COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

PhD Thesis Defense

Analytical Query Execution Optimized for all Layers of Modern Hardware

Orestis Polychroniou

Ορέστης Πολυχρονίου

Big Data

- **Volume** and **value** of (big) data
  - 20 **zettabytes** of data by 2020
  - $125 **billion** in 2015
  - $40 **billion** for databases

- Relational Analytics
  - Business Intelligence
  - Decision Support
  - > $10 **billion** market

Database Systems

- Disk-based / Traditional DBMS
  - Data on (hard) **disk**
  - Query execution **disk-bound**
  - **Not** very distributed (e.g. Oracle)

new hardware **!**

- In-Memory / "**Modern**" DBMS
  - Data (mostly) in **RAM**
  - Query execution **memory-bound**
  - **Very** distributed (e.g. cloud)

Impact of Hardware

- Traditional —> Modern DBMS
  - Driven by **hardware** advances !
- Hardware advances affecting databases
  - Large main memory capacity
  - Complex multi-core processors
  - Scalable memory hierarchy (including fast networks)
- How can we achieve **high** performance in a **modern** database ?
  - Database system **specialization**
  - **Adapting** to the hardware dynamics

Modern Database Specialization

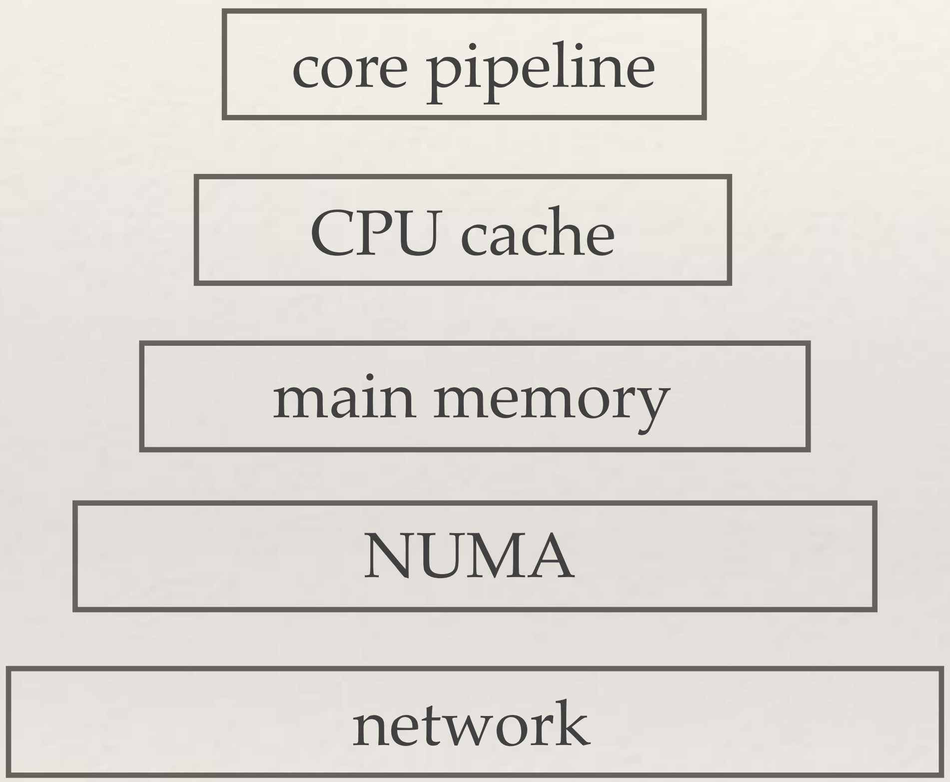
- Transactional DBMS
  - Focus on **transactions**
  - **Update** a **few tuples** per transaction
  - **Row**-store
- **Analytical** DBMS
  - Focus on **queries** for **analysis**
  - **Read** a **few columns** from **many tuples** per query
  - **Column**-store
- Others (e.g. scientific, graph, …)

