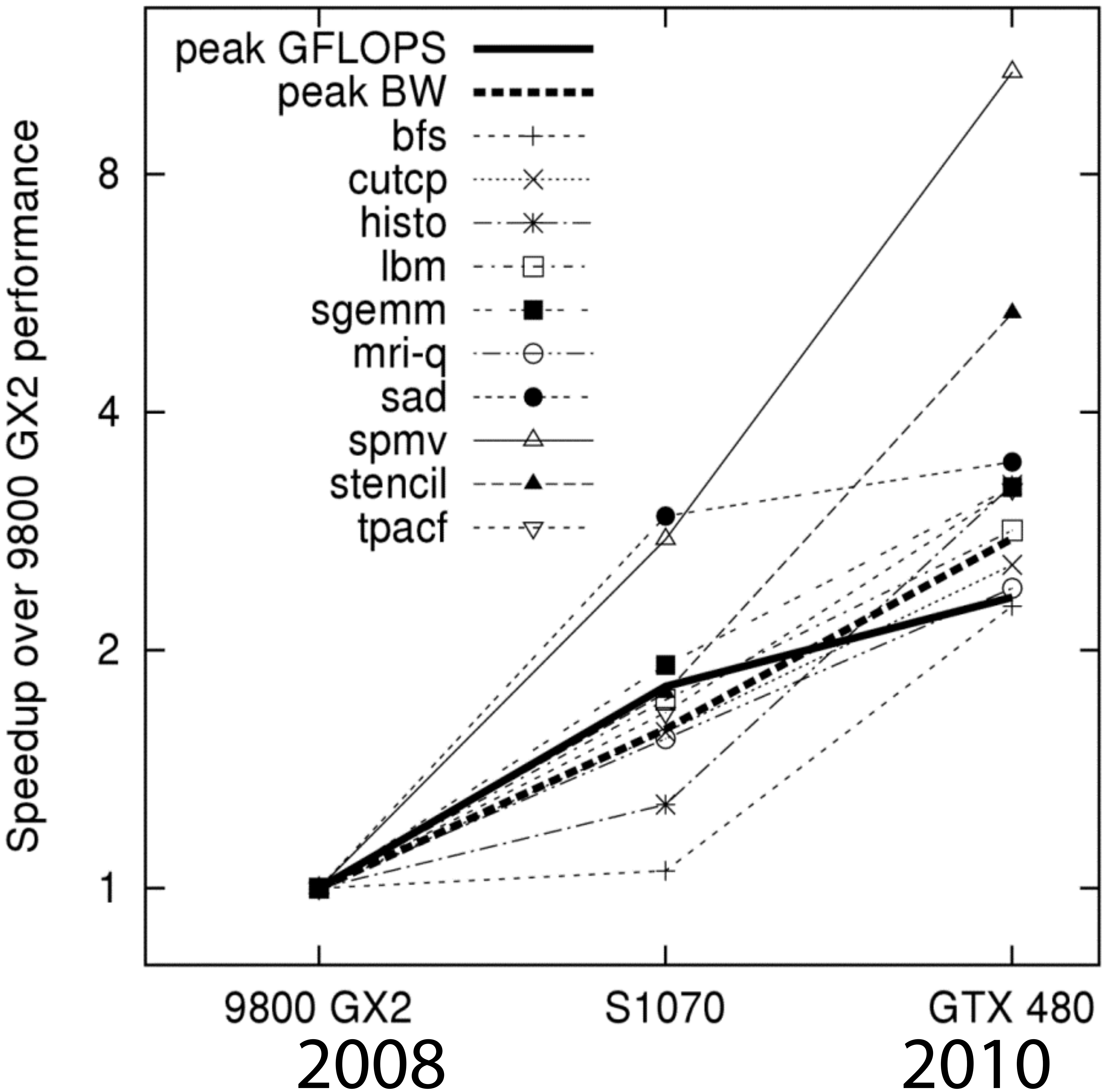
Unoptimized Code Has Improved Drastically



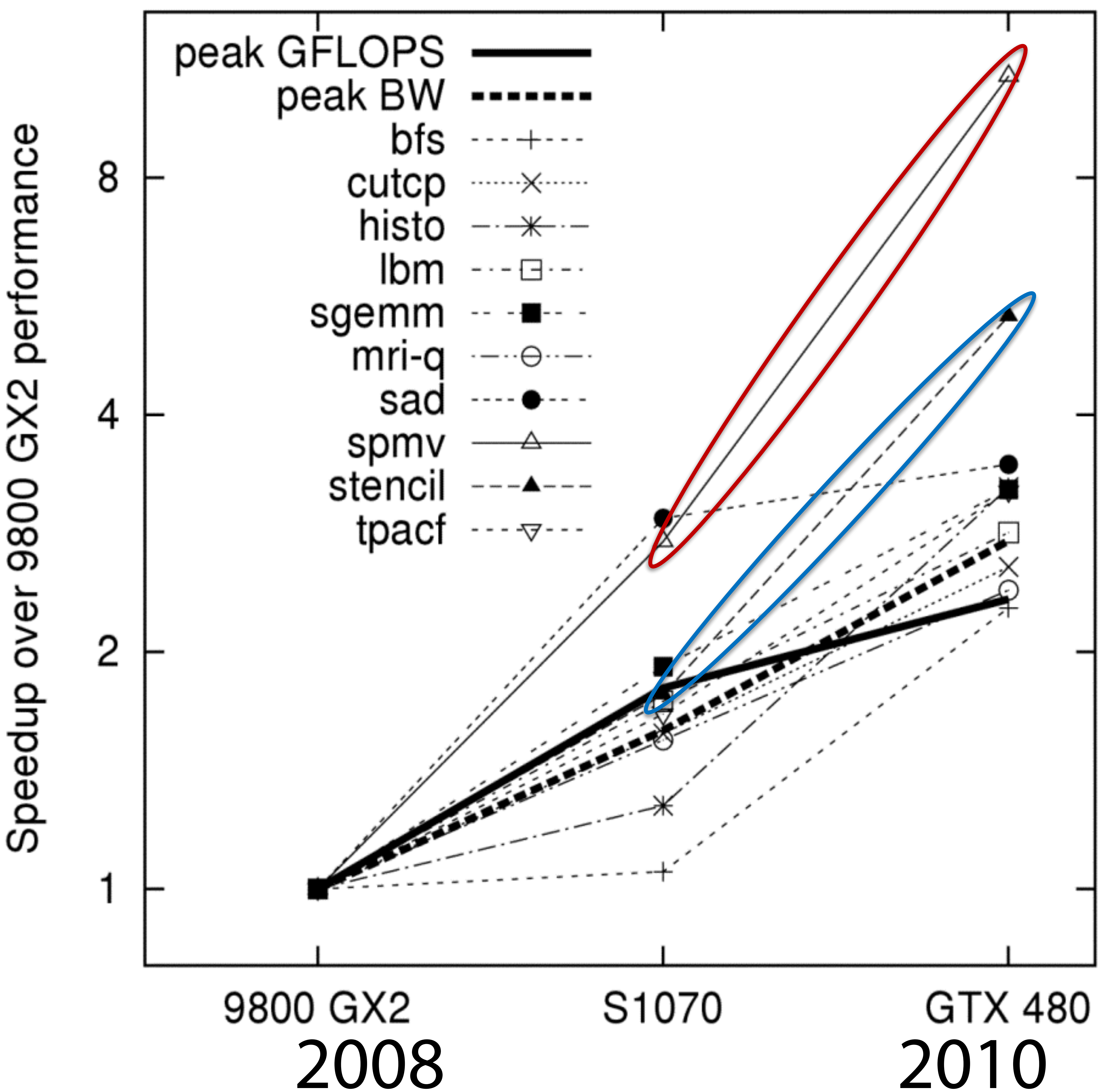
2008 2010

- Orders of magnitude speedup in many cases
- Hardware does not solve all problems
  - Coalescing (lbm)
  - Highly contentious atomics (bfs)

Optimized Code Is Improving Faster than "Peak Performance"

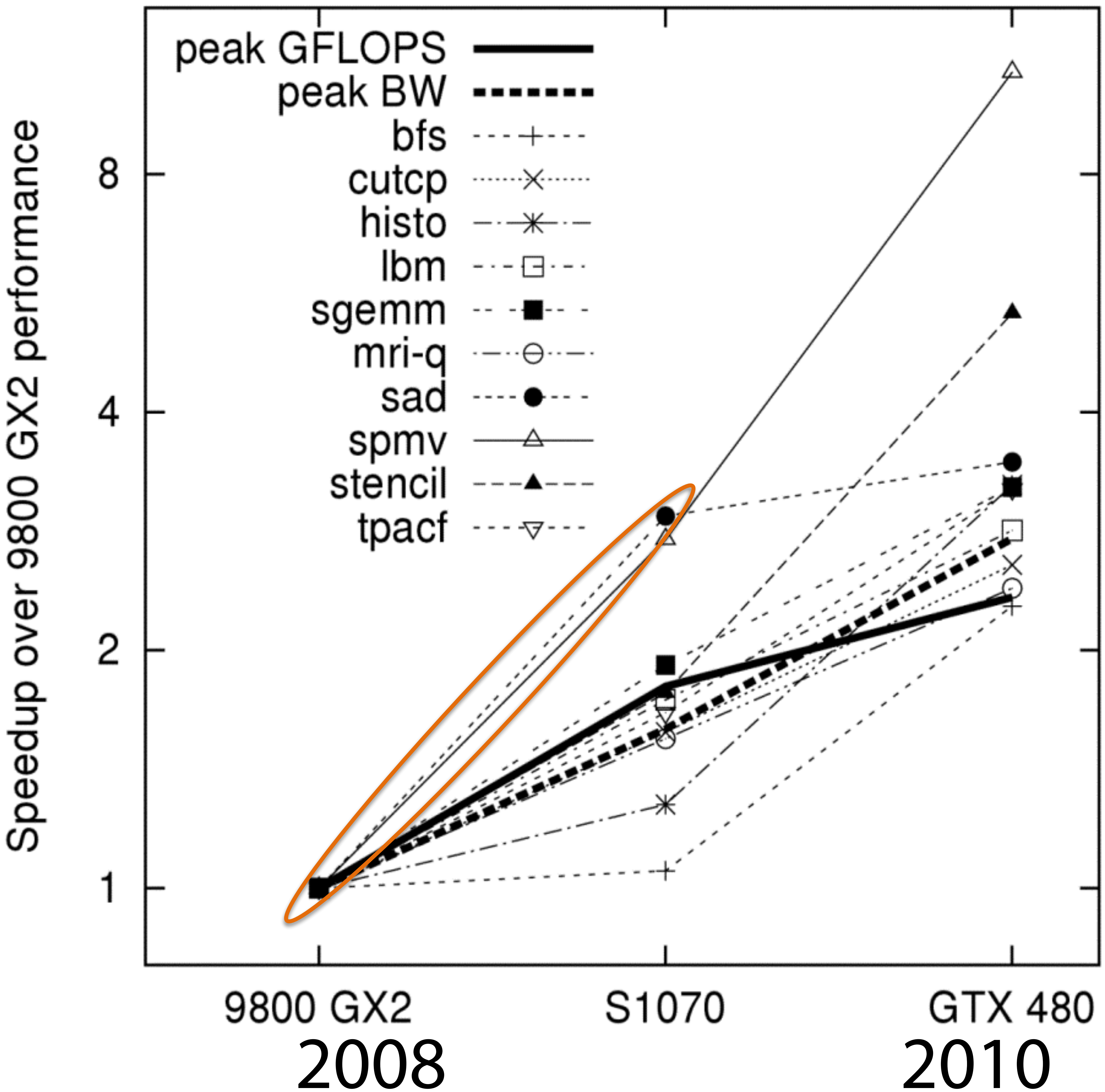


Optimized Code Is Improving Faster than "Peak Performance"



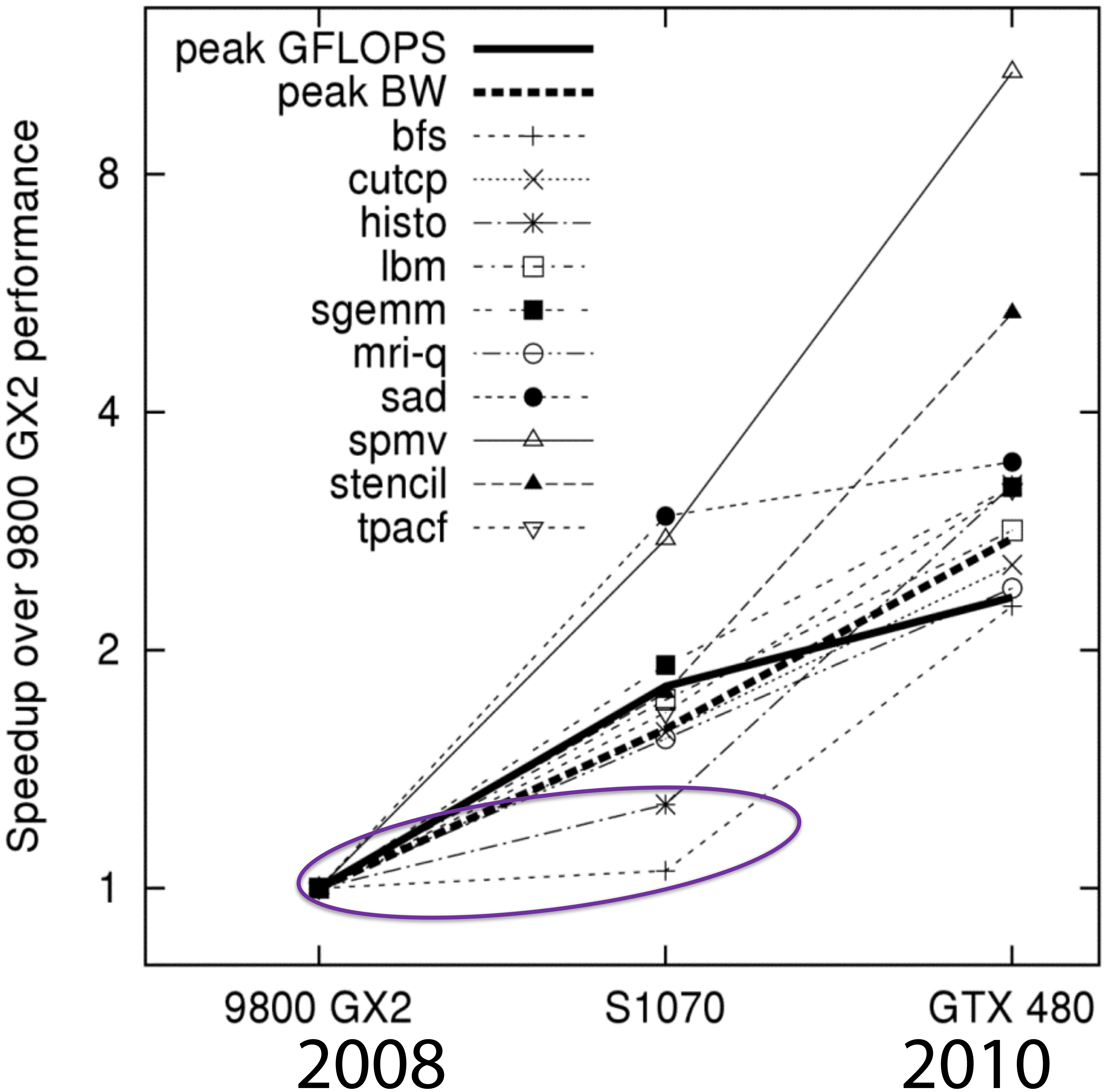
- **Caches capture locality scratchpad can't efficiently (spmv, stencil)**

Optimized Code Is Improving Faster than "Peak Performance"



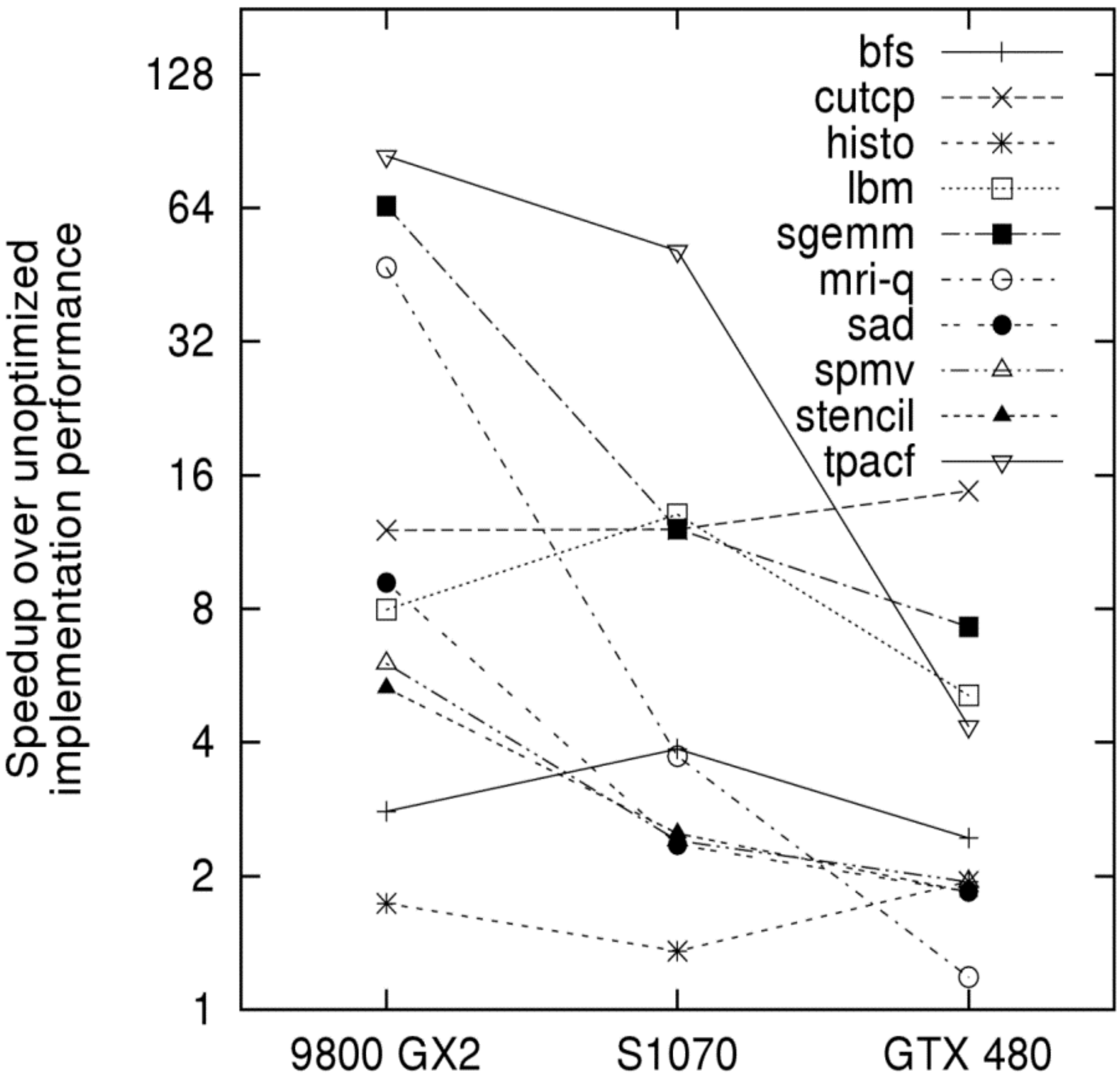
- Caches capture locality scratchpad can't efficiently (spmv, stencil)
- Increased local storage capacity enables extra optimization (sad)

Optimized Code Is Improving Faster than "Peak Performance"



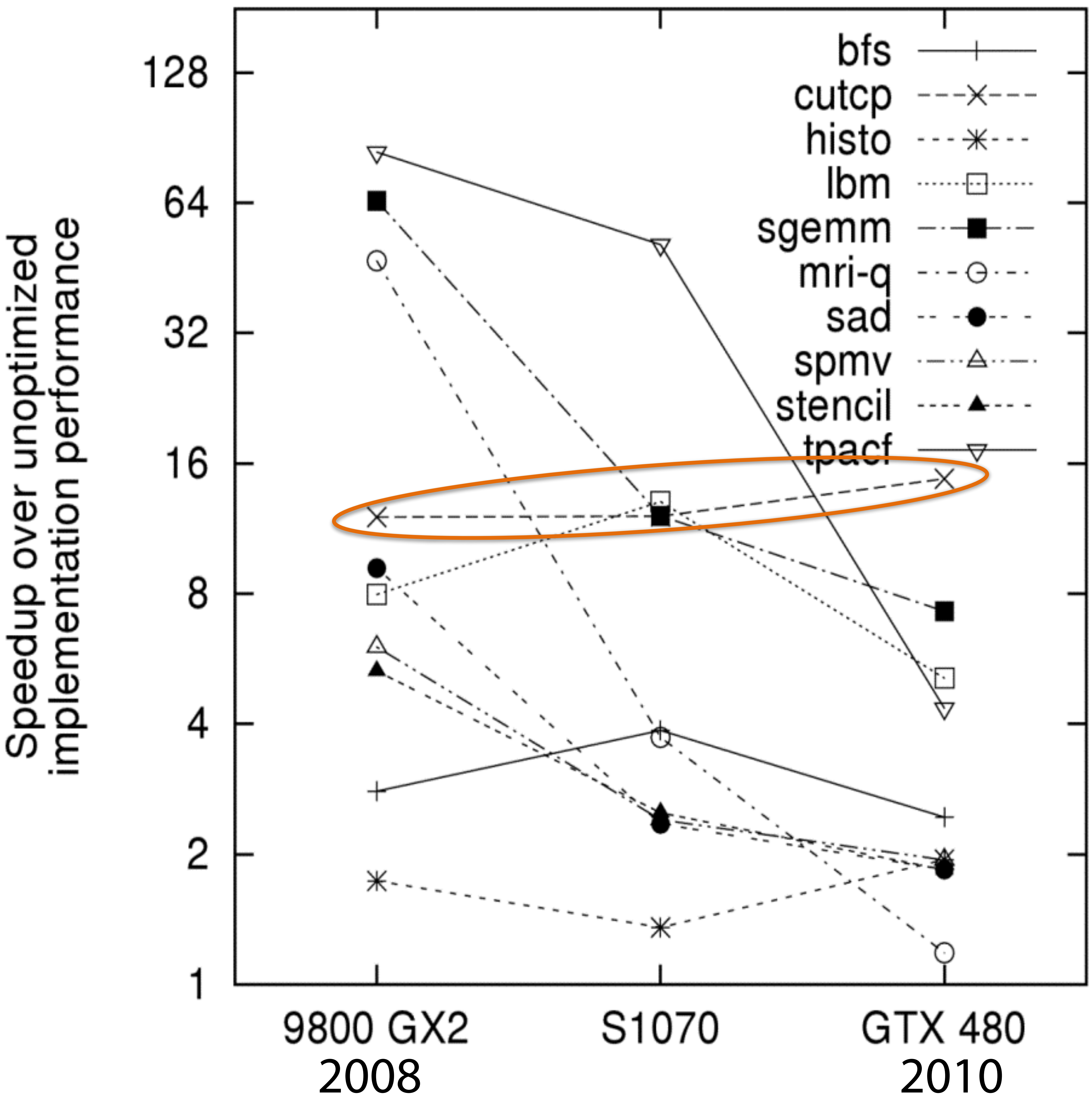
- Caches capture locality scratchpad can't efficiently (spmv, stencil)
- Increased local storage capacity enables extra optimization (sad)
- Some benchmarks need atomic throughput more than flops (bfs, histo)