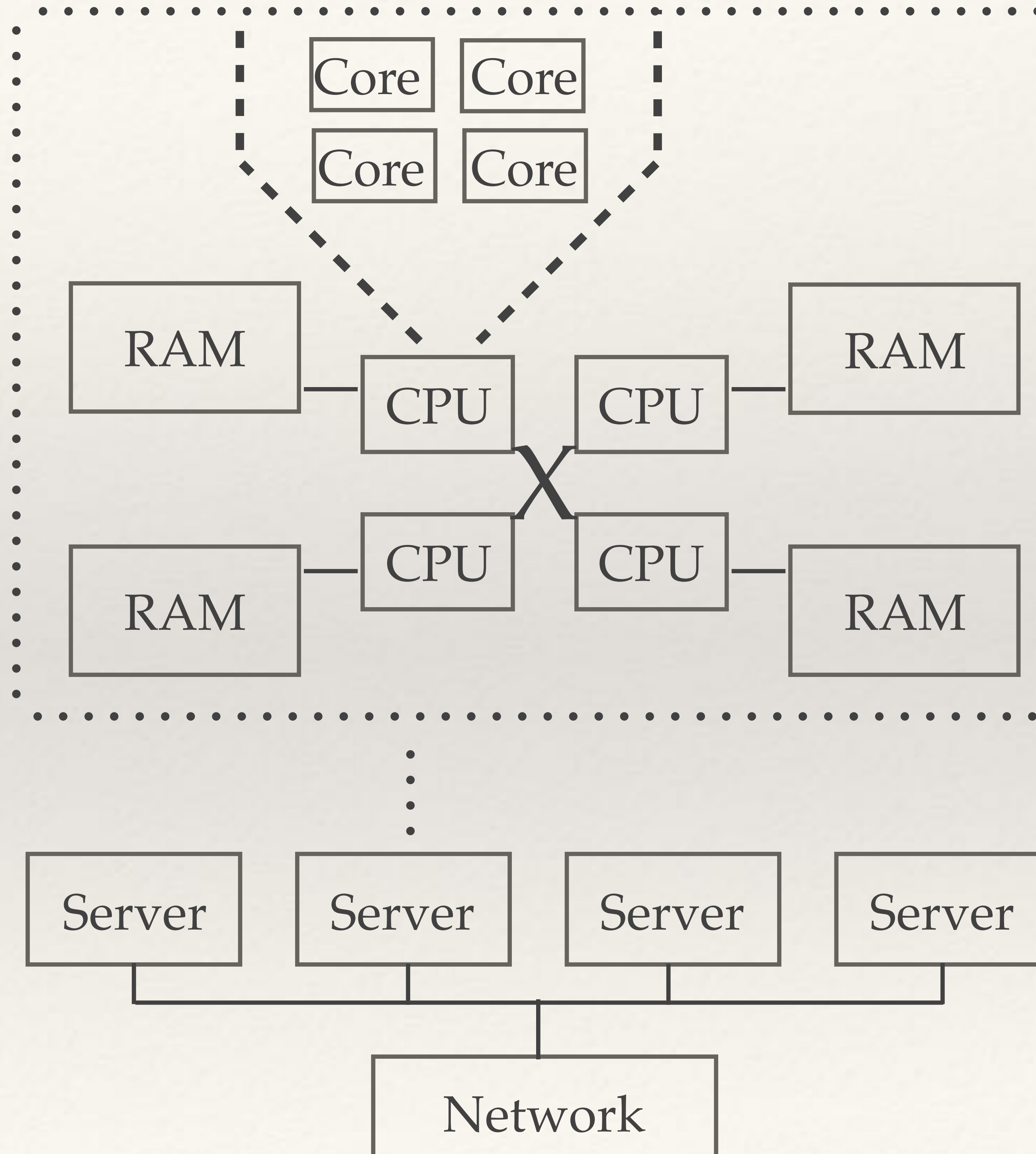
Research Statement

- How can we **increase performance** in a **modern analytical database** ?
  - By optimizing **analytical query execution** for each **hardware layer !**

- **Why** does it work **?**
  - Hardware has **always** driven database **design** & **implementation** (disks originally)
  - Hardware becomes more **complex** making hardware-oblivious designs **ineffective**
  - **Our** solutions are **hardware-conscious** and utilize complex modern hardware **features**
  - Hardware becomes increasingly **parallel** to efficiently process **larger** datasets
  - **Our** solutions push the boundaries of parallelism (via data parallelism, many-cores, …)

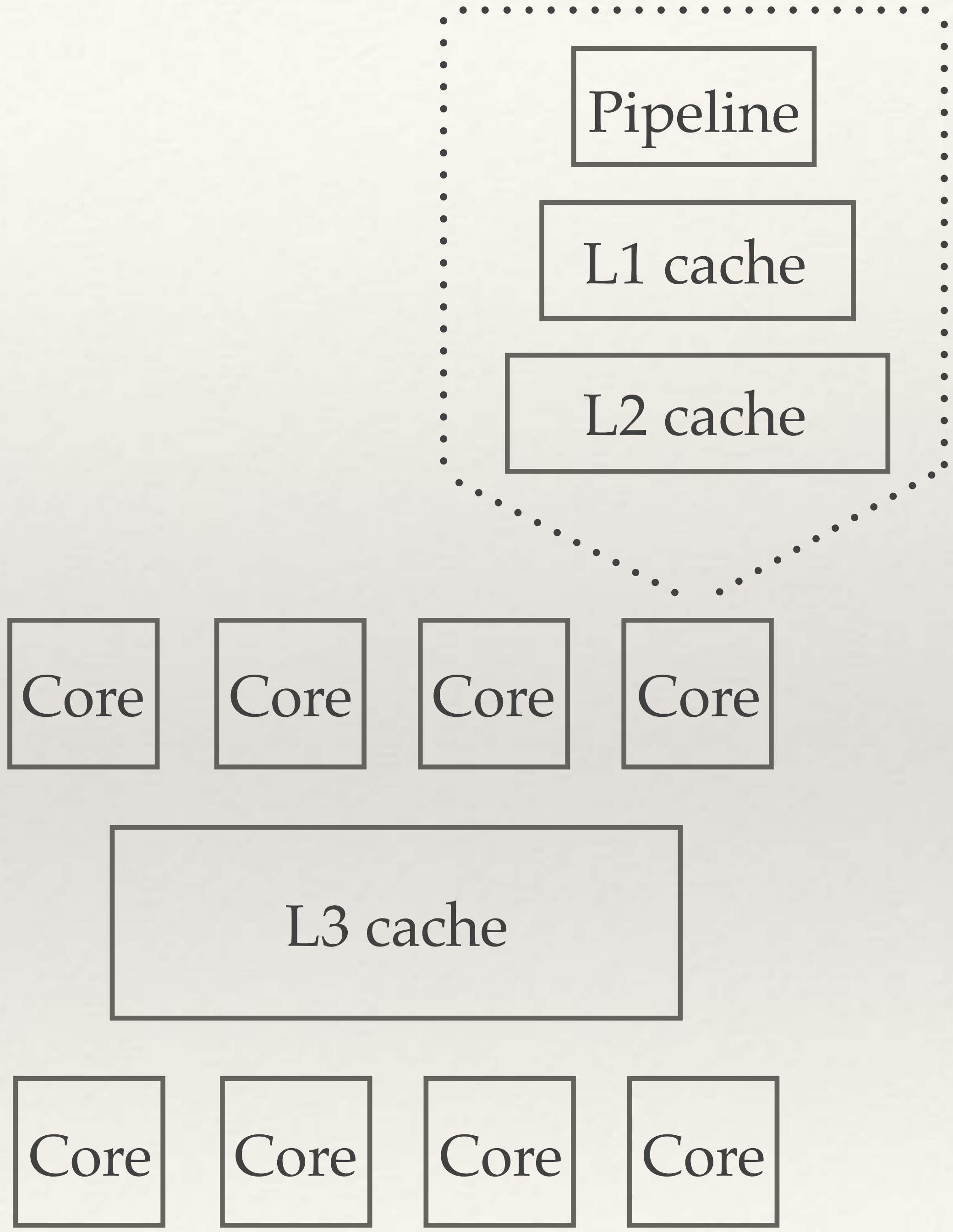
Layers of Modern Hardware





Modern Mainstream CPUs

- Thread parallelism
  - Multiple cores
  - Multiple threads per core
- Instruction level parallelism
  - Out-of-order execution
  - Super-scalar pipeline
- Data parallelism
  - SIMD vectorization



SIMD Vectorization

- As compiler optimization
  - Works for **simple** loops only
  - Insufficient for database operators

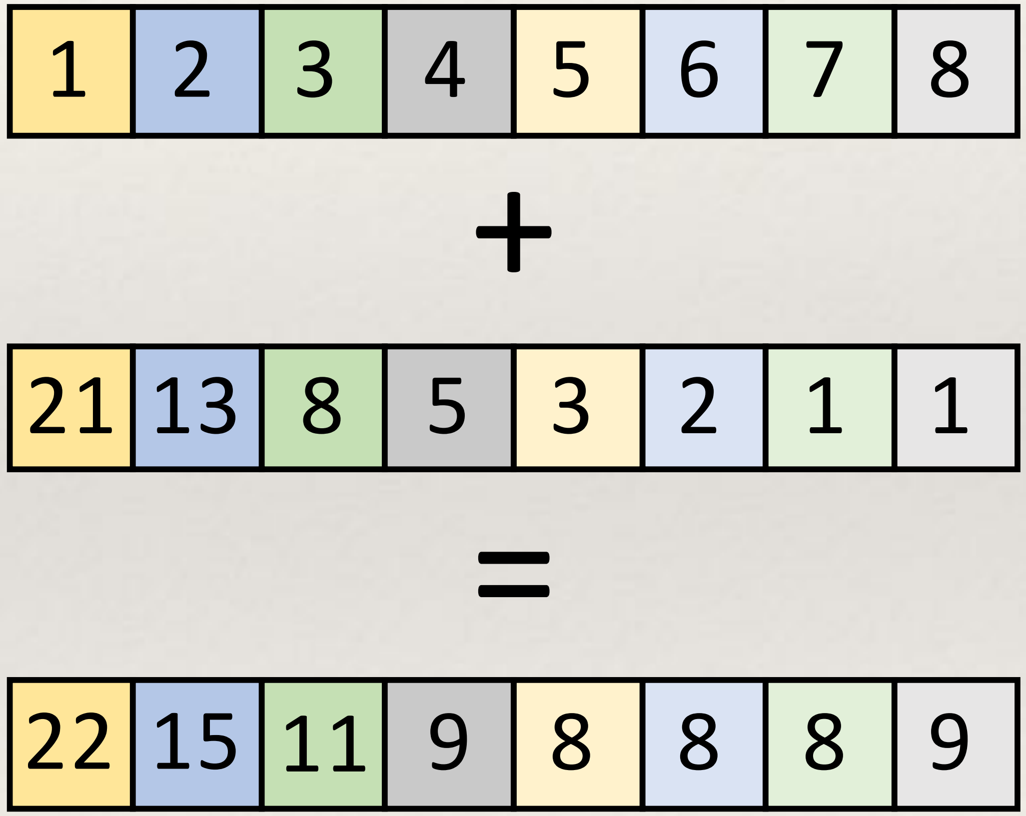
```
for (i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
}
```