Optimization Still Matters



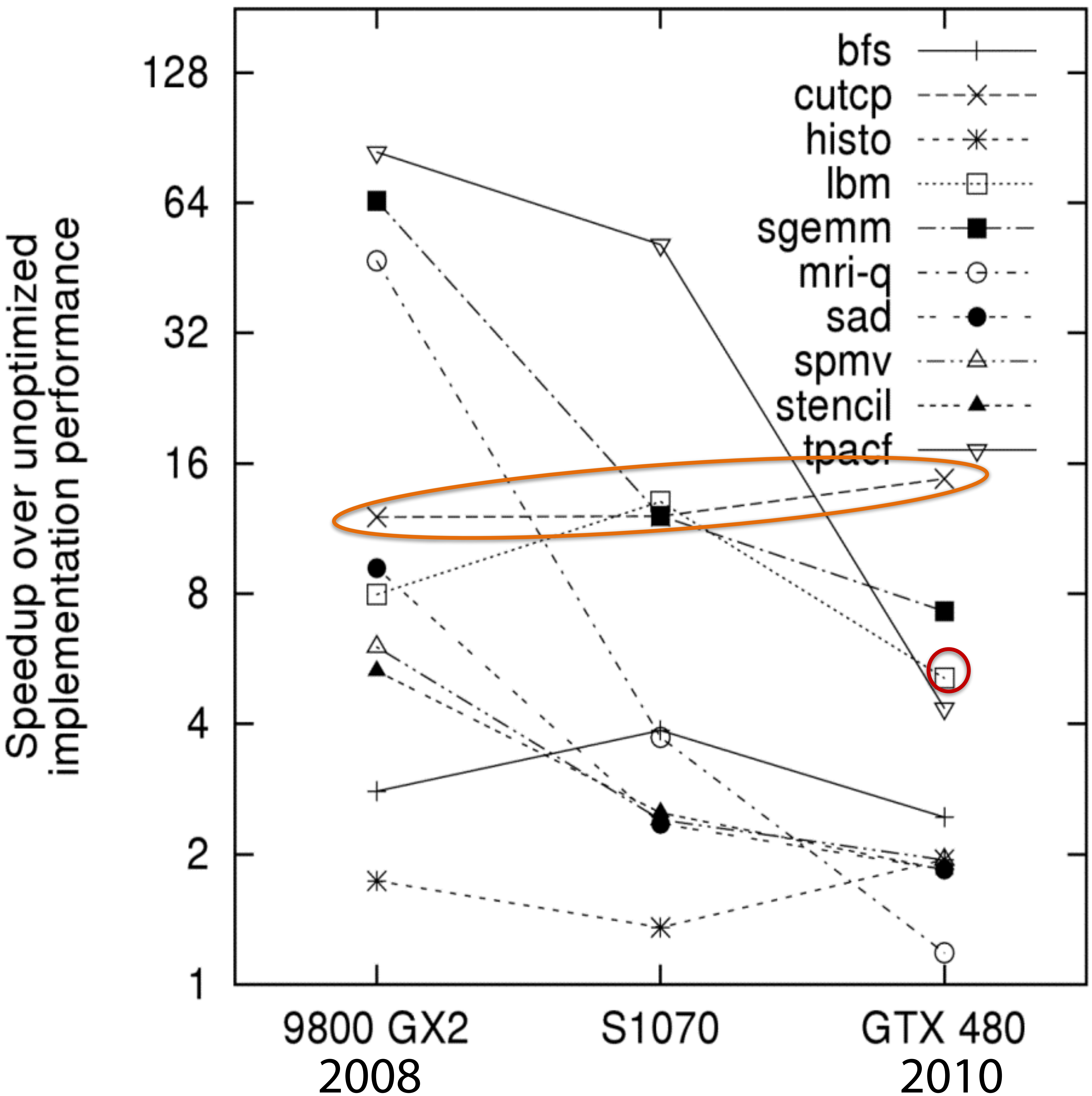
2008 2010

Optimization Still Matters



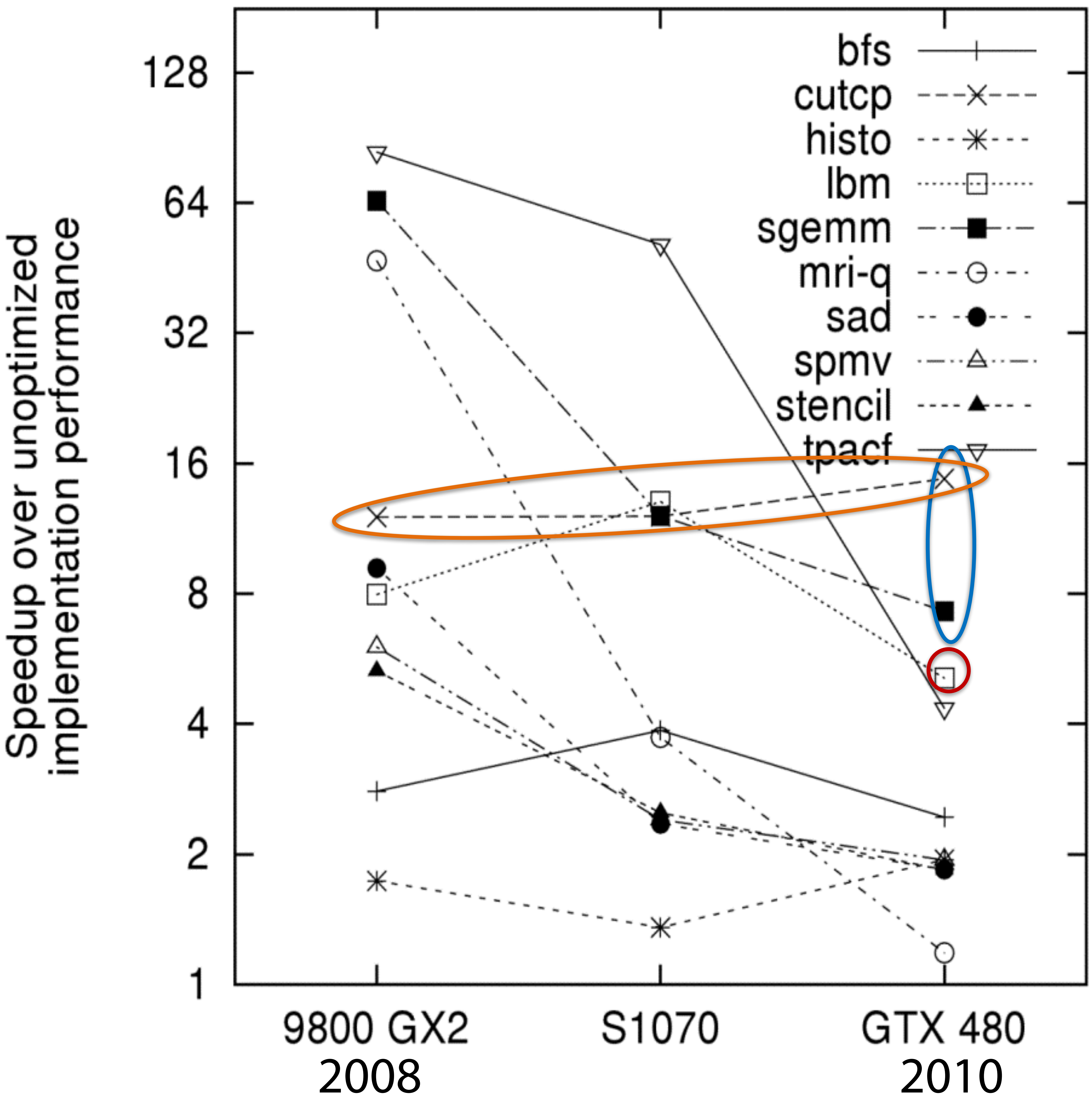
- **Hardware never changes algorithmic complexity (cutcp)**

Optimization Still Matters



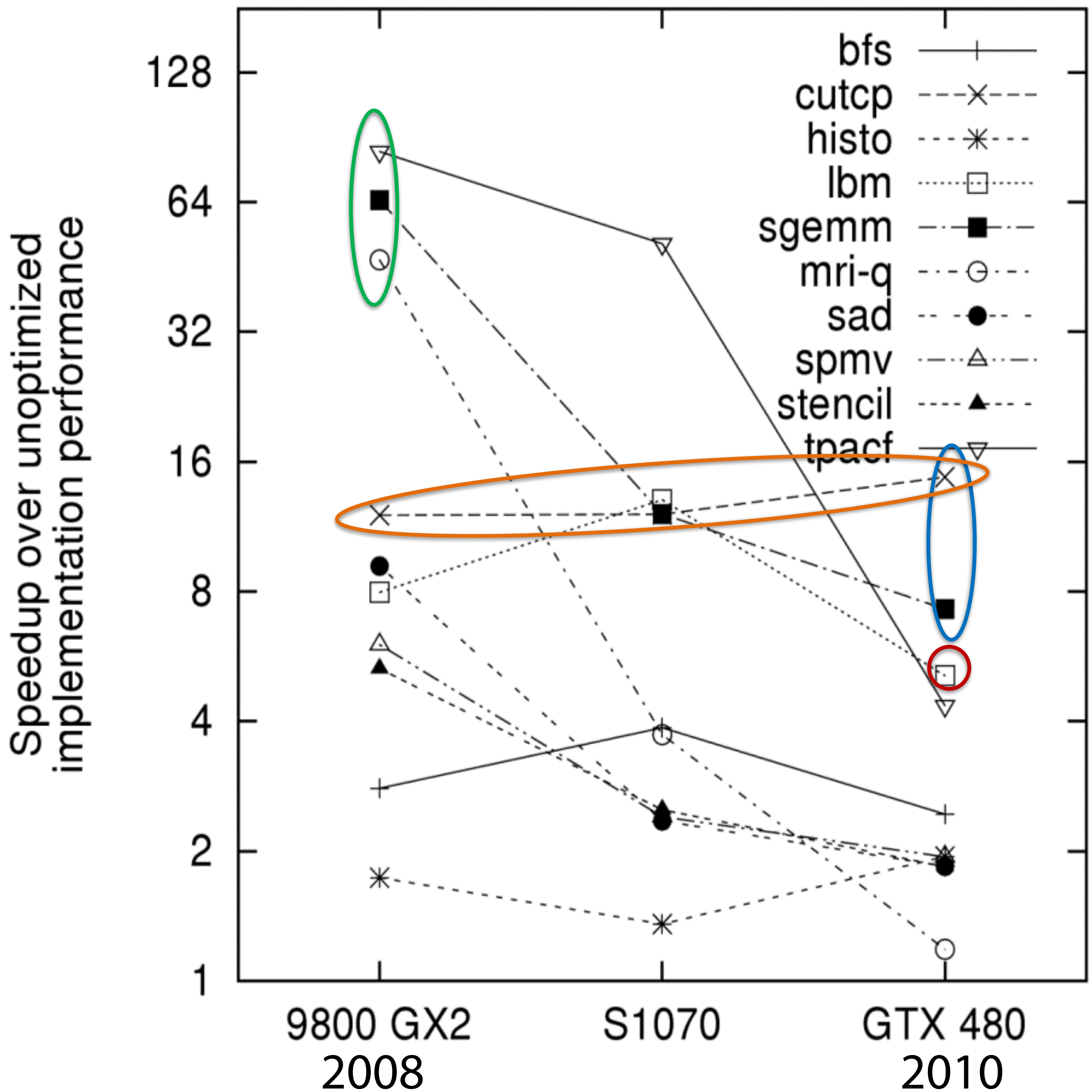
- **Hardware never changes algorithmic complexity (cutcp)**
- **Caches do not solve layout problems for big data (lbm)**

Optimization Still Matters



- Hardware never changes algorithmic complexity (cutcp)
- Caches do not solve layout problems for big data (lbm)
- Coarsening still makes a big difference (cutcp, sgemm)

Optimization Still Matters



- **Hardware never changes algorithmic complexity (cutcp)**
- **Caches do not solve layout problems for big data (lbm)**
- **Coarsening still makes a big difference (cutcp, sgemm)**
- **Many artificial performance cliffs are gone (sgemm, tpacf, mri-q)**

Speculations and Takeaways

Speculations and Takeaways

- Optimizations still necessary today are unlikely to be magically solved by future hardware
  - Still necessary for highly parallel CPUs, after all

Speculations and Takeaways

- Optimizations still necessary today are unlikely to be magically solved by future hardware
  - Still necessary for highly parallel CPUs, after all
- Features matter just as much as FLOPS and GBytes/sec on lots of applications
  - Having a cache is critical, period

Speculations and Takeaways

- Optimizations still necessary today are unlikely to be magically solved by future hardware
  - Still necessary for highly parallel CPUs, after all
- Features matter just as much as FLOPS and GBytes/sec on lots of applications
  - Having a cache is critical, period
- Beware of unscalable implementation decisions
  - Global contention and synchronization will get worse

Application Survey

- Surveyed the GPU Computing Gems chapters
- Studied the Parboil benchmarks in detail

- Results:
- Nine (for now) major categories of optimization transformations
  - Performance impact of individual optimizations on certain Parboil benchmarks included in the paper

Sparse matrix-vector multiplication (SpMV)

Sparse matrix
(most elements are zero)

$$y = Ax$$

Dense vectors
(most elements are non-zero)

Sparse matrix-vector multiplication (SpMV)

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{2,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} xA_{0,0} + yA_{0,1} + zA_{0,2} + wA_{0,3} \\ xA_{1,0} + yA_{1,1} + zA_{1,2} + wA_{1,3} \\ xA_{2,0} + yA_{2,1} + zA_{2,2} + wA_{2,3} \\ xA_{3,0} + yA_{3,1} + zA_{3,2} + wA_{3,3} \end{pmatrix}$$

Sparse matrix-vector multiplication (SpMV)

Row in input matrix => row in output vector

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{2,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} xA_{0,0} + yA_{0,1} + zA_{0,2} + wA_{0,3} \\ xA_{1,0} + yA_{1,1} + zA_{1,2} + wA_{1,3} \\ xA_{2,0} + yA_{2,1} + zA_{2,2} + wA_{2,3} \\ xA_{3,0} + yA_{3,1} + zA_{3,2} + wA_{3,3} \end{pmatrix}$$

Sparse matrix-vector multiplication (SpMV)

Row in input matrix => row in output vector

$$\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{2,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} xA_{0,0} + yA_{0,1} + zA_{0,2} + wA_{0,3} \\ xA_{1,0} + yA_{1,1} + zA_{1,2} + wA_{1,3} \\ xA_{2,0} + yA_{2,1} + zA_{2,2} + wA_{2,3} \\ xA_{3,0} + yA_{3,1} + zA_{3,2} + wA_{3,3} \end{pmatrix}$$

Elements from input vector in every output row

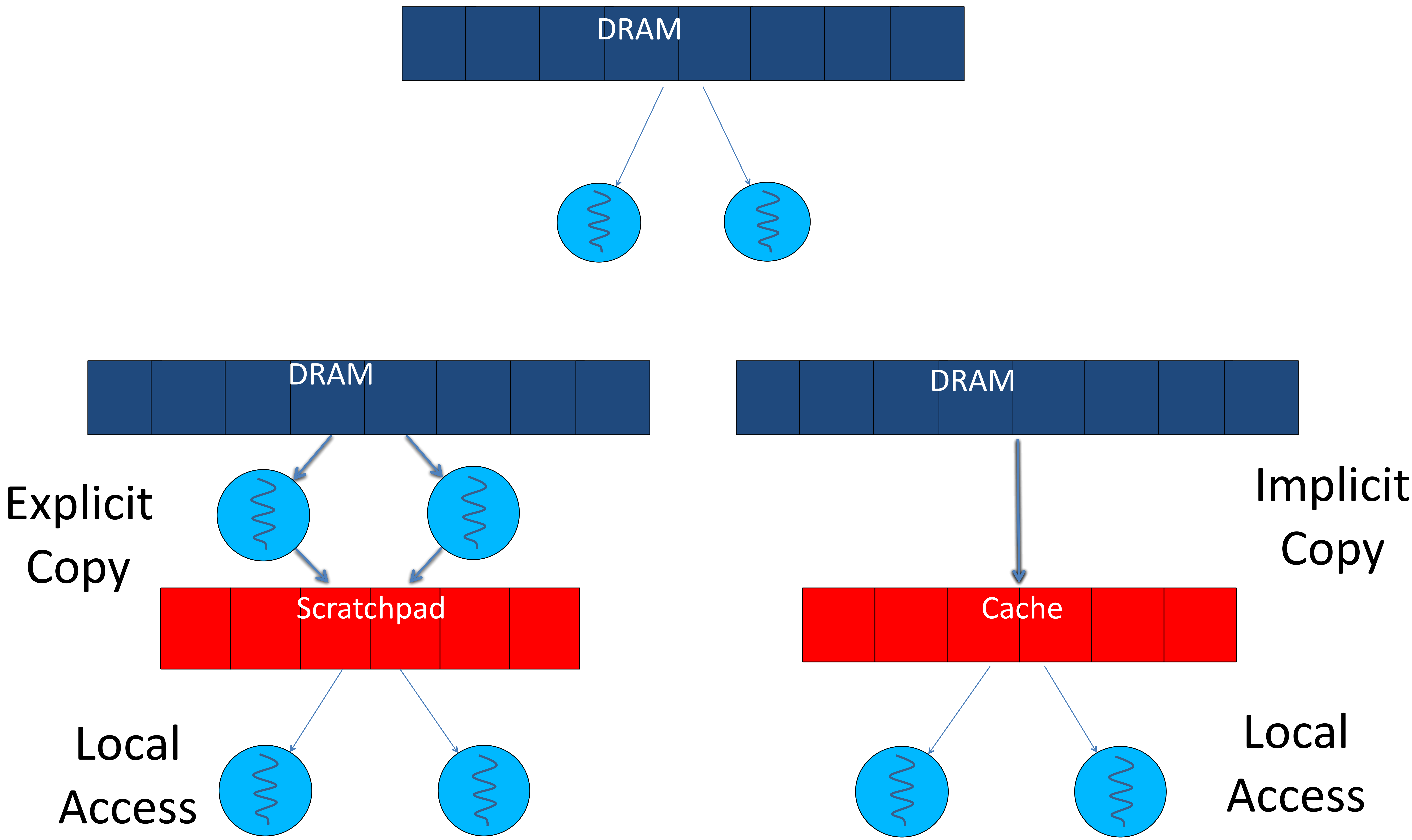
SpMV Kernel

```
__global__ void spmv(float **m, float *v, float *y) {
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    int col = threadIdx.y + blockIdx.y * blockDim.y;

    y[row] += m[row][col] * v[col];
}
```

SpMV Kernel

```
__global__ void spmv(float **m, float *v, float *y) {
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    int col = threadIdx.y + blockIdx.y * blockDim.y;

    y[row] += m[row][col] * v[col];
    atomicAdd(&y[row], m[row][col] * v[col]);
}
```

1: (Input) Data Access Tiling



Data-Access Tiling

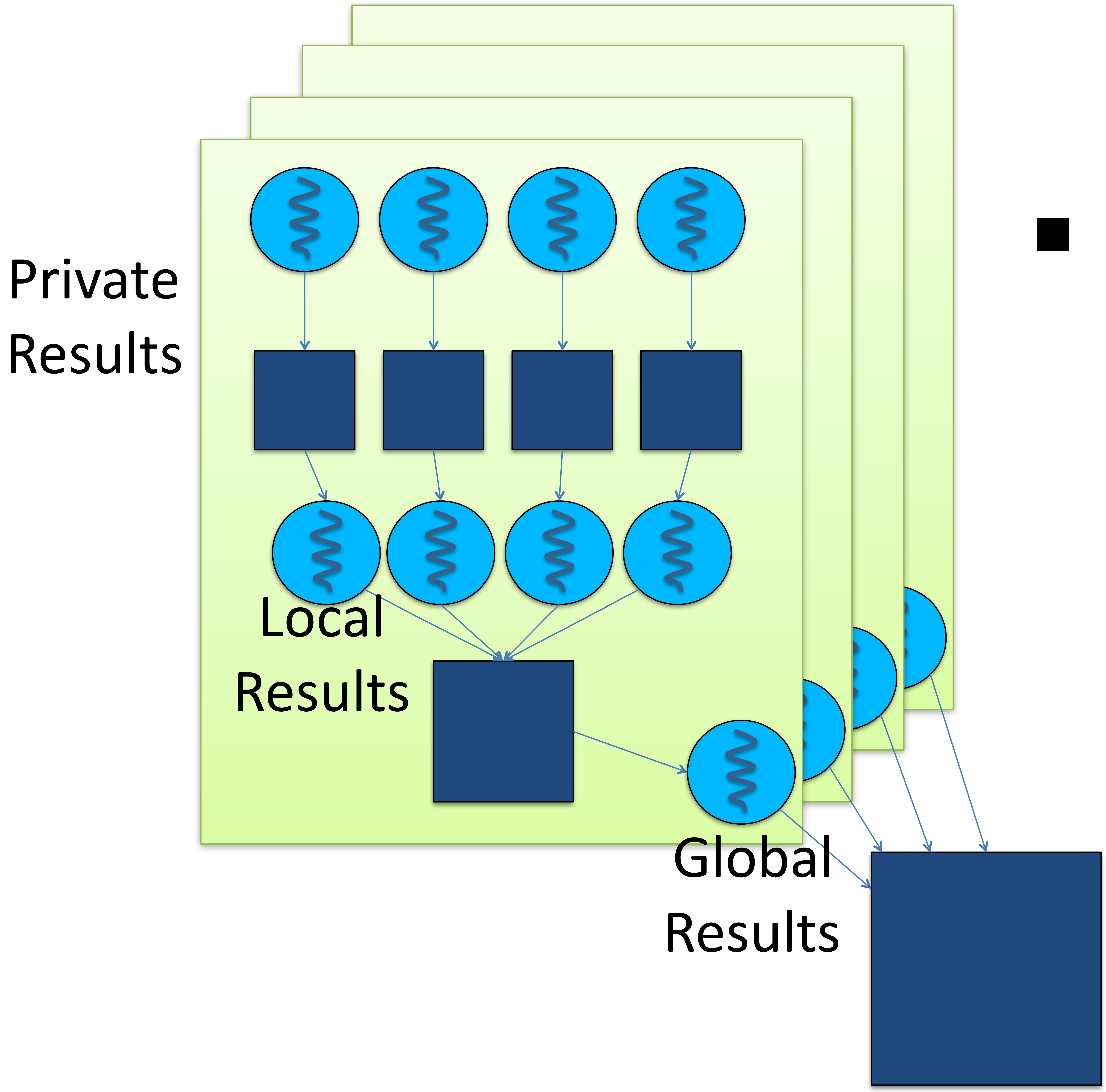
```
__global__ void spmv(float **m, float *v, float *y) {
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    int col = threadIdx.y + blockIdx.y * blockDim.y;

    __shared__ float vs[VECTOR_SIZE];
    if(row == 0) {
      vs[col] = v[col];
    }
    __syncthreads()

    atomicAdd(&y[row], m[row][col] * vs[col]);
}
```

1. (Input) Data Access Tiling

- **Pro: Better use of the memory system**
  - **Coalesced accesses**
  - **Data reuse**

- **Con: Reduced scheduling flexibility**
  - **Threads must synchronize**
  - **Larger shared memory use -> fewer blocks per SM**