scalar code

SIMD code

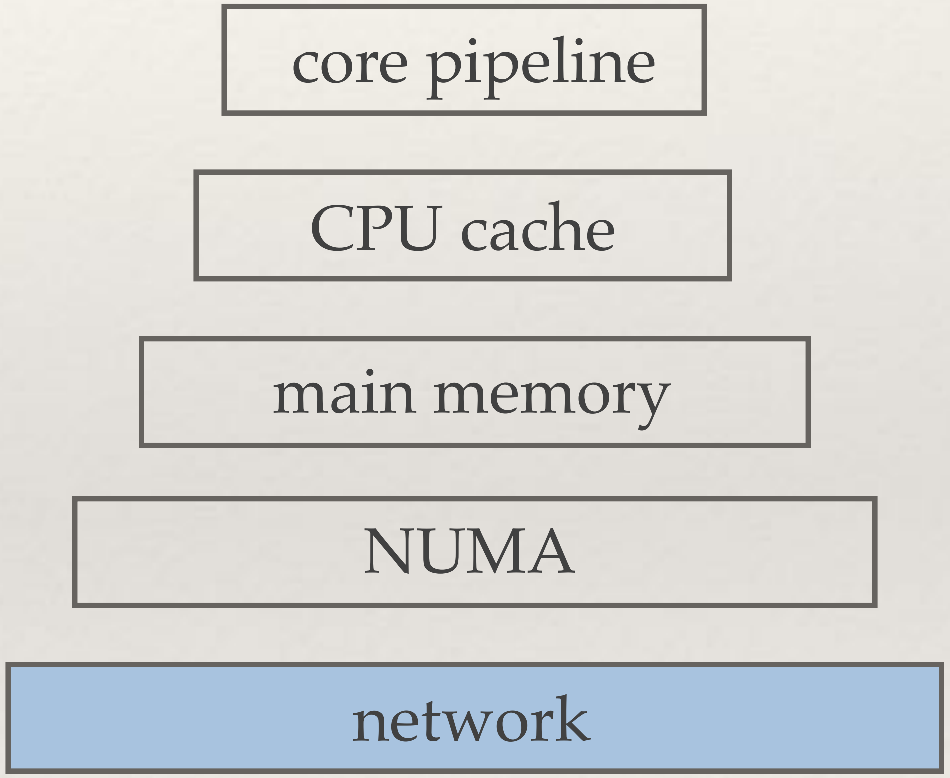
```
for (i = 0; i < n; i += 16) {
    __m512i x = _mm512_load_si512(&a[i]);
    __m512i y = _mm512_load_si512(&b[i]);
    __m512i z = _mm512_add_epi32(x, y);
    _mm512_store_si512(&c[i], z);
}
```

8-way SIMD addition



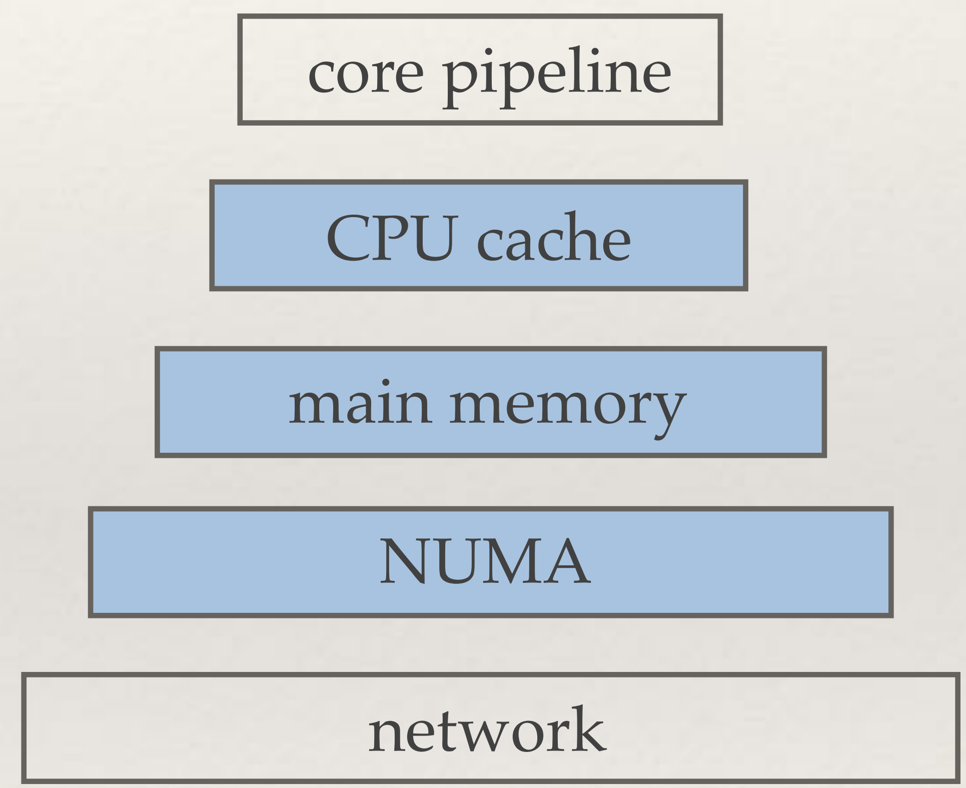
Bottlenecks of Query Execution

- Network-bound
  - Distributed joins with minimal network traffic (Part 1)