2. (Output) Privatization



- Avoid contention by aggregating updates locally
- Requires storage resources to keep copies of data structures

Output Privatization

```
__global__ void spmv(float **m, float *v, float **yLocal) {
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    int col   = threadIdx.y + blockIdx.y * blockDim.y;

    yLocal[row][col] = m[row][col] * v[col]);
}


spmv<<<…>>>(m, v, yLocal);
for(int row = 0; row < NUM_ROWS; ++row) {
    y[row] = reduce("+", yLocal[row]);
}
```

2. (Output) Privatization

- **Pro: Reduce write contention**
  - Don't need atomics for every update

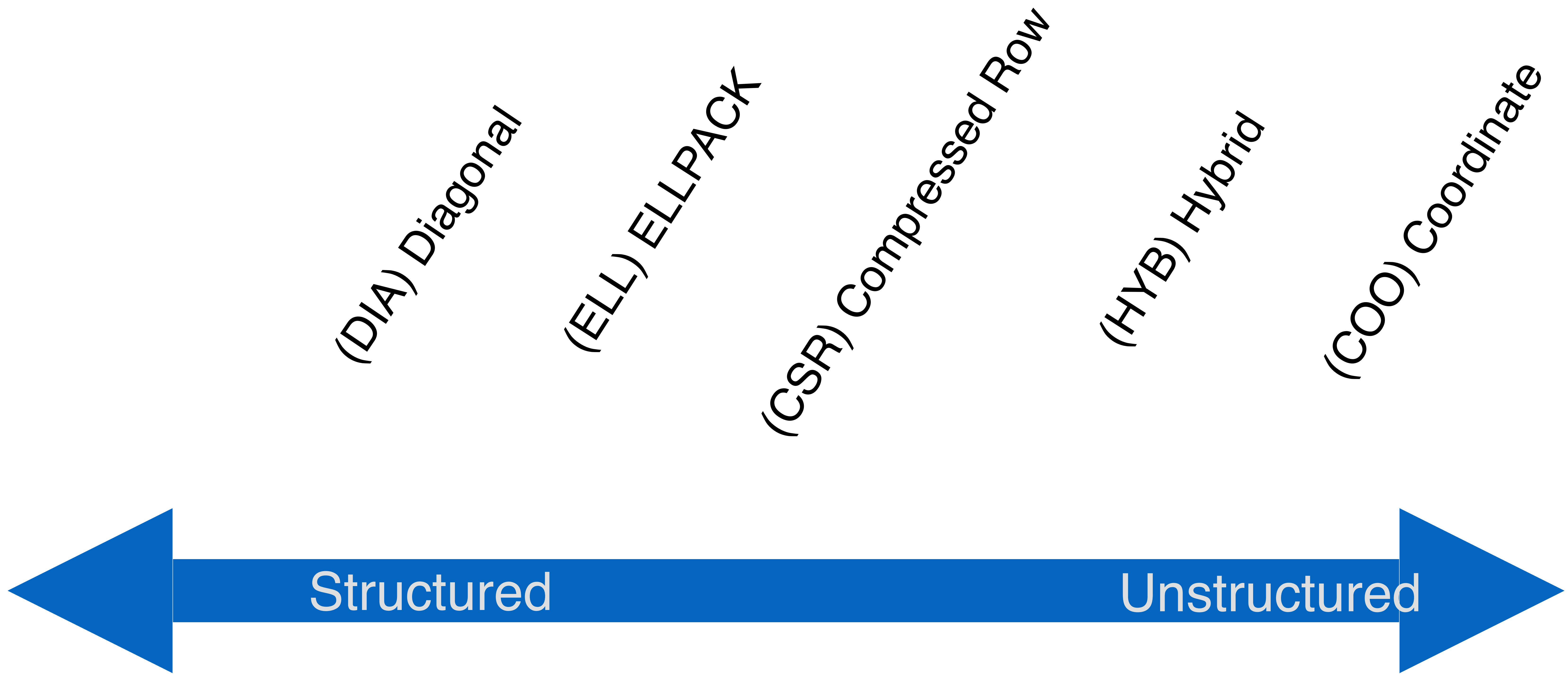
- **Con: More memory usage**
  - Need copy of data per thread

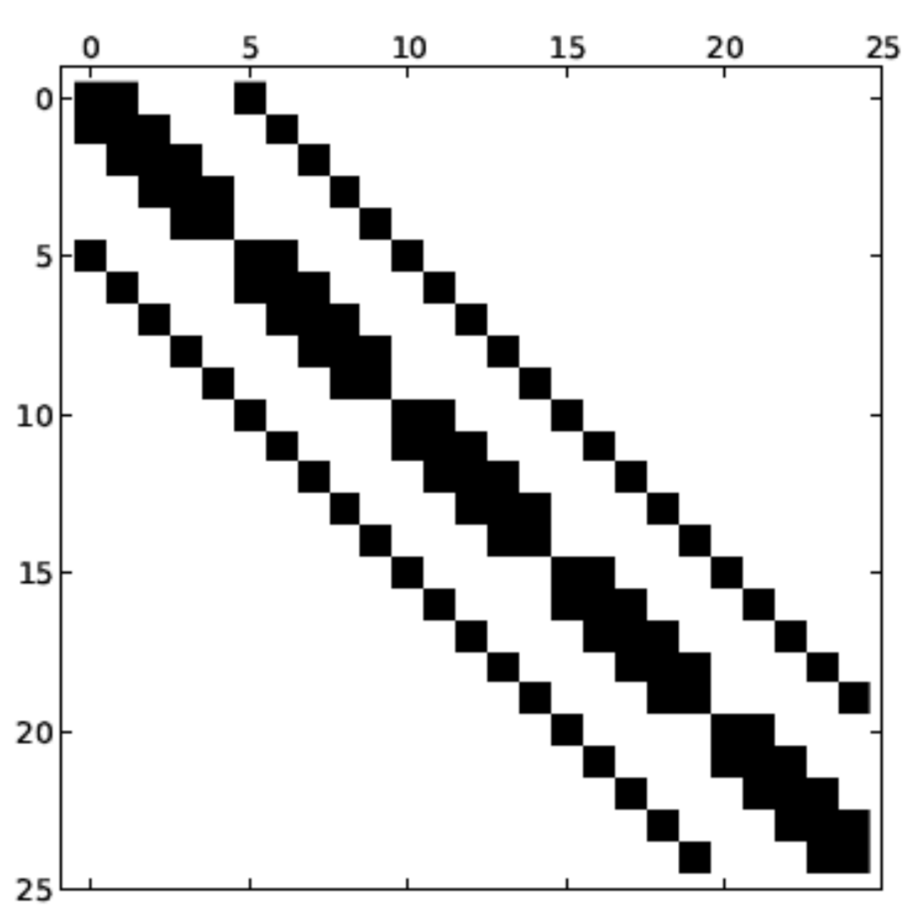
- **Variant: One copy per block + smem atomics**
  - smem atomics are faster on newer hardware

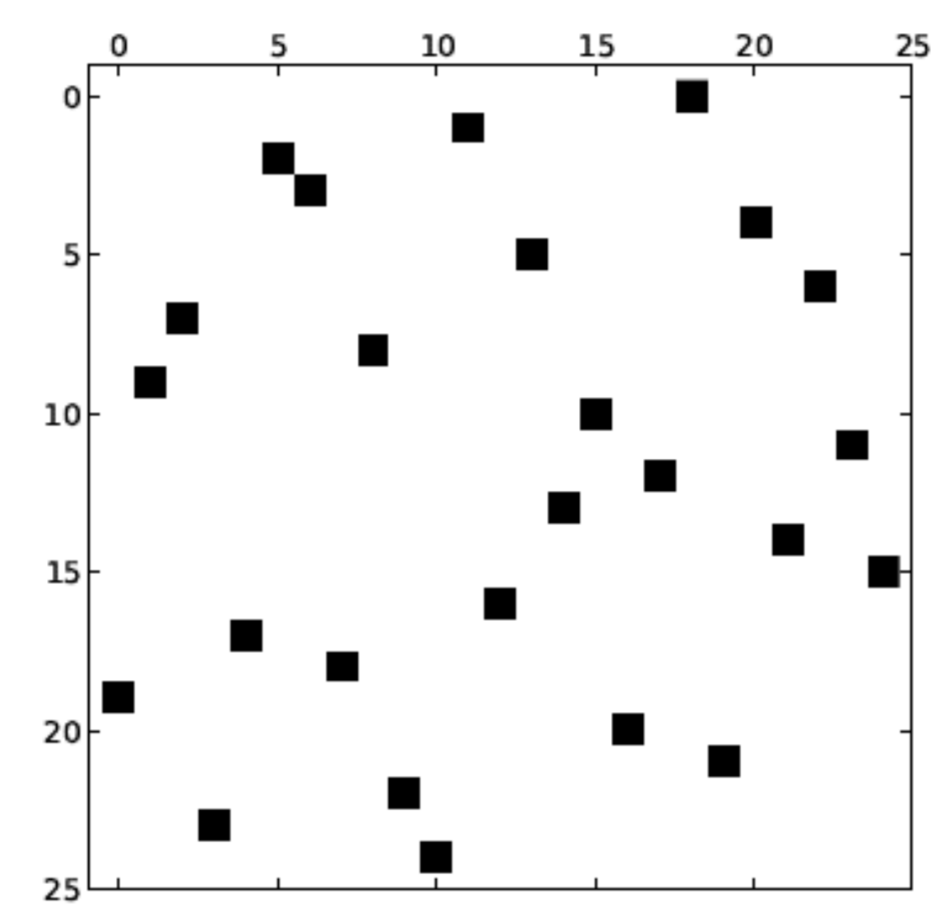
Output Privatization (Variant)

```
__global__ void spmv(float **m, float *v, float *y) {
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    int col   = threadIdx.y + blockIdx.y * blockDim.y;

    __shared__ float ys[VECTOR_SIZE];
    if(col == 0) {
      ys[row] = 0;
    }
    __syncthreads()
    atomicAdd(&ys[row], m[row][col] * v[col]); // Shared memory atomic
    __syncthreads()
    atomicAdd(&y[row], ys[row]);  // Global memory atomic
}
```

Storage Format Comparison







Slide credit: Nathan Bell

Coordinate Format (COO)

$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Coordinate Format (COO)

$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Non-zero values = { 3, 1, 2, 4, 1, 1, 1 }

Row indices = { 0, 0, 2, 2, 2, 3, 3 }

Column indices = { 0, 2, 1, 2, 3, 0, 3 }

Coordinate Format (COO)

$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

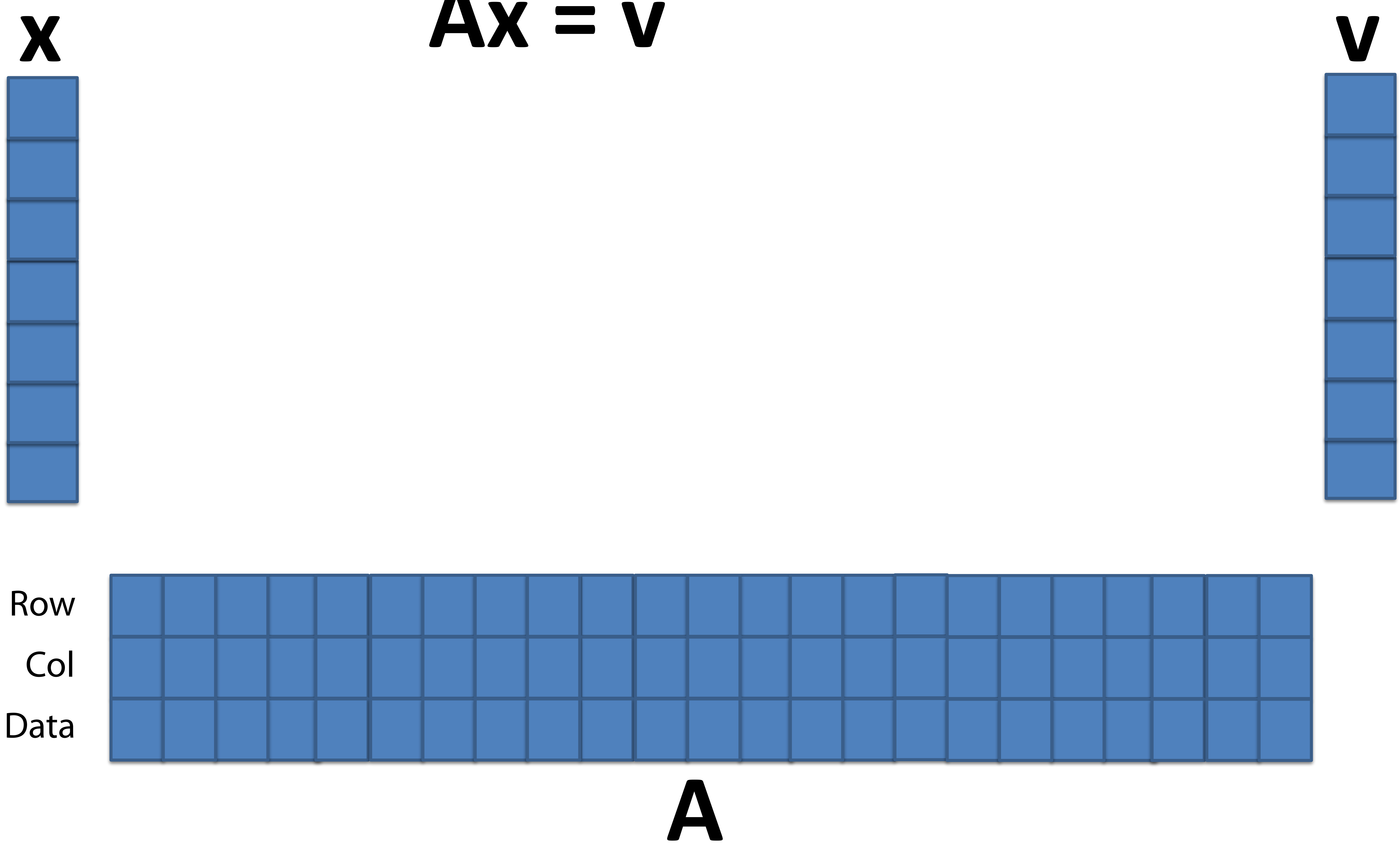
Non-zero values = { 3, 1, 2, 4, 1, 1, 1 }

Row indices = { 0, 0, 2, 2, 2, 3, 3 }

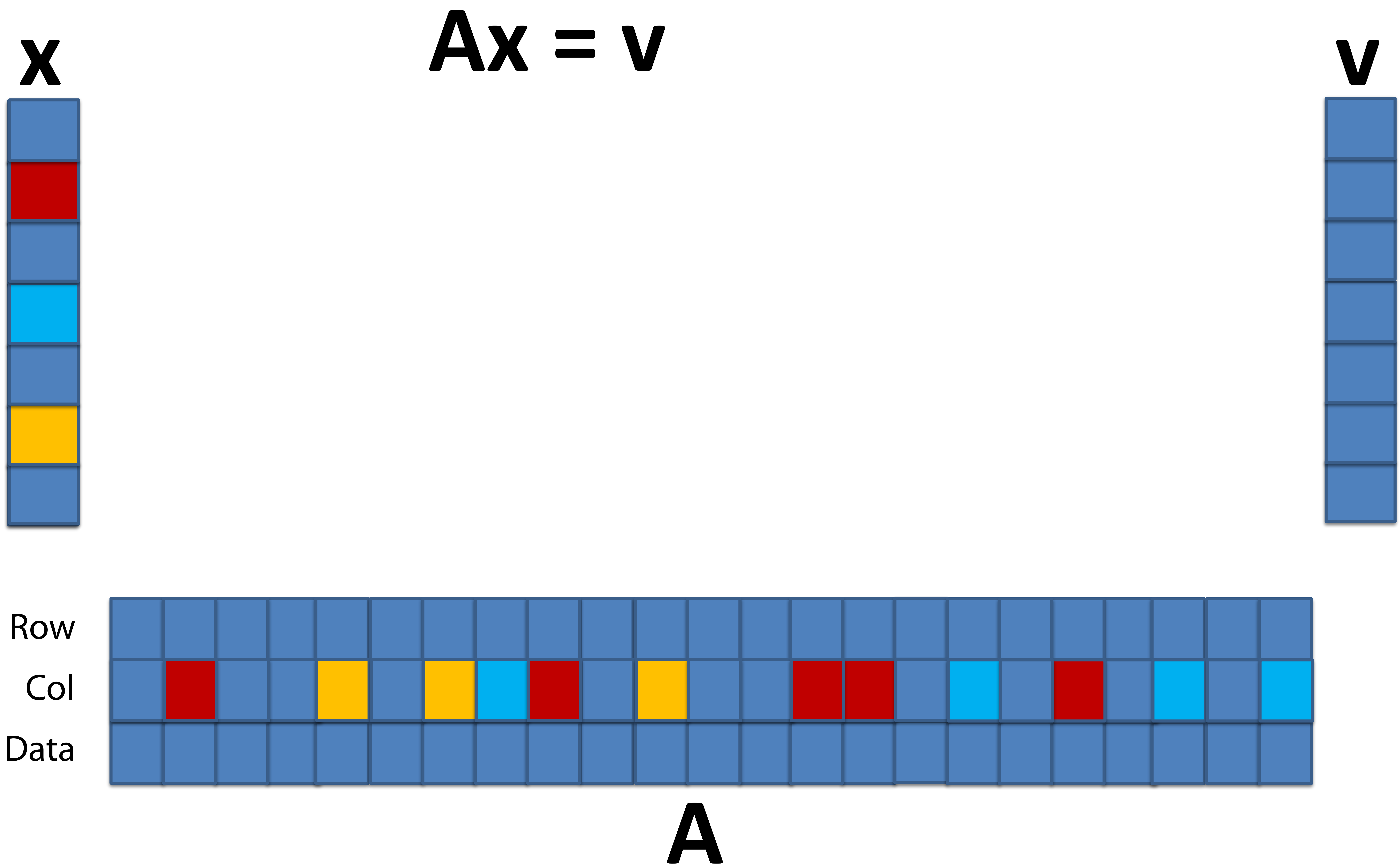
Column indices = { 0, 2, 1, 2, 3, 0, 3 }

Running Example: SpMV

$$Ax = v$$

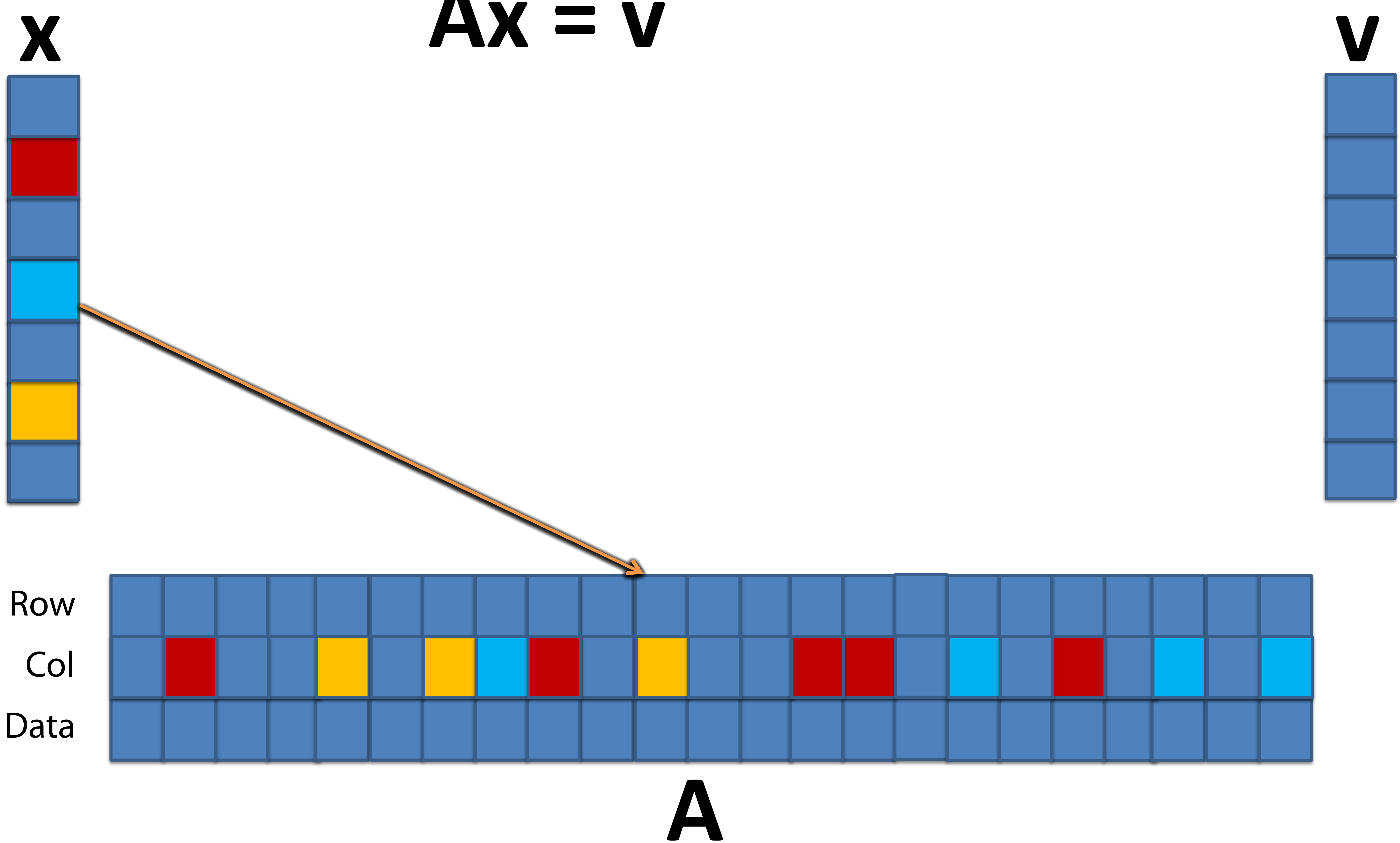


Running Example: SpMV

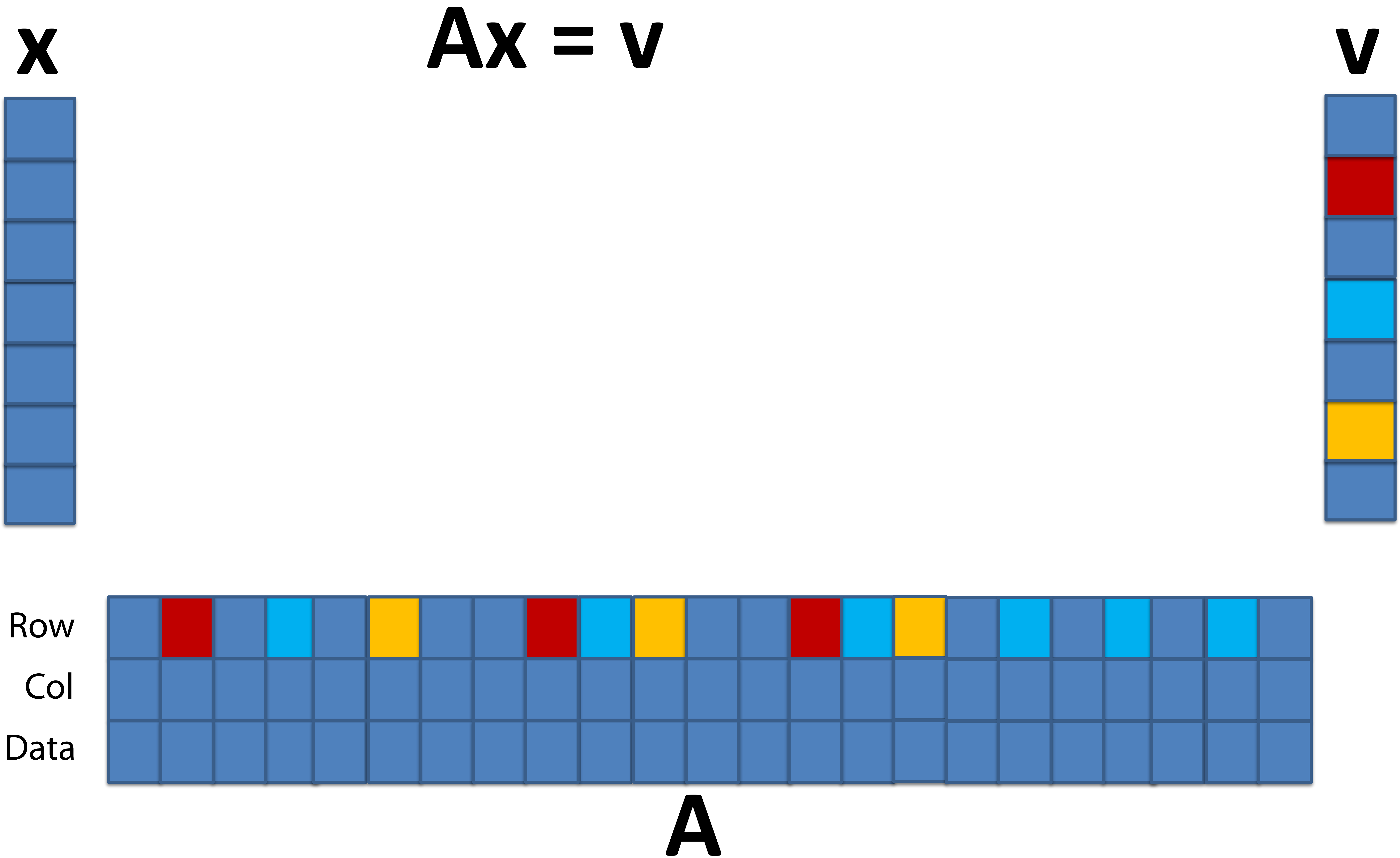


Running Example: SpMV

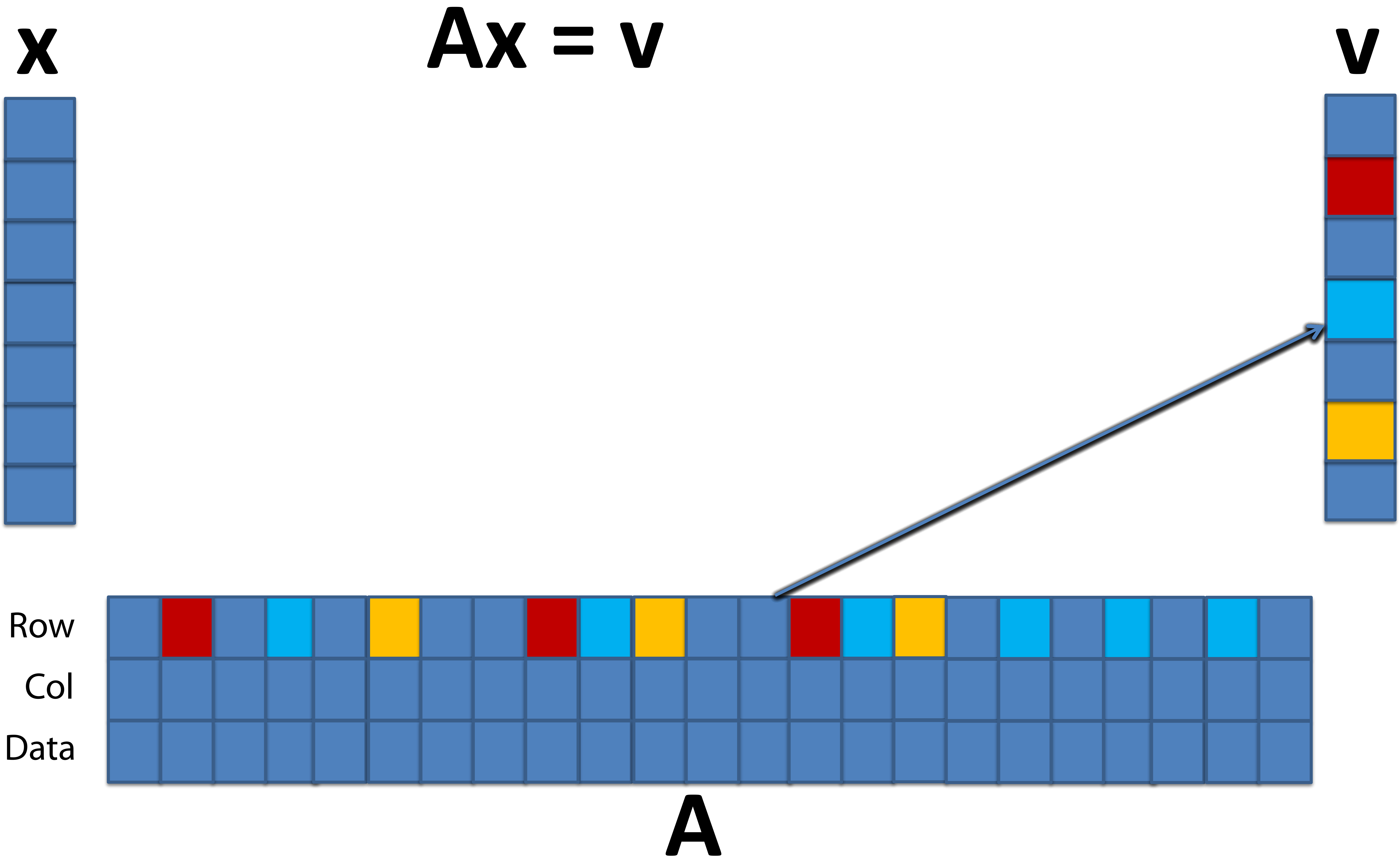
$$Ax = v$$



Running Example: SpMV



Running Example: SpMV

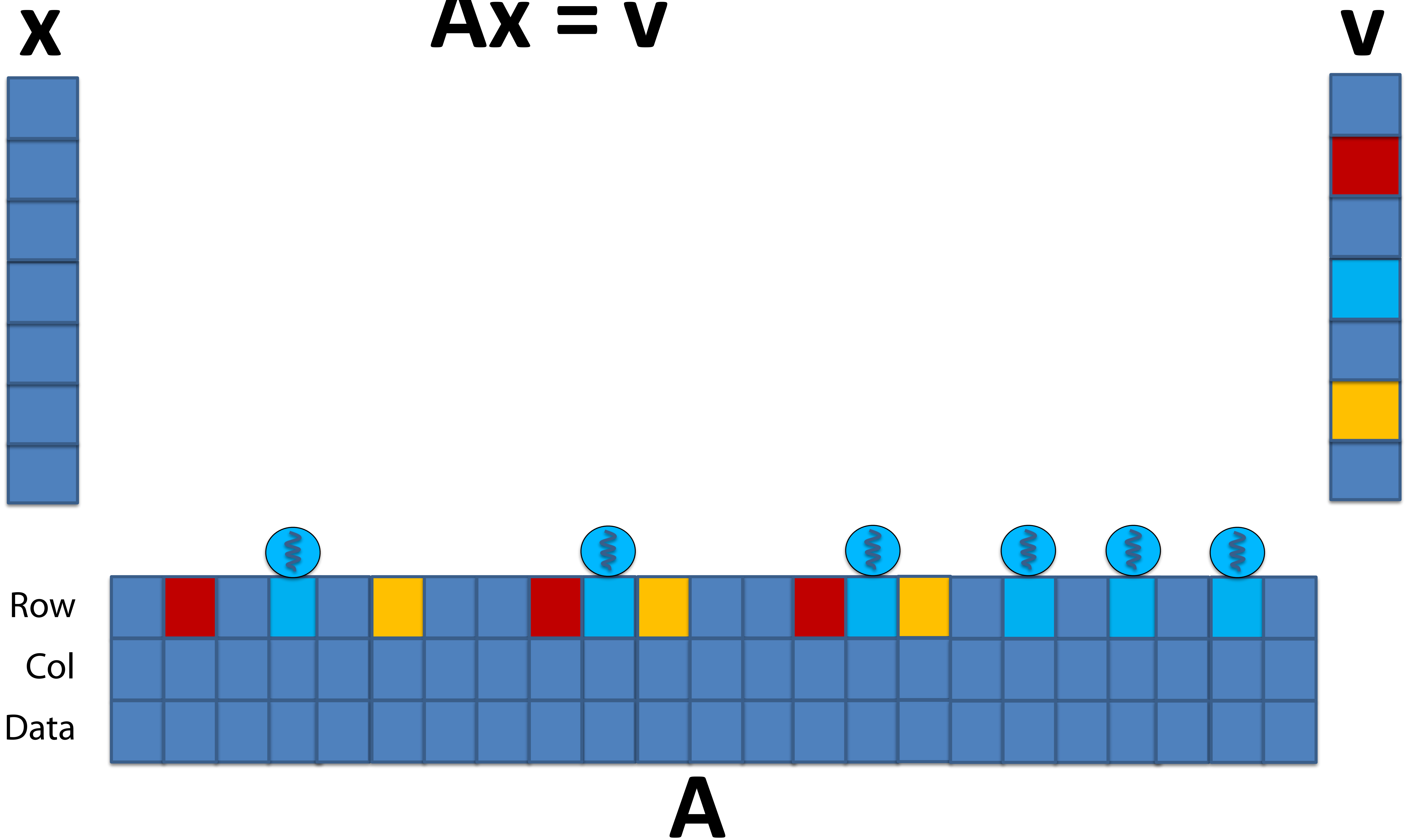


3. “Scatter to Gather” Transformation

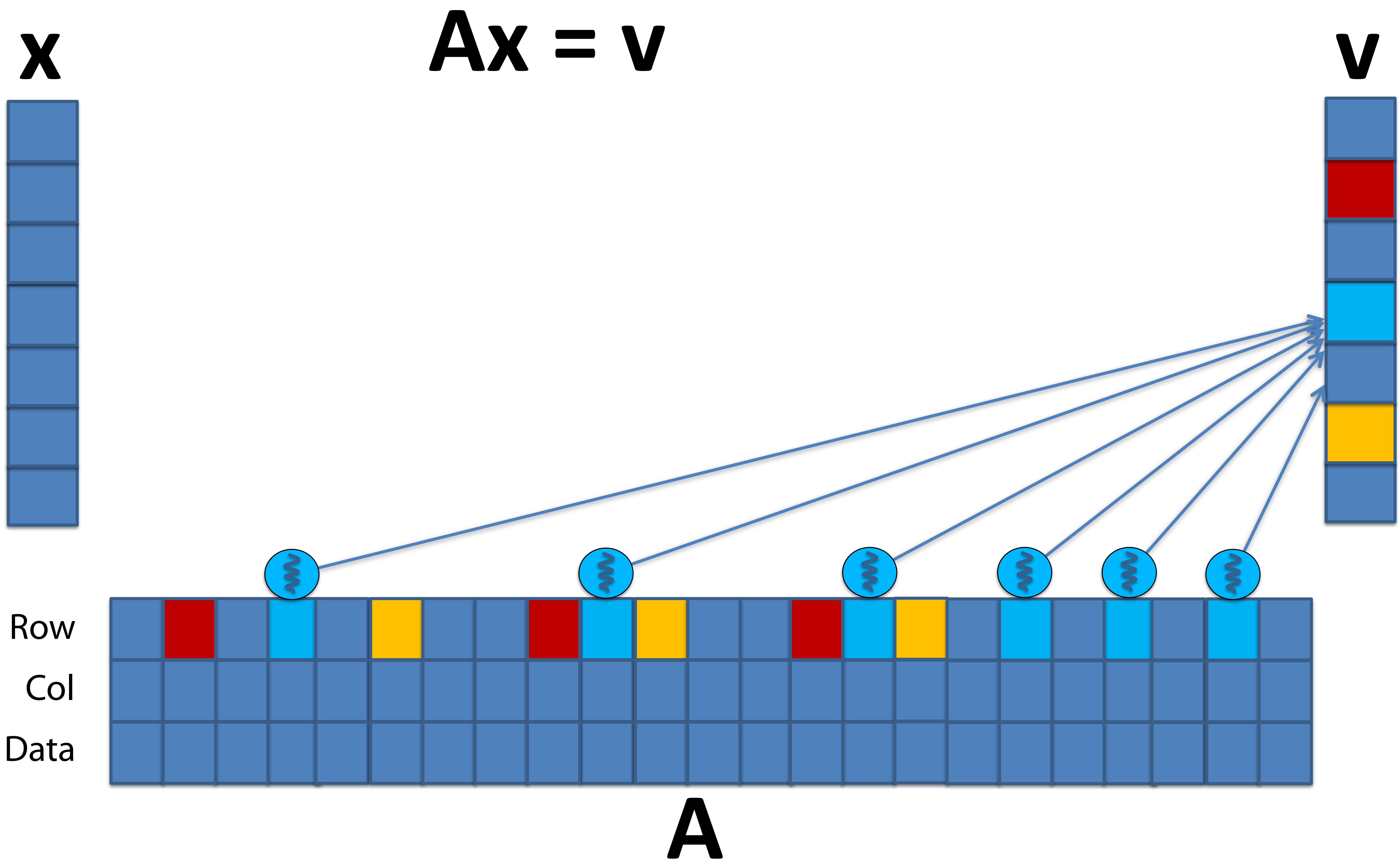
- Write conflicts have to be serialized (atomics)
- Turn overlapping writes into overlapping reads
- Hardware can handle overlapping reads more efficiently

3. “Scatter to Gather” Transformation

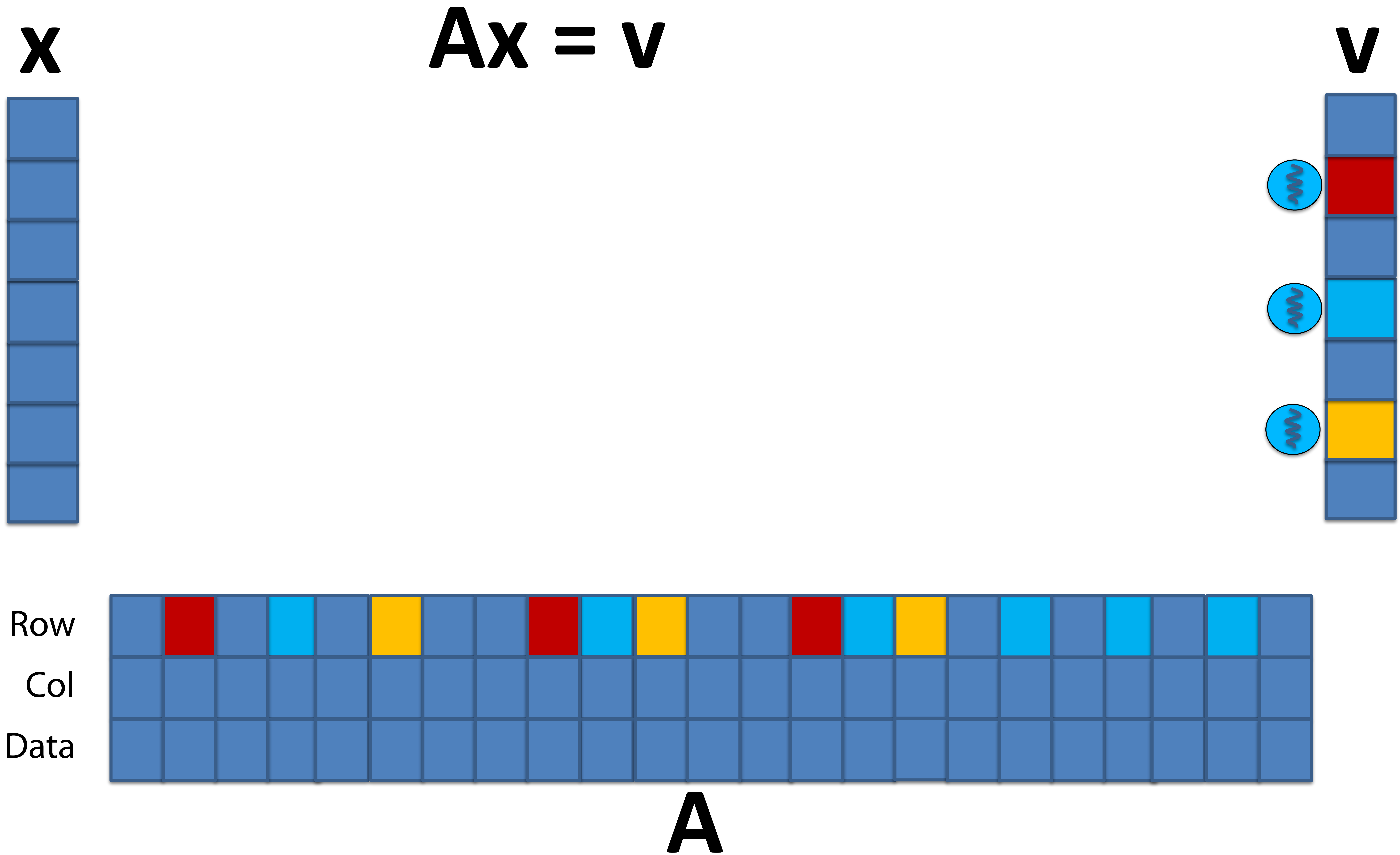
$$Ax = v$$



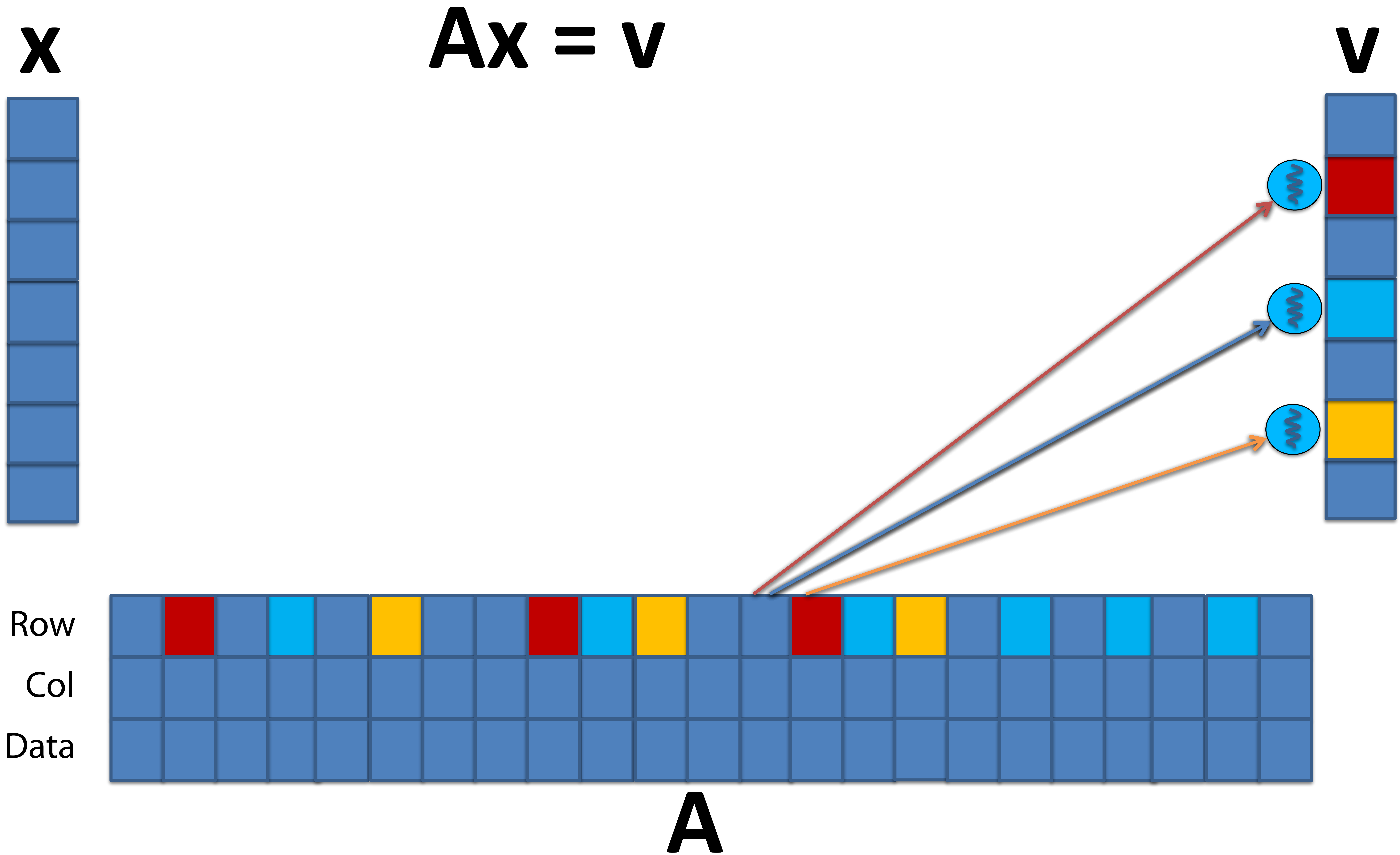
3. "Scatter to Gather" Transformation



3. "Scatter to Gather" Transformation



3. “Scatter to Gather” Transformation



3. "Scatter to Gather" Transformation

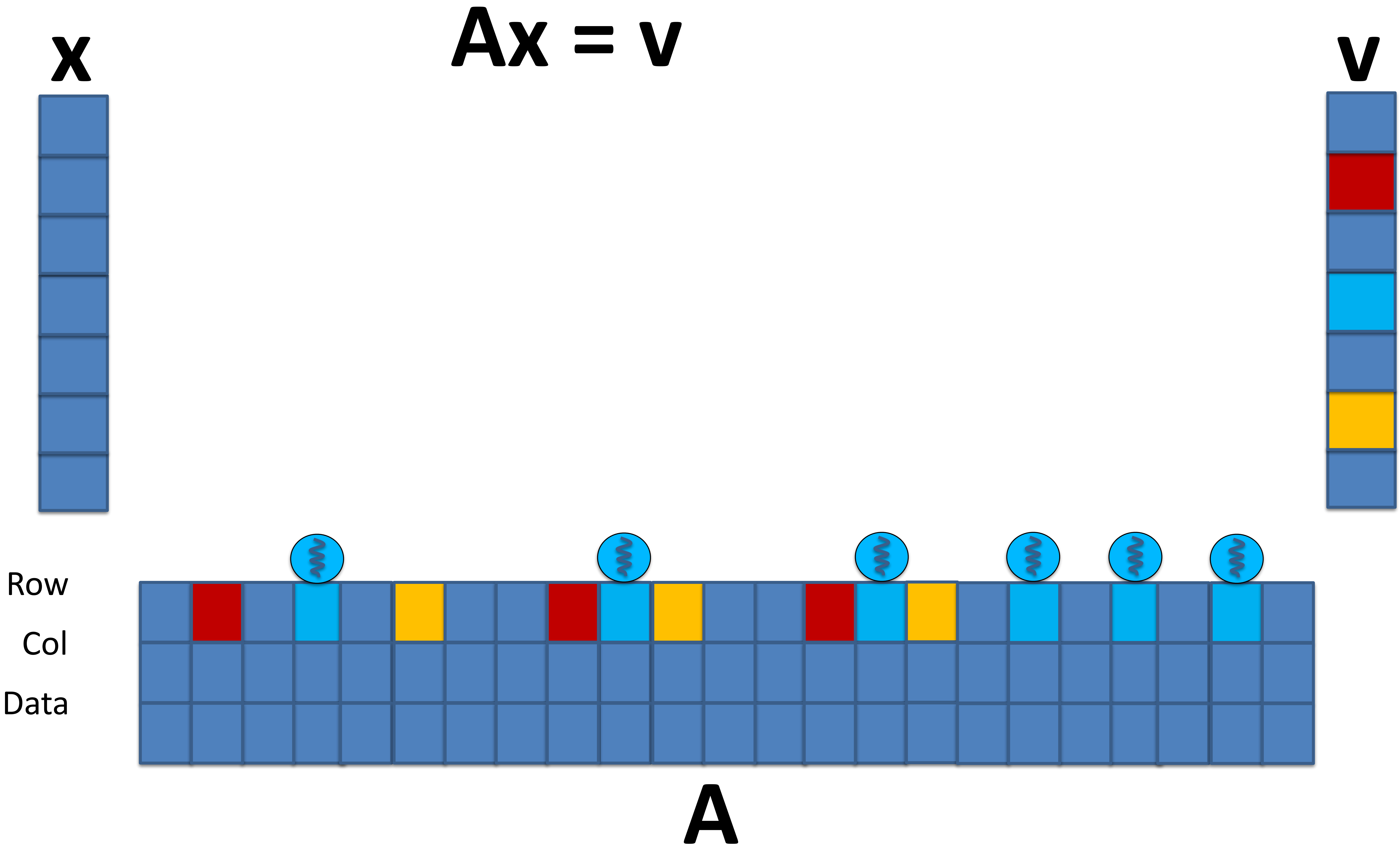
- **Pro: Reduce write contention**
  - **Don't need atomics for every update**