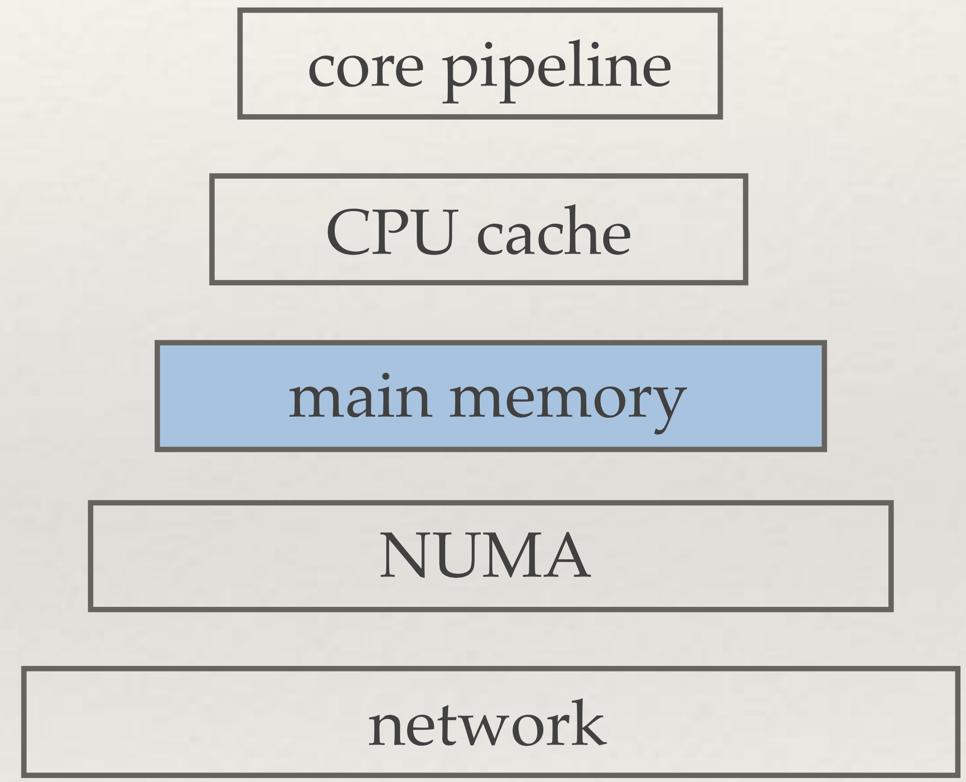
Bottlenecks of Query Execution

- Network-bound
  - Distributed joins with minimal network traffic (Part 1)
- Memory-bound (random access)
  - (Cache-RAM-NUMA)-aware partitioning (Part 2)



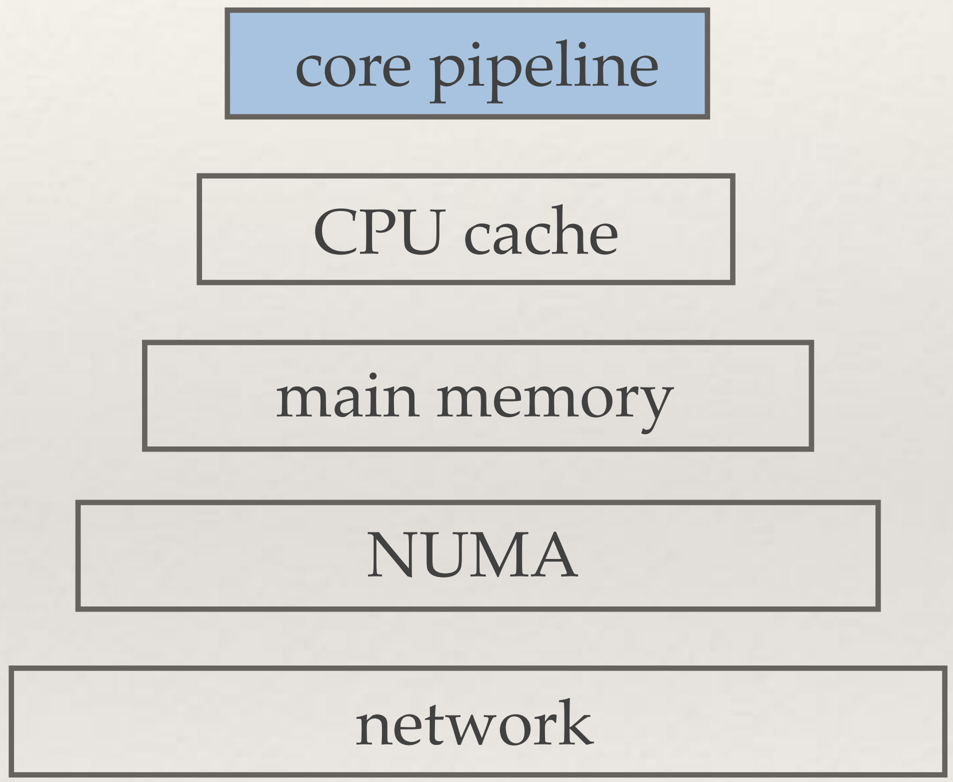
Bottlenecks of Query Execution

- Network-bound
  - Distributed joins with minimal network traffic (Part 1)
- Memory-bound (random access)
  - (Cache-RAM-NUMA)-aware partitioning (Part 2)
- Memory-bound (sequential access)
  - Lightweight in-memory compression (Part 3)