- **Con: Harder to program efficiently**
  - **Harder to get memory coalescing**

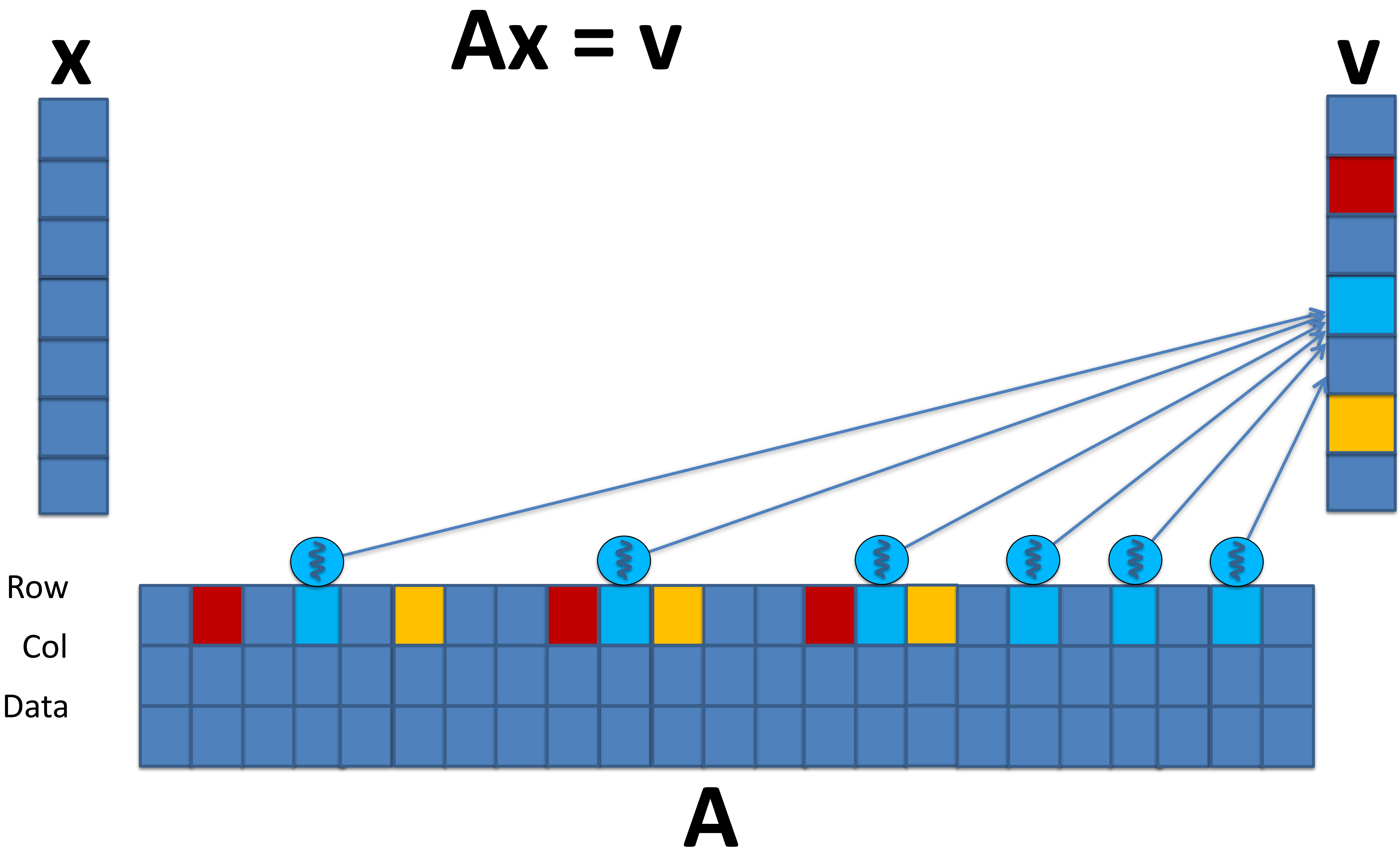
4. Binning

- Usually easy to find scatter address from input
  - E.g., Matrix row number -> output location
- Often difficult to find gather addresses efficiently
  - E.g., For sparse matrices, read a lot of unnecessary data

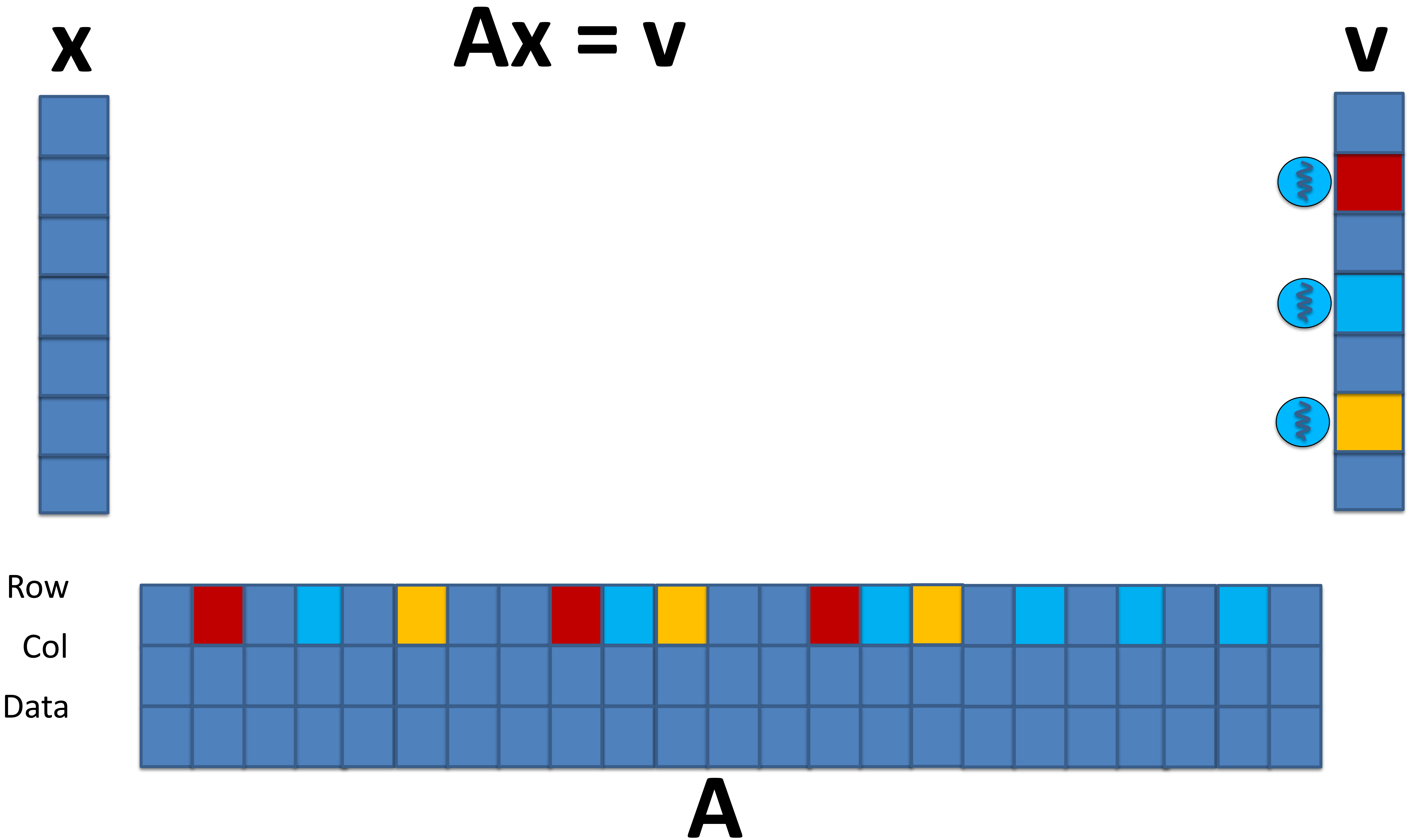
3. "Scatter to Gather" Transformation



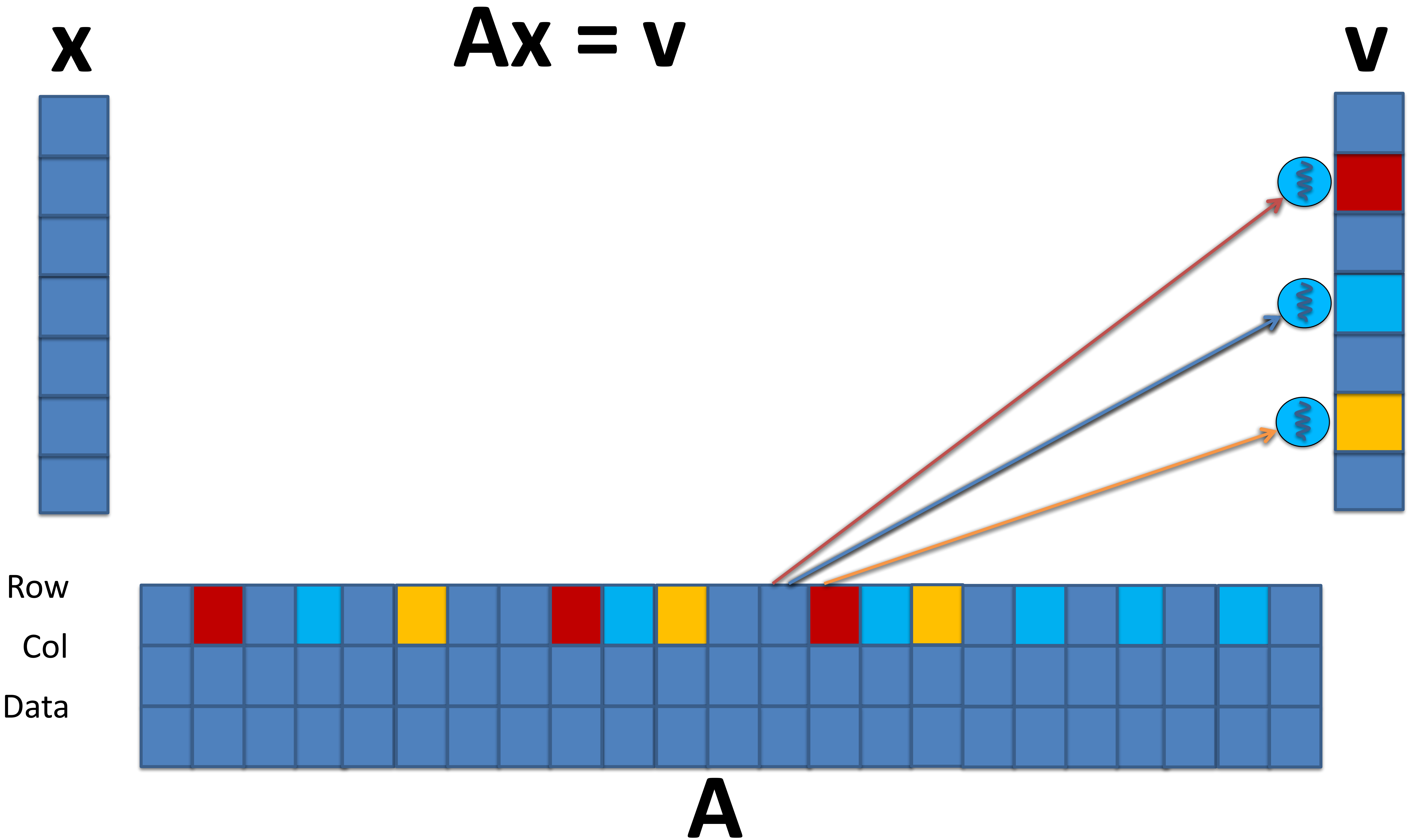
3. "Scatter to Gather" Transformation



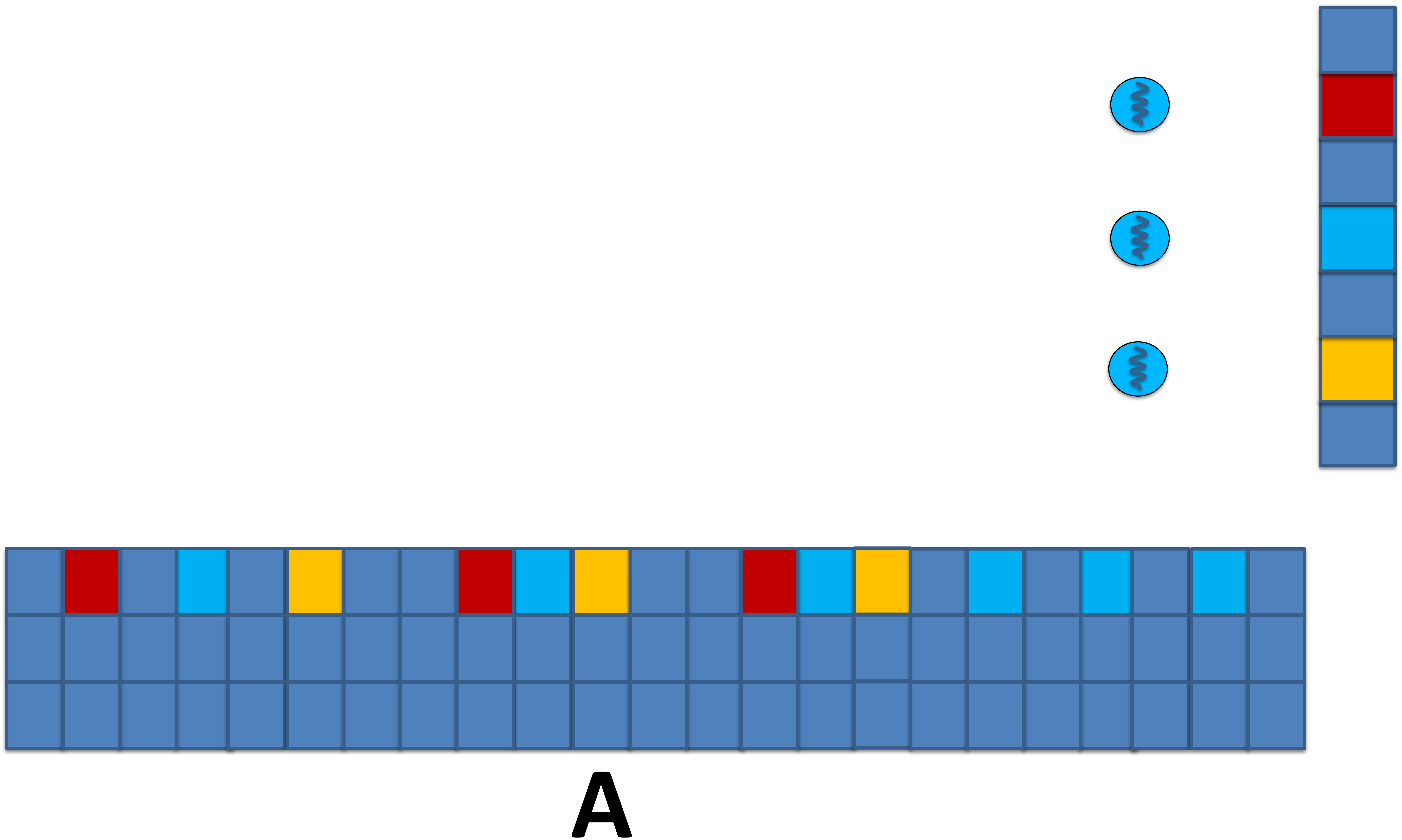
3. "Scatter to Gather" Transformation



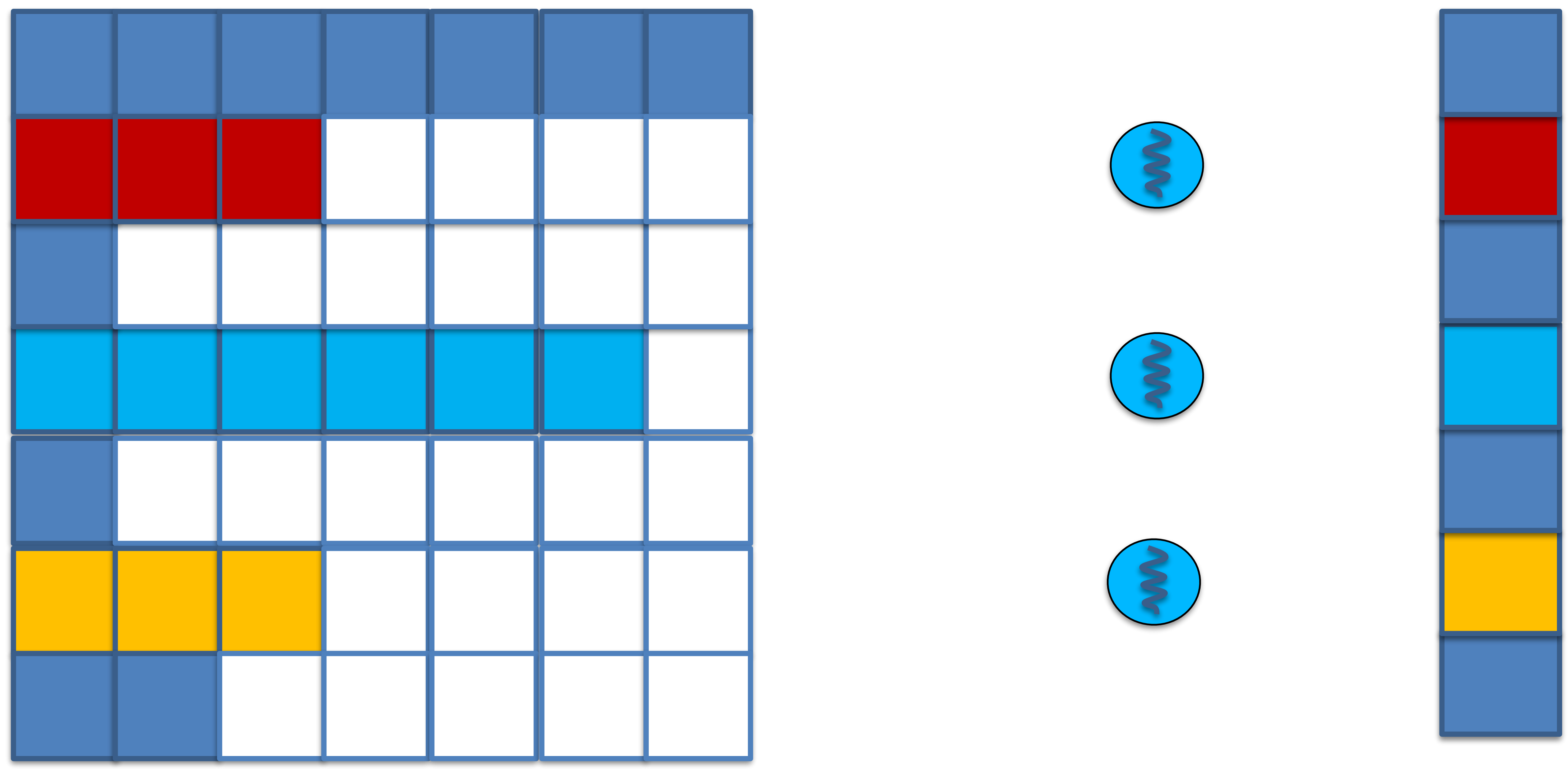
3. "Scatter to Gather" Transformation



4. Binning

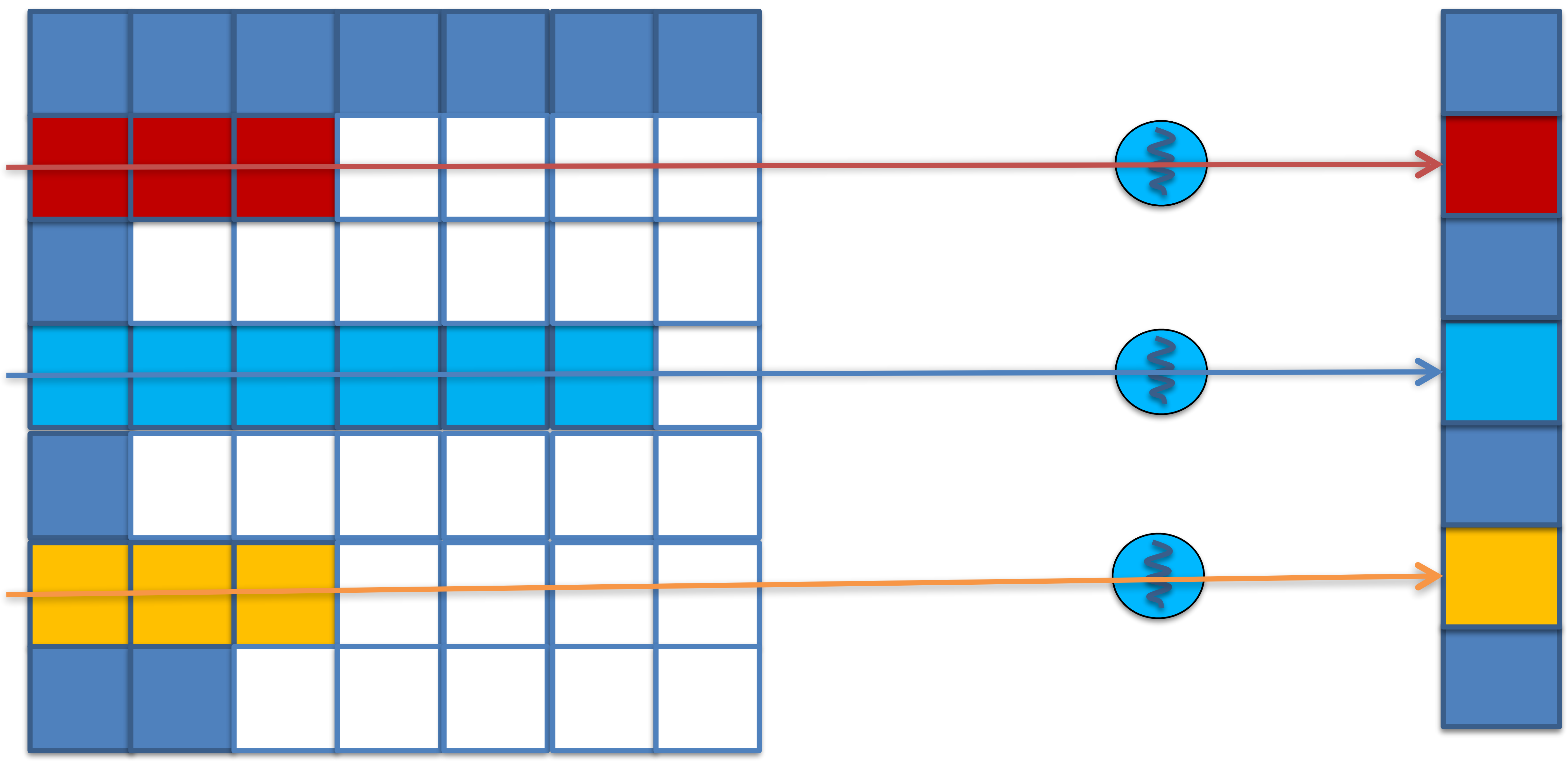


4. Binning



A

4. Binning



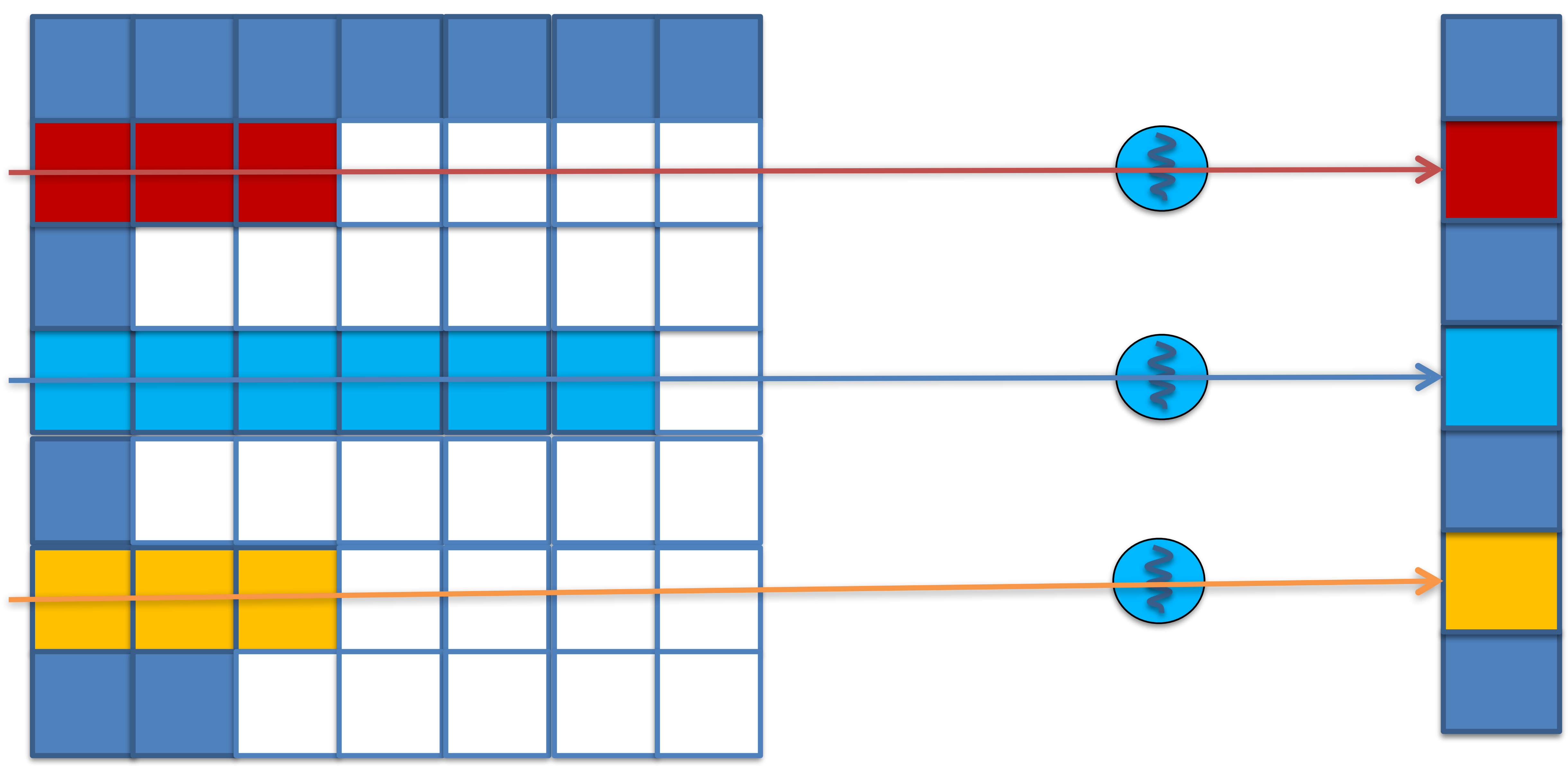
A

4. Binning

- Pro: Organize data for easy access
  - Without accessing unnecessary data
- Con: Large memory requirement
  - Lots of wasted space (How can we fix this?)
- Con: Binning takes time

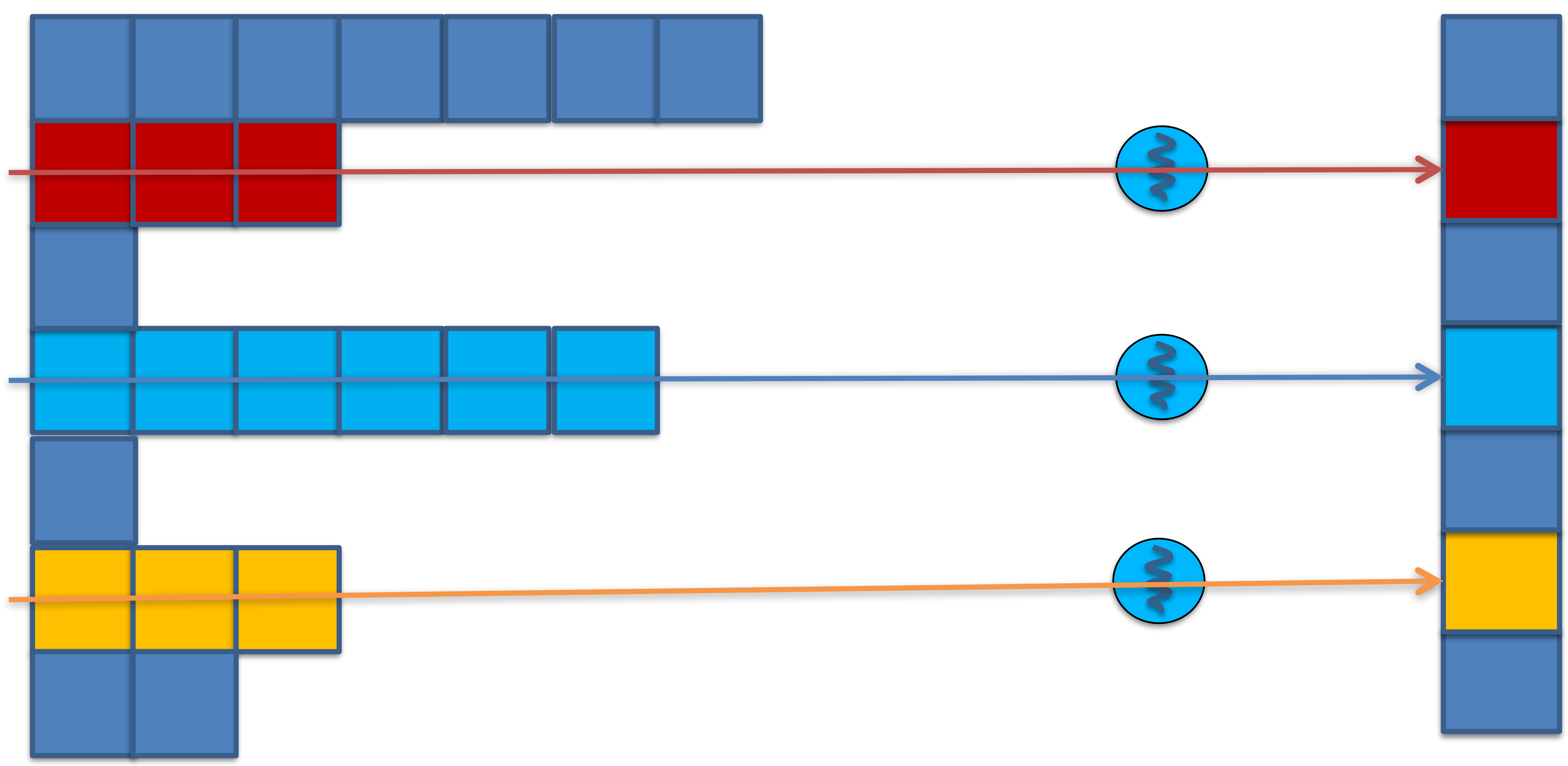
5. Compaction

$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$



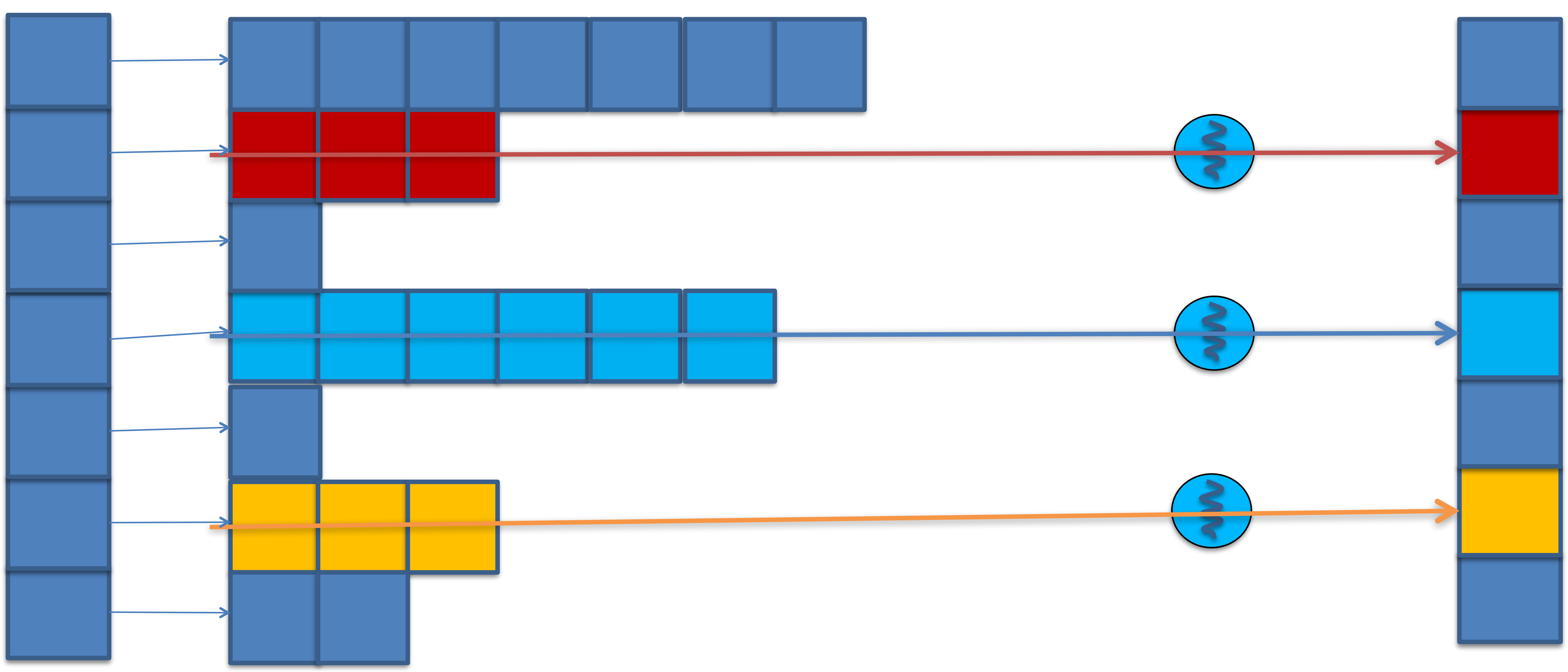
5. Compaction

$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$



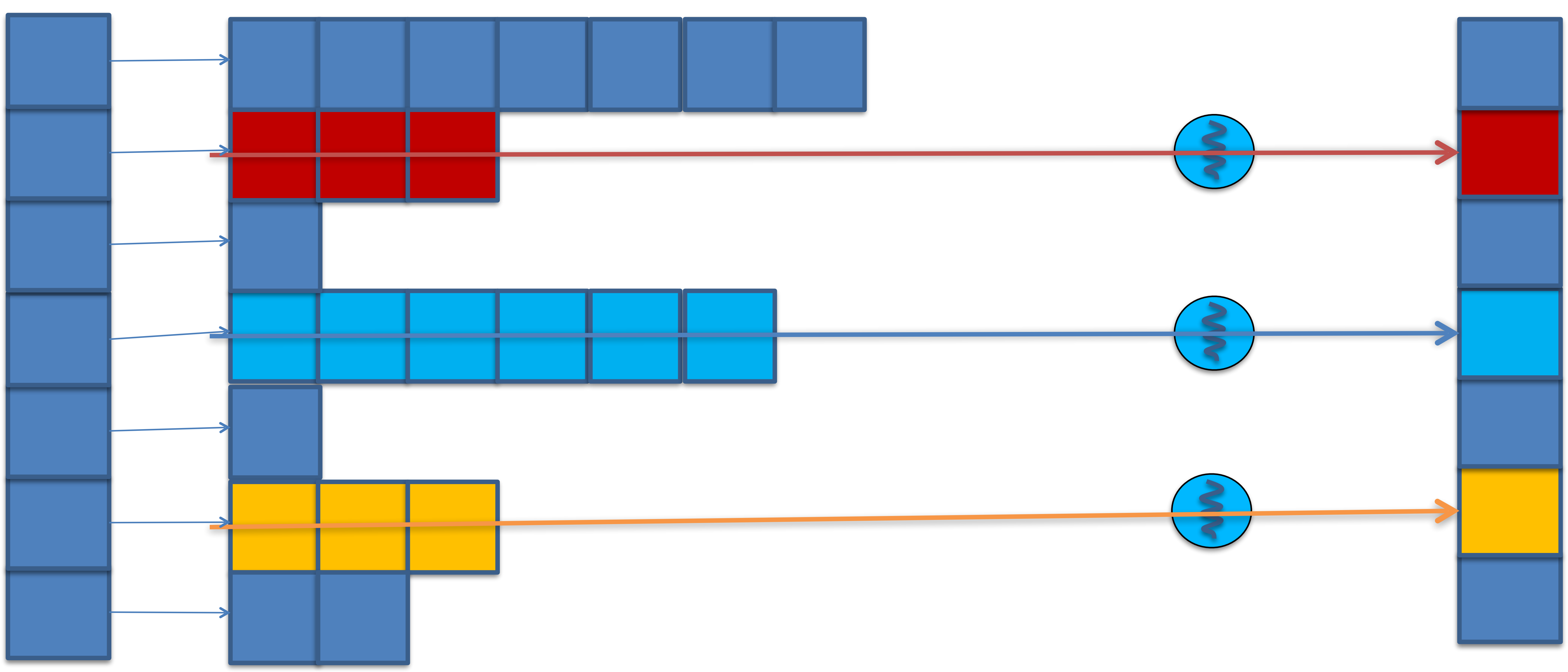
5. Compaction

$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$



5. Compaction

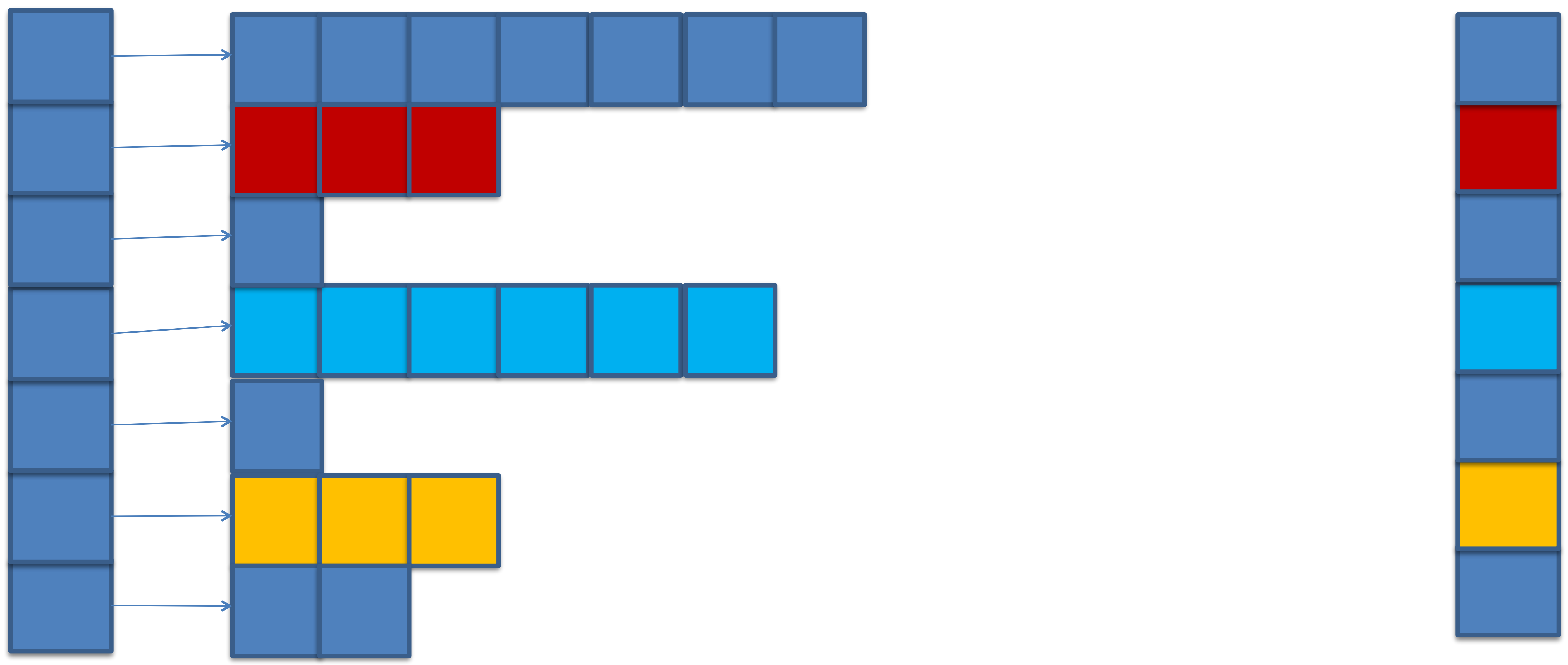
$$\begin{pmatrix} 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$



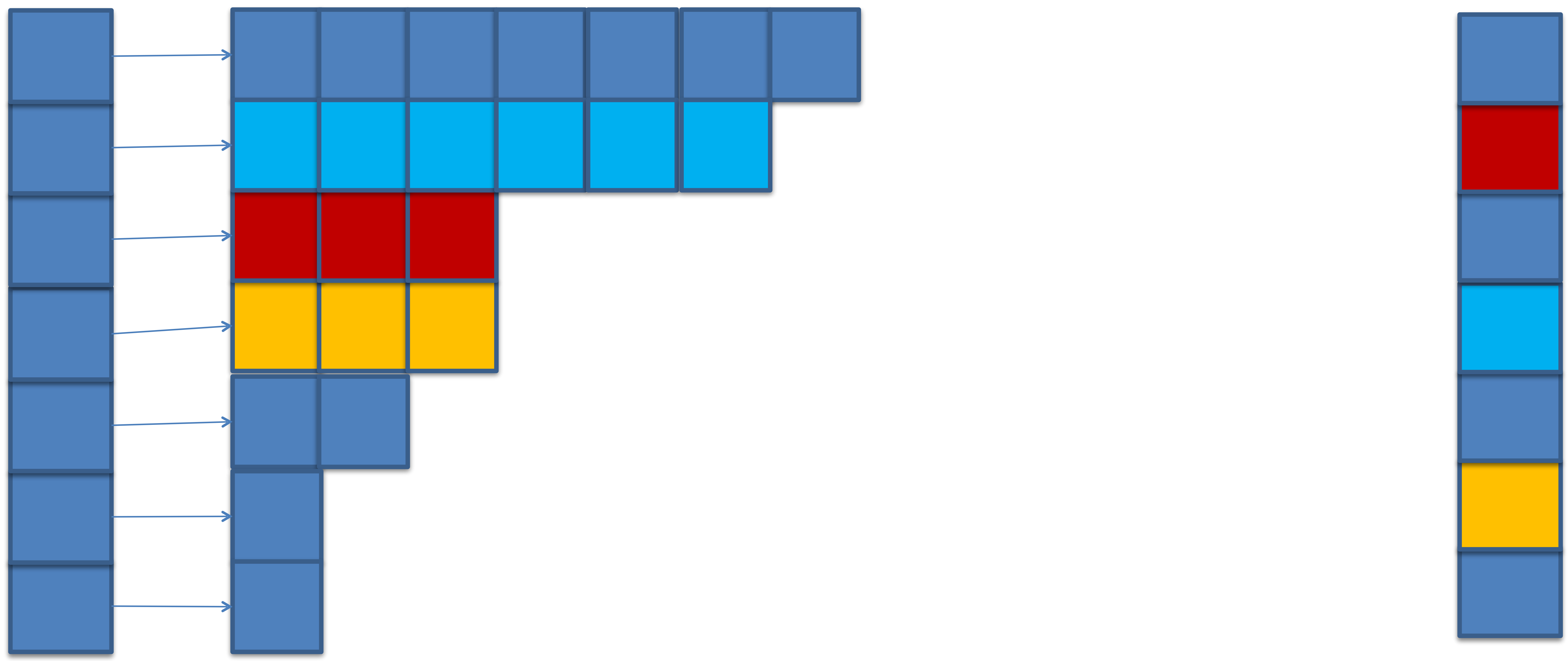
6. Regularization (Load Balancing)

- Some threads have more work than others
  - Poor hardware utilization if we're waiting for a few threads to finish
  - Can't release resources until all threads in block have finished
- Want to balance the work between threads