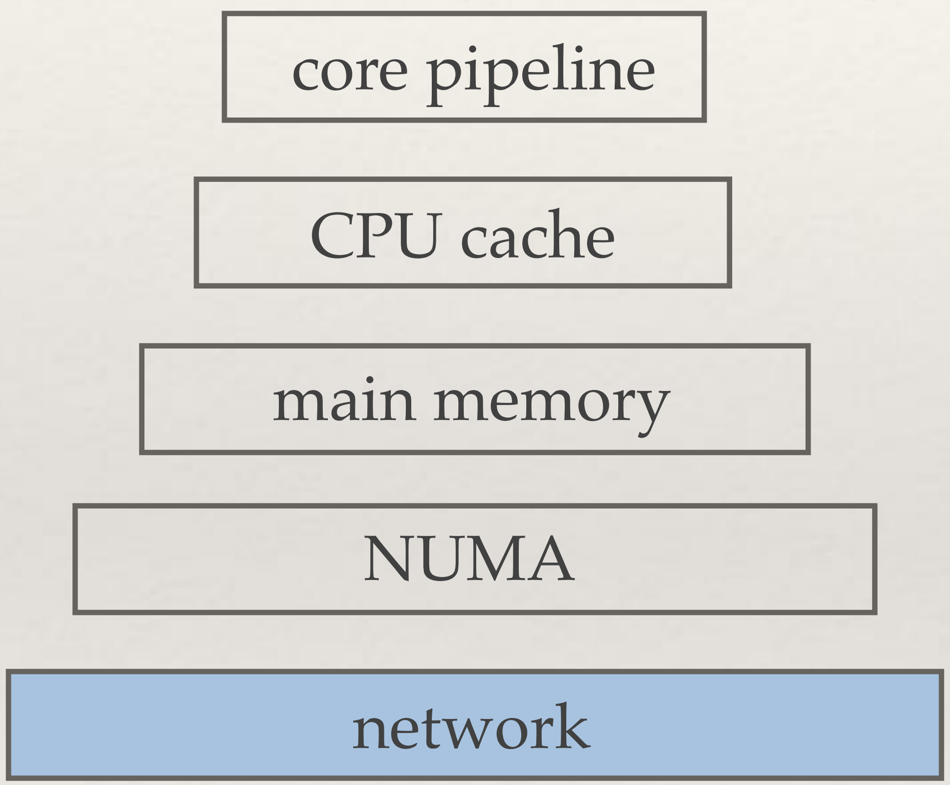


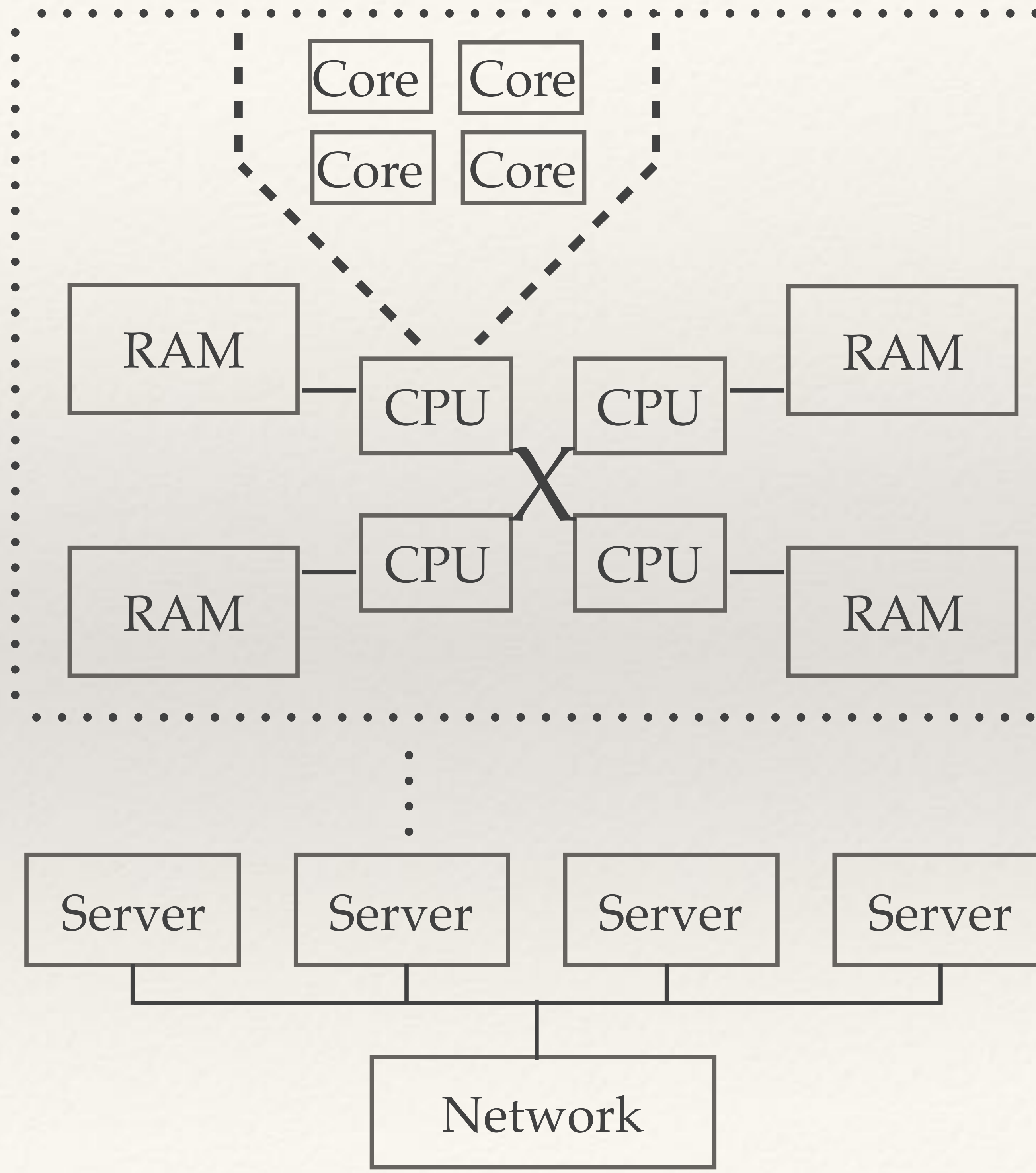
Bottlenecks of Query Execution

- Network-bound
  - Distributed joins with minimal network traffic (Part 1)
- Memory-bound (random access)
  - (Cache-RAM-NUMA)-aware partitioning (Part 2)
- Memory-bound (sequential access)
  - Lightweight in-memory compression (Part 3)
- Compute-bound
  - Advanced SIMD vectorization techniques (Part 4)