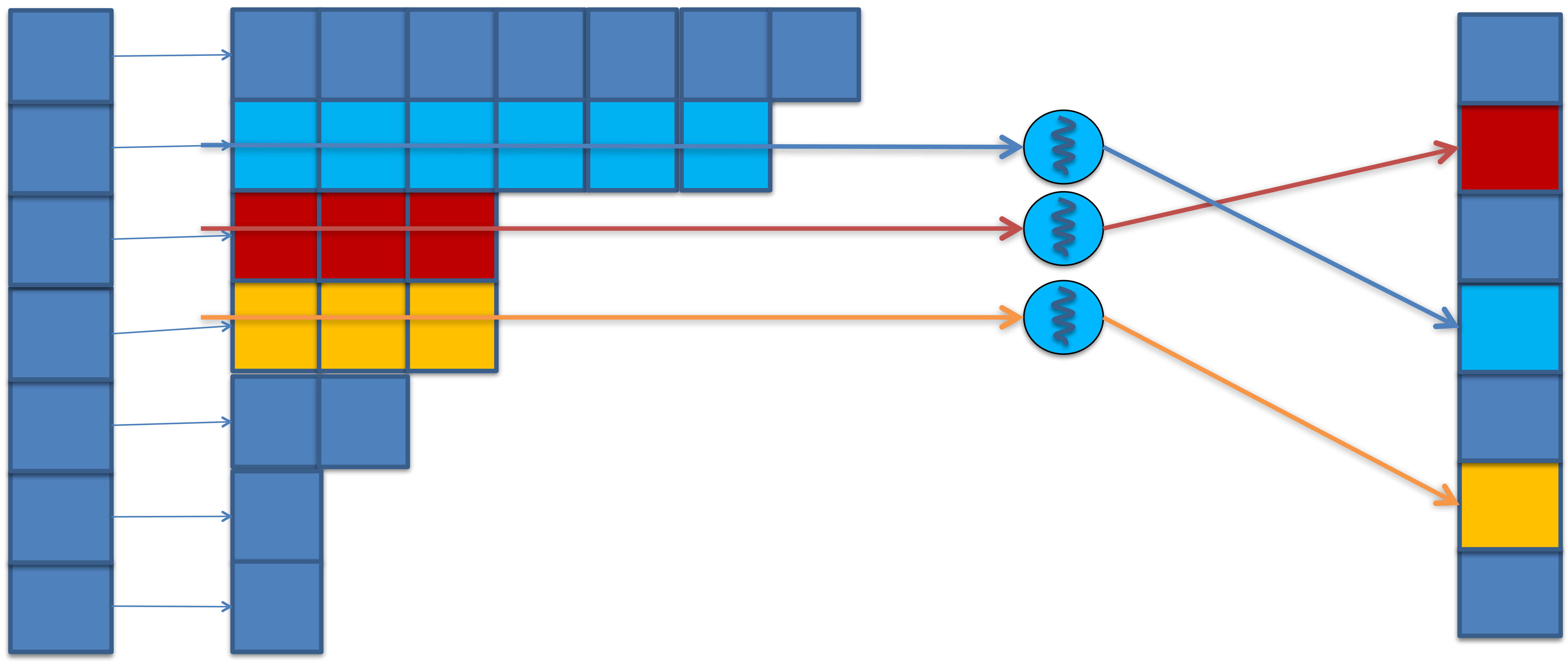
6. Regularization (Load Balancing)



6. Regularization (Load Balancing)



6. Regularization (Load Balancing)



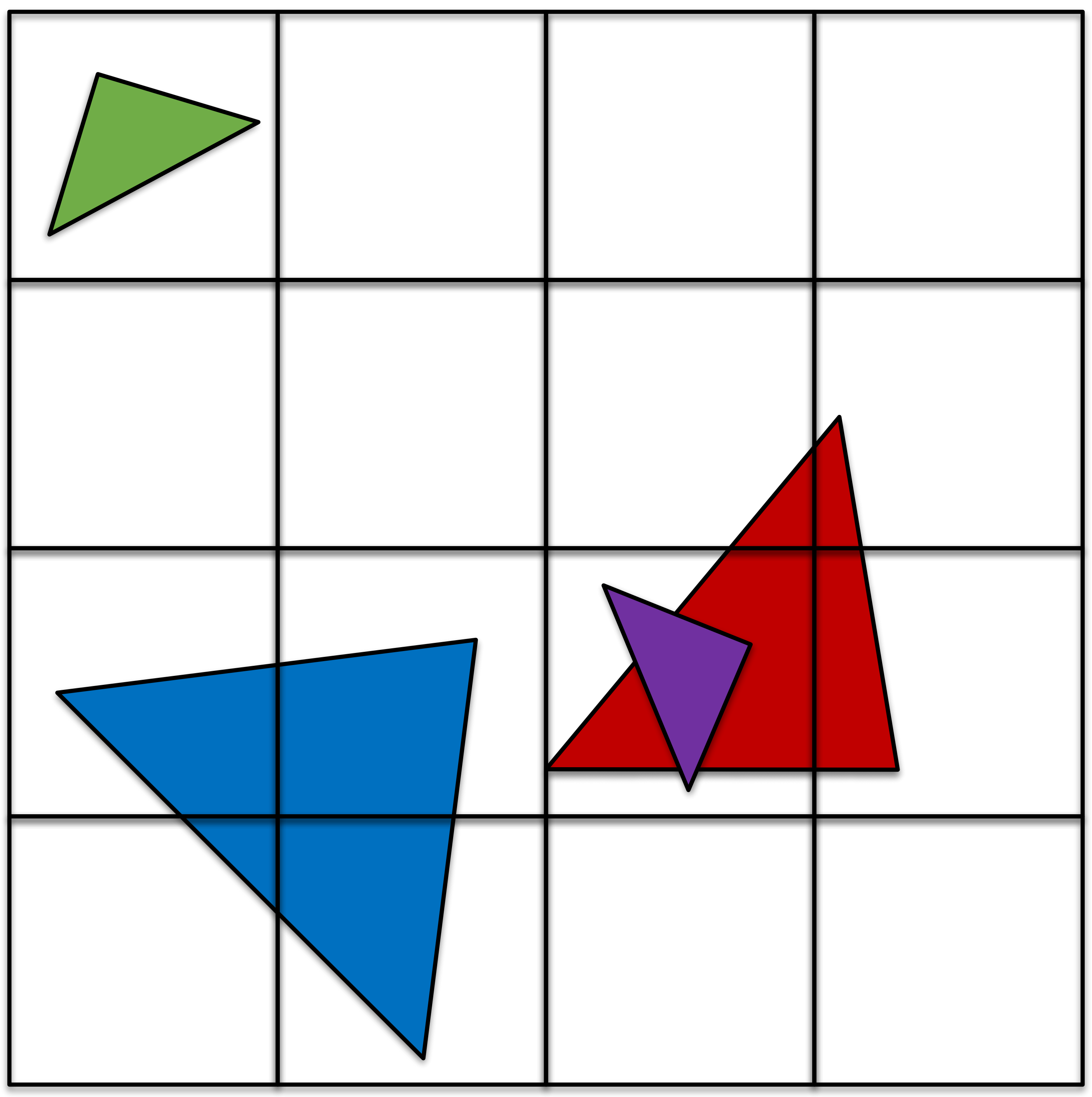
6. Regularization (Load Balancing)

- Pro: Better hardware utilization

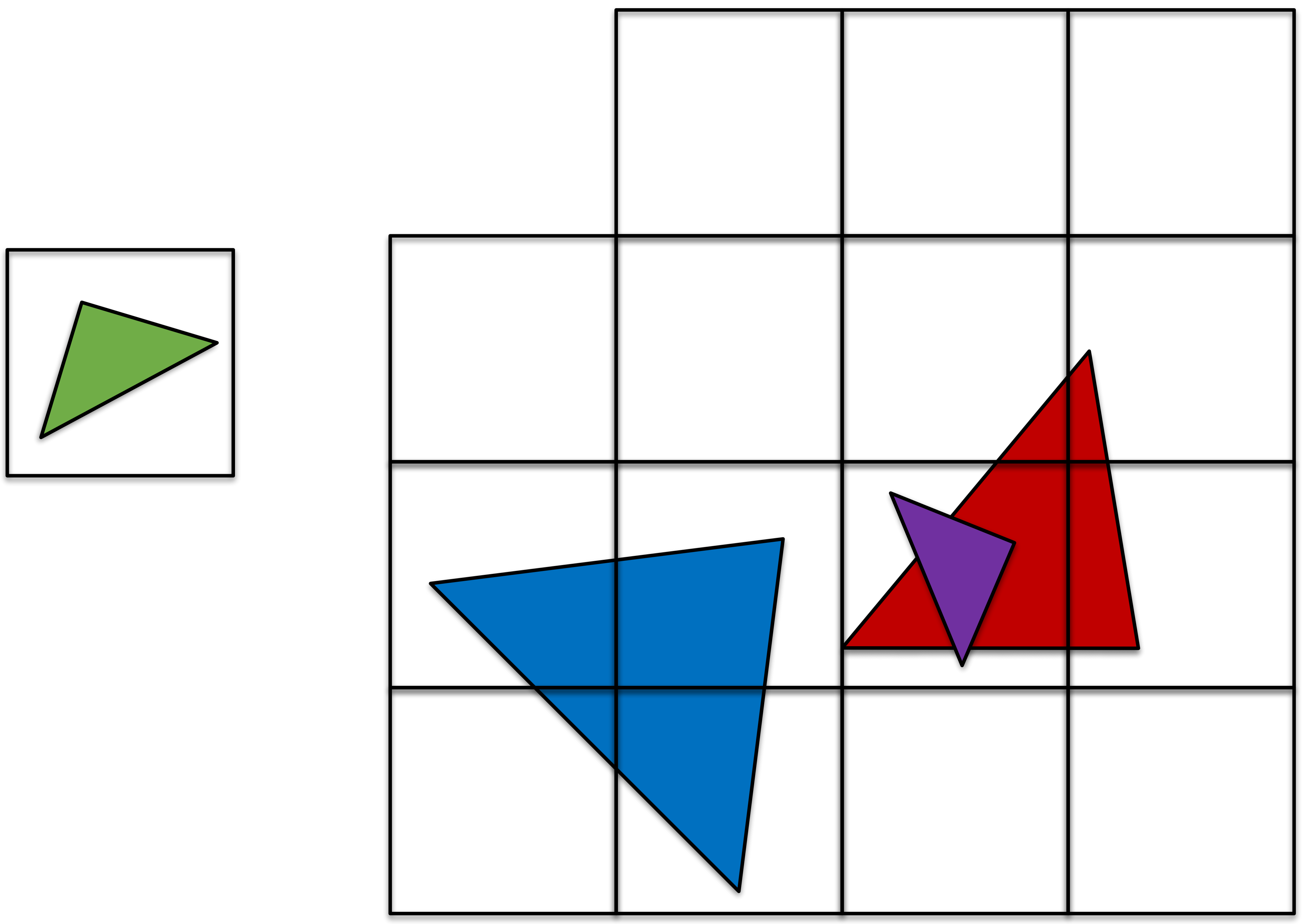
- Con: Some applications difficult to load balance

- Con: Load balancing operations take finite time

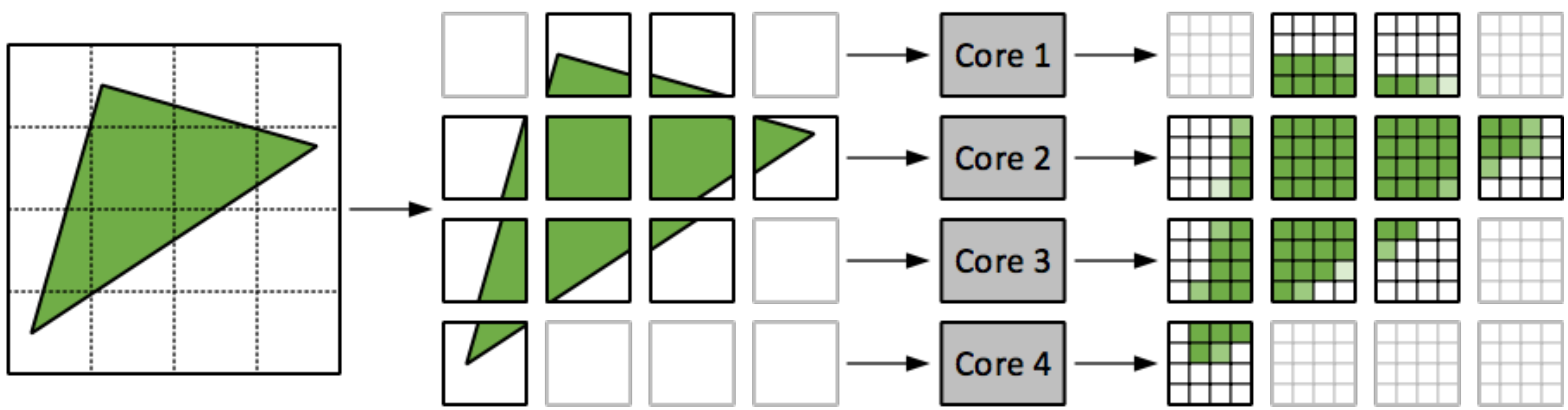
Piko: Spatial Tiling for Parallelism



Piko: Spatial Tiling for Parallelism



Piko: Spatial Tiling for Parallelism

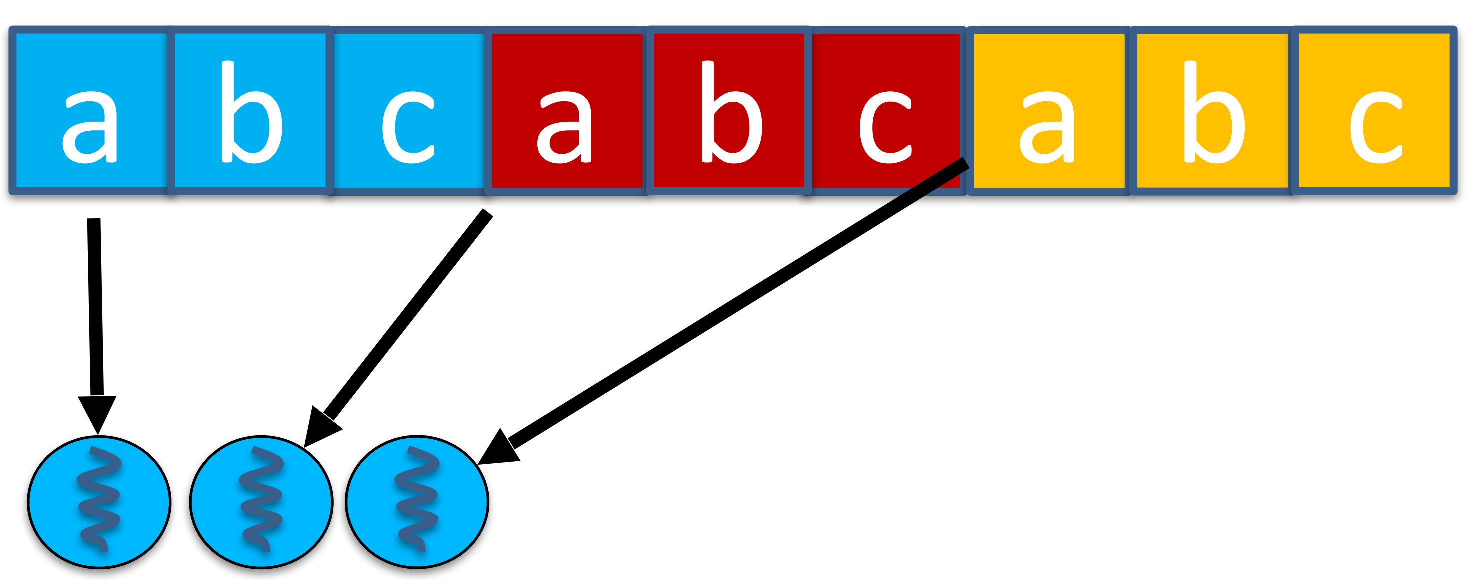


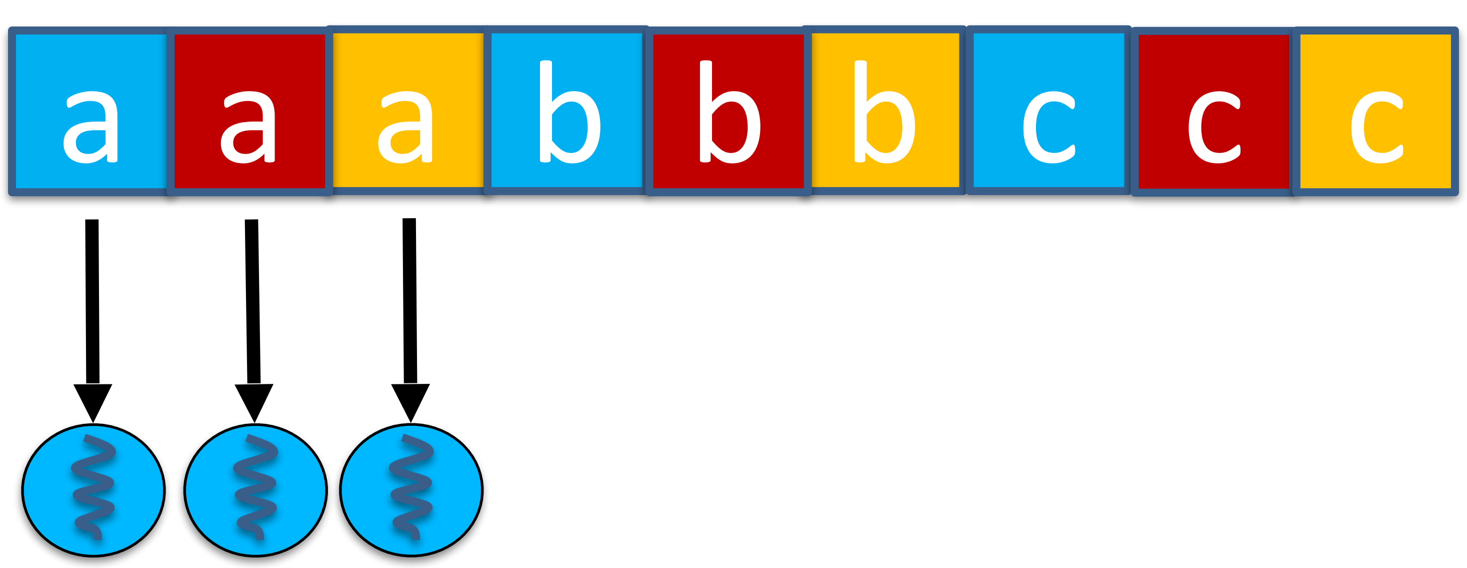
7. Data Layout Transformation

- "Array of Structs" vs. "Struct of Arrays"

```
struct Data {
  float a;
  float b;
  float c;
};
```

```
struct Data {
  float a[];
  float b[];
  float c[];
};
```





7. Data Layout Transformation

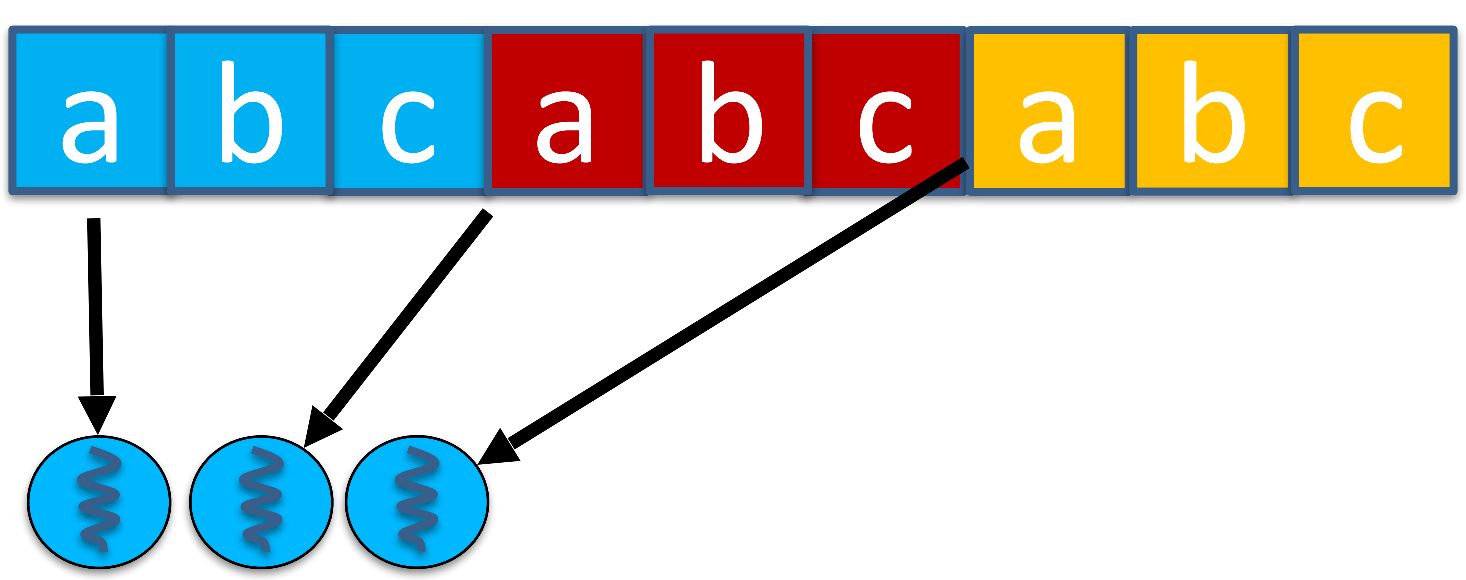
- "Array of Structs" vs. "Struct of Arrays"

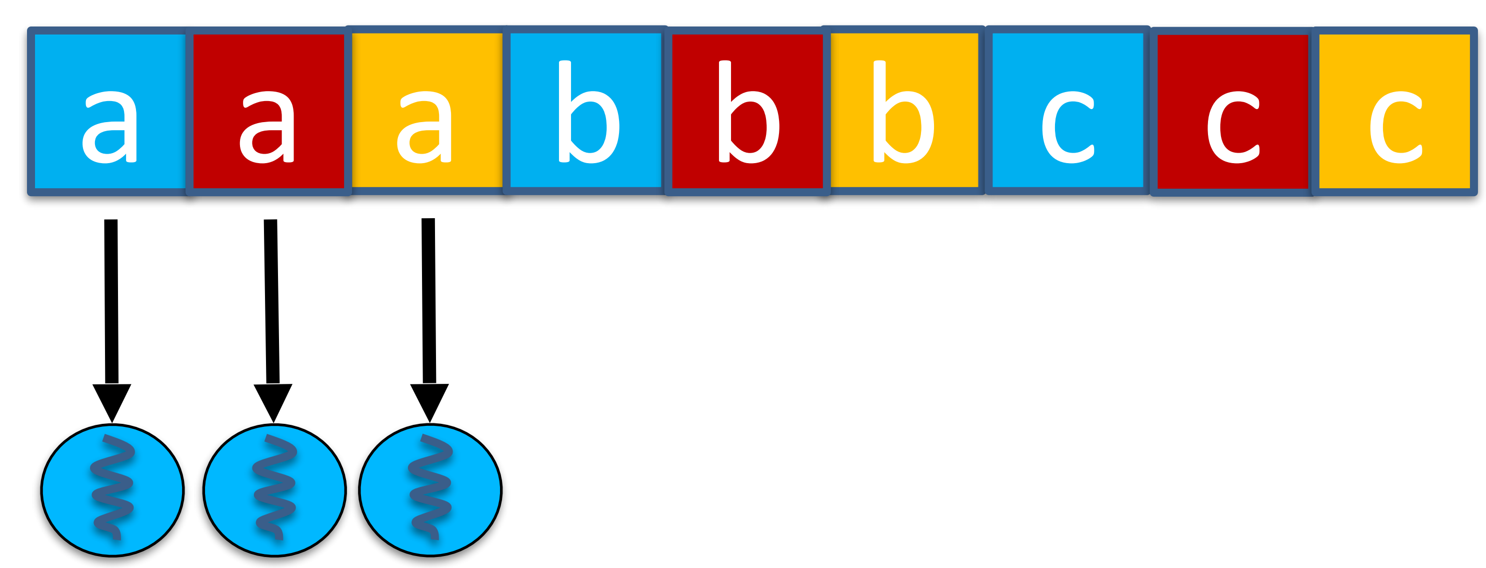
```
struct Data {
  float a;
  float b;
  float c;
};
```

Better for CPU

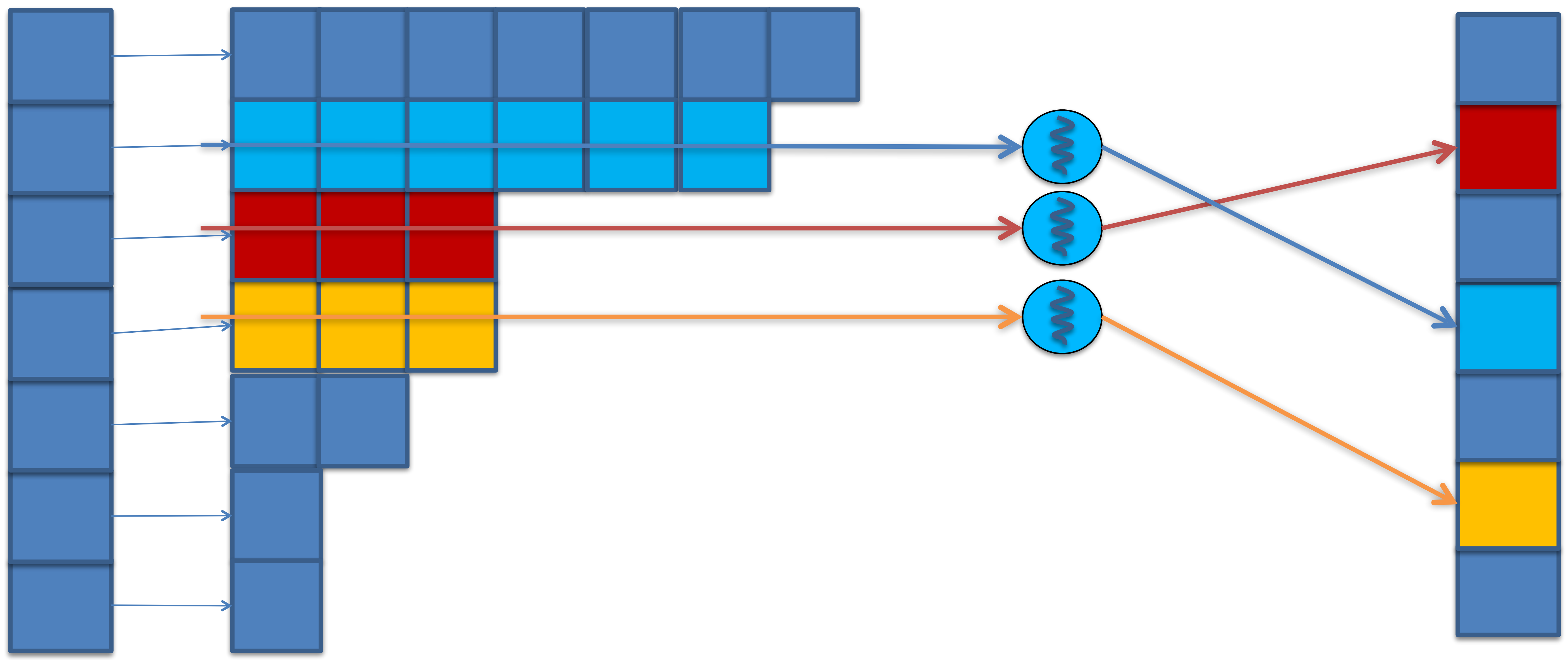
```
struct Data {
  float a[];
  float b[];
  float c[];
};
```

Better for GPU

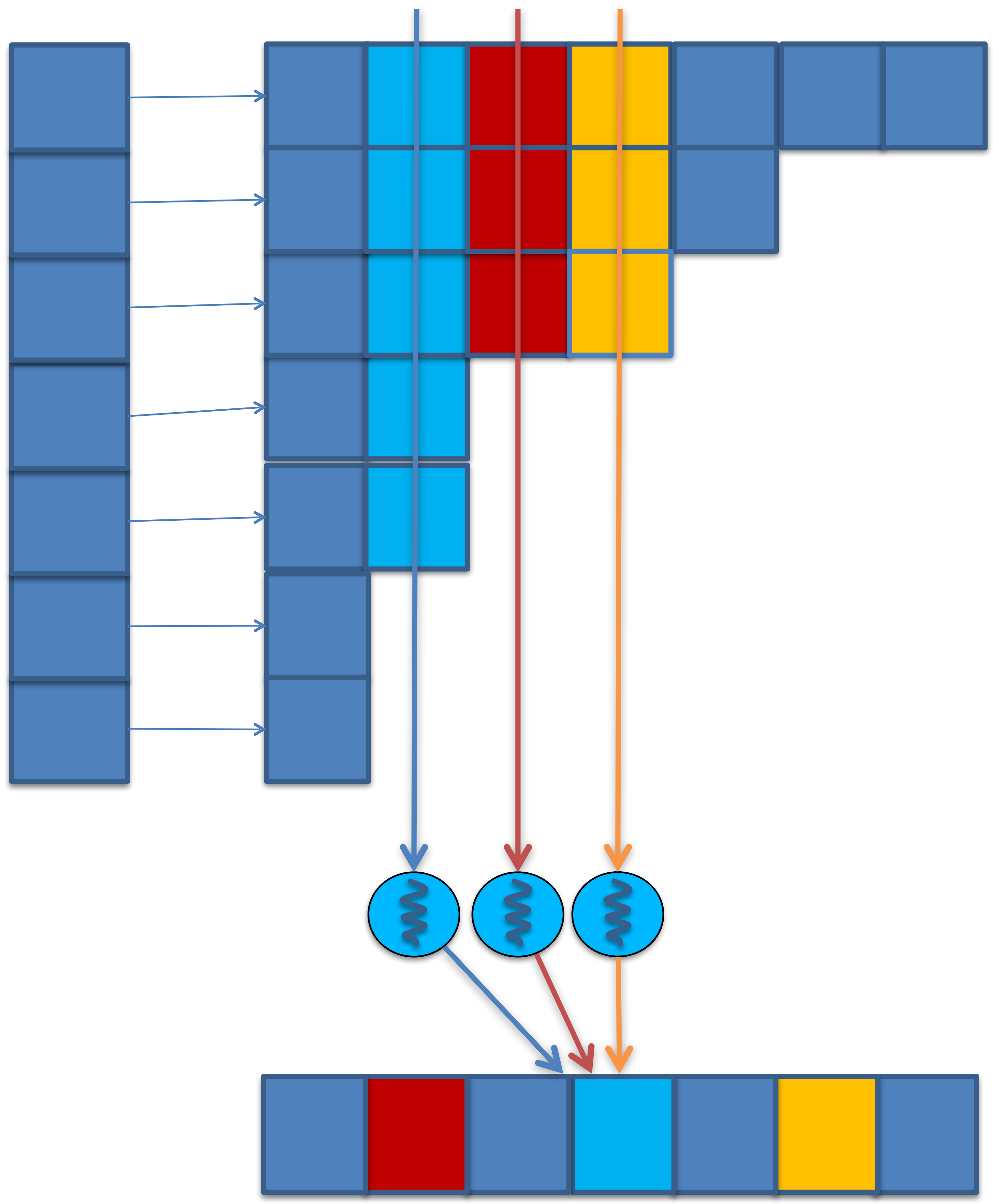




7. Data Layout Transformation



7. Data Layout Transformation



7. Data Layout Transformation

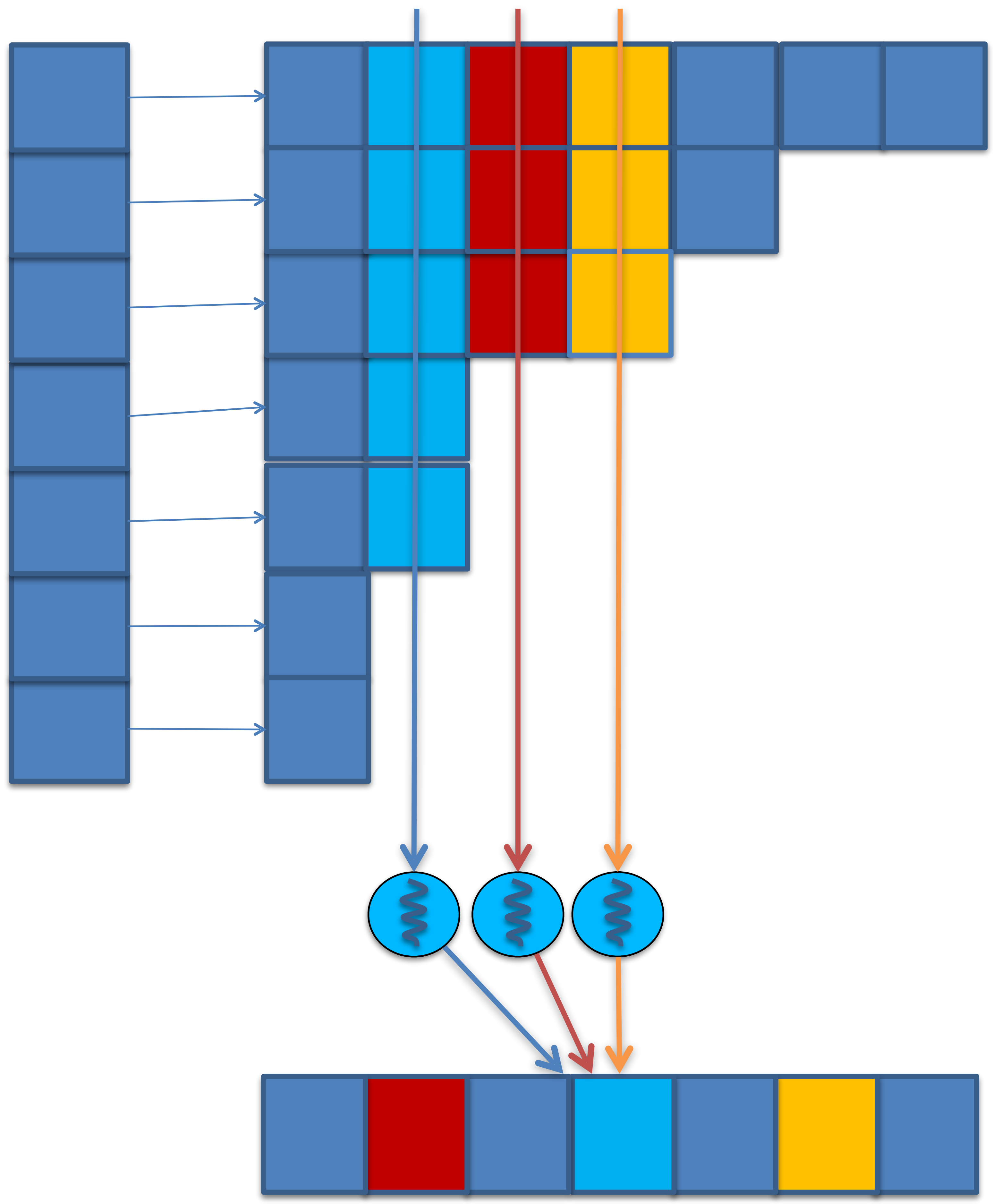
- Pro: Better memory access patterns for GPU

- Con: Usually requires reorganizing data (takes time)
  - Beautiful piece of work from NVIDIA called “Trove” that does a matrix transpose on the fly
  - Many pieces of work that discuss AOS<->SOA

8. Granularity Coarsening

- Parallel execution often requires redundant work
- Let each thread process >1 element to reduce redundancy

8. Granularity Coarsening

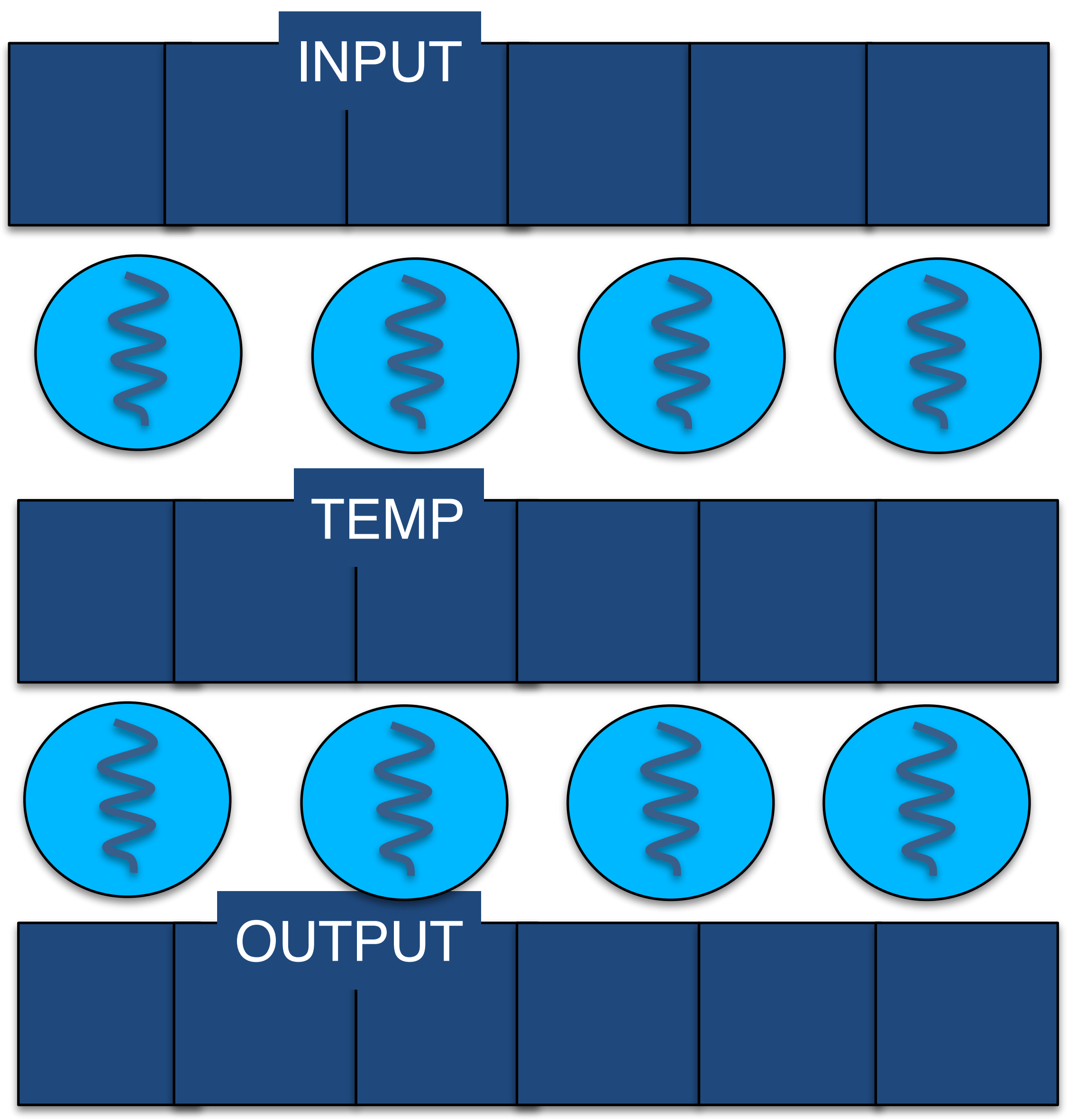


8. Granularity Coarsening

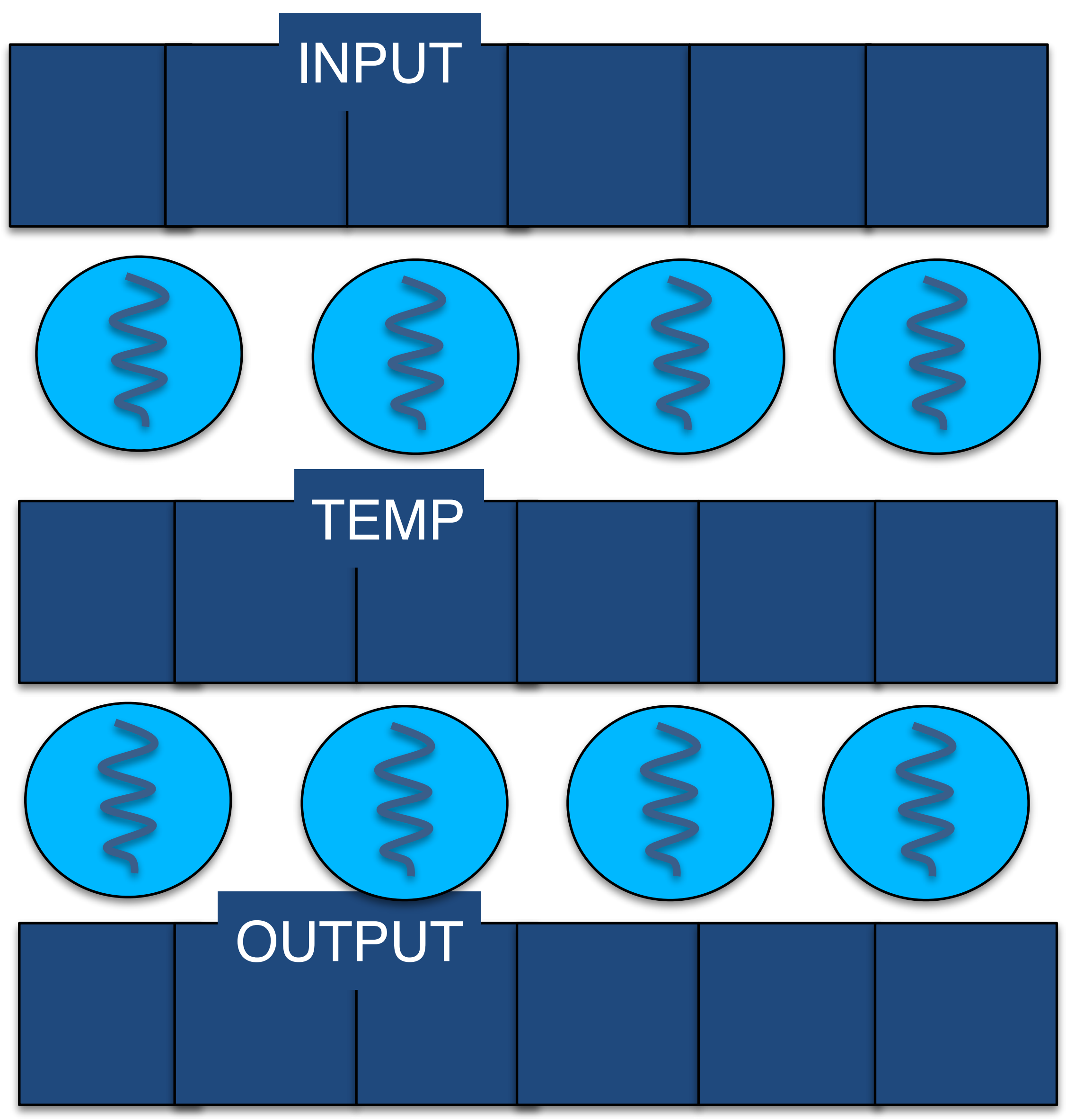
- Pro: Reduces redundant computation
- Con: Reduces parallelism
  - And, thus, latency hiding potential

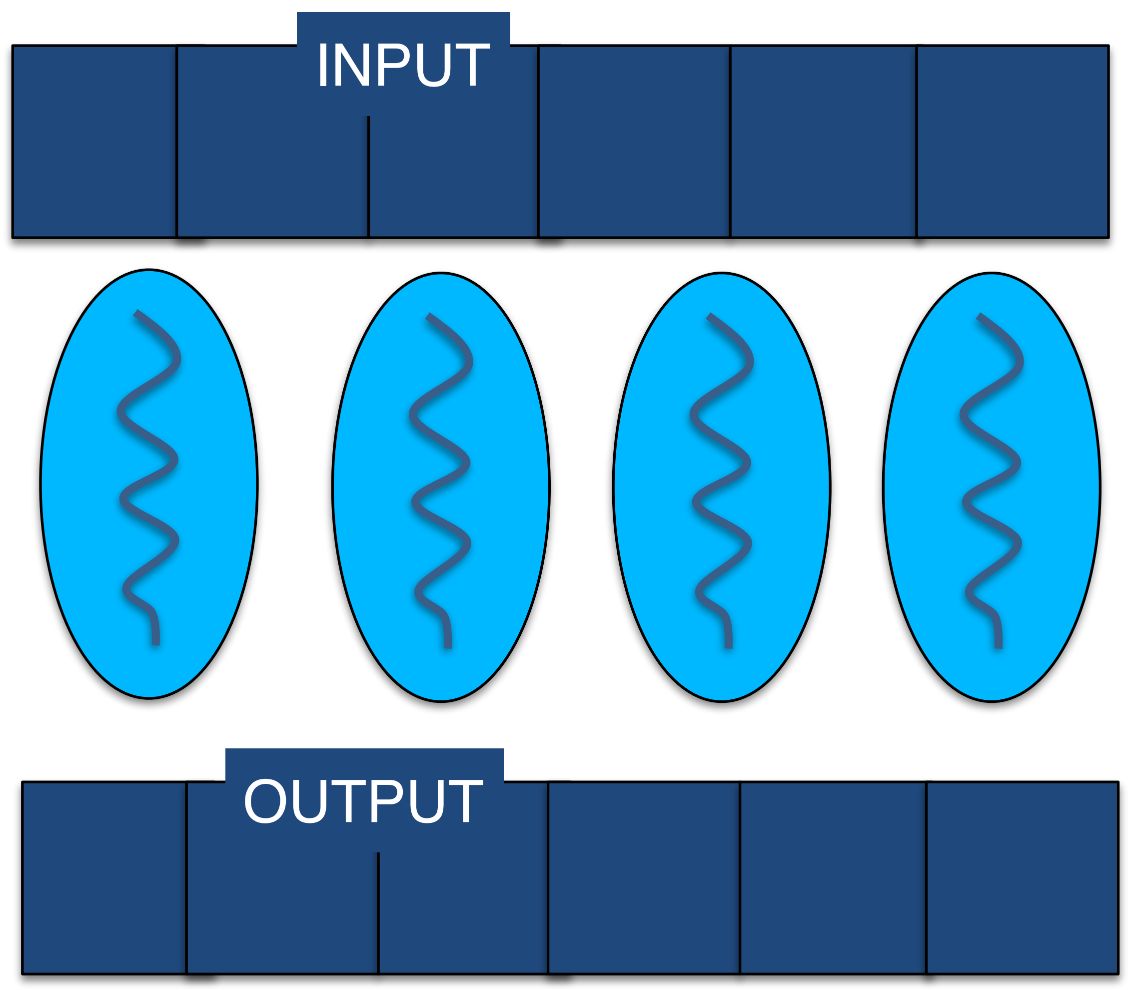
9. Kernel Fusion

9. Kernel Fusion



9. Kernel Fusion





9. Kernel Fusion

- Pro: Removes unnecessary reads/writes
- Con: Might lead to load imbalance
  - (when different threads generate different amounts of intermediate data/work)

Optimization Summary

- (Input) Data Access Tiling
- (Output) Privatization
- "Scatter to Gather" Transformation
- Binning
- Compaction
- Regularization (Load Balancing)
- Data Layout Transformation
- Granularity Coarsening
- Kernel Fusion