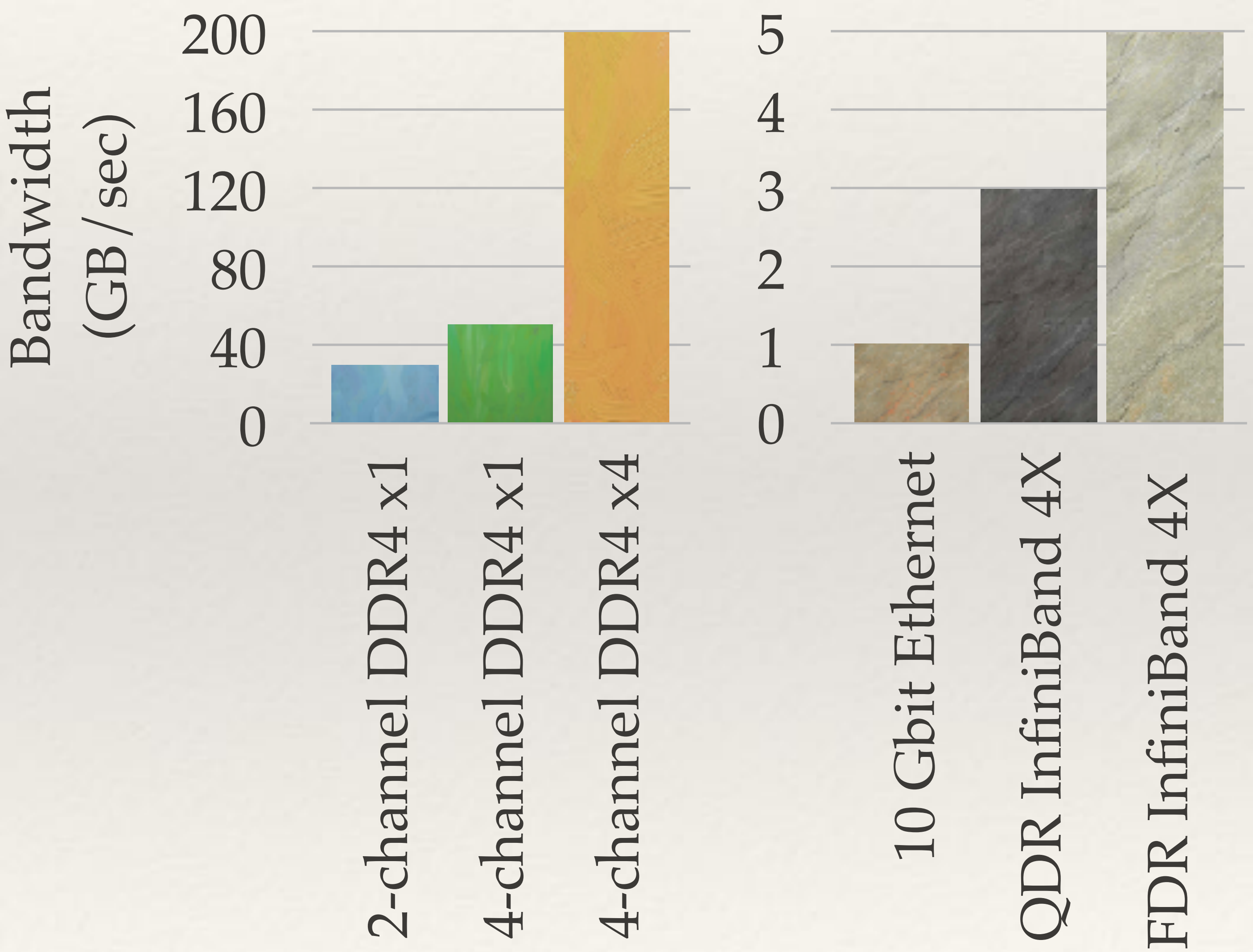
Part 1: Network-bound



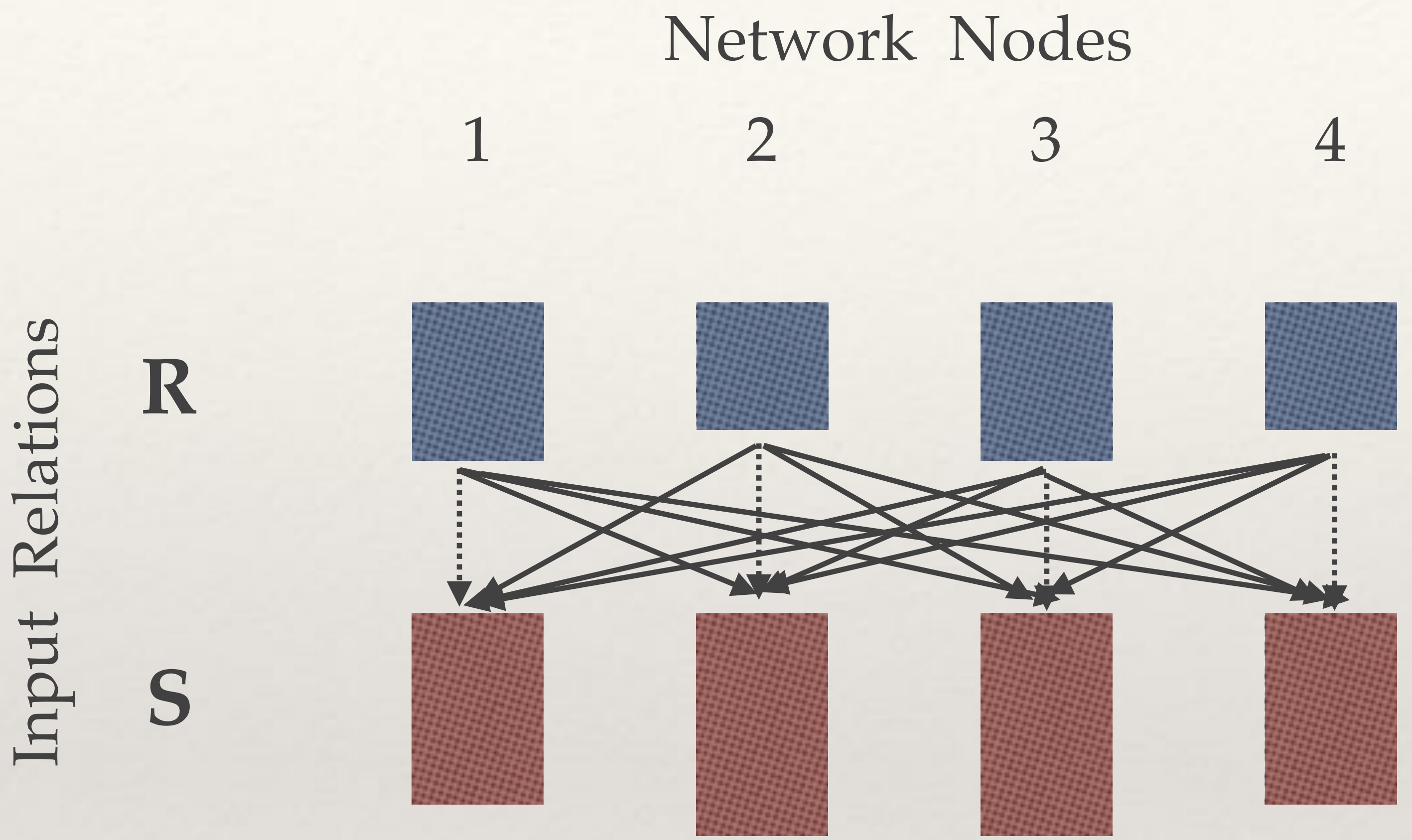


Network << RAM

- Network is slower than RAM
  - 4-channel DDR4 x4 = ~200 GB/s
  - FDR InfiniBand 4X = ~5 GB/s
- Optimize for network traffic
  - Joins are **dominant**
  - Disk hash join ~ network hash join
  - **Our contribution**: track join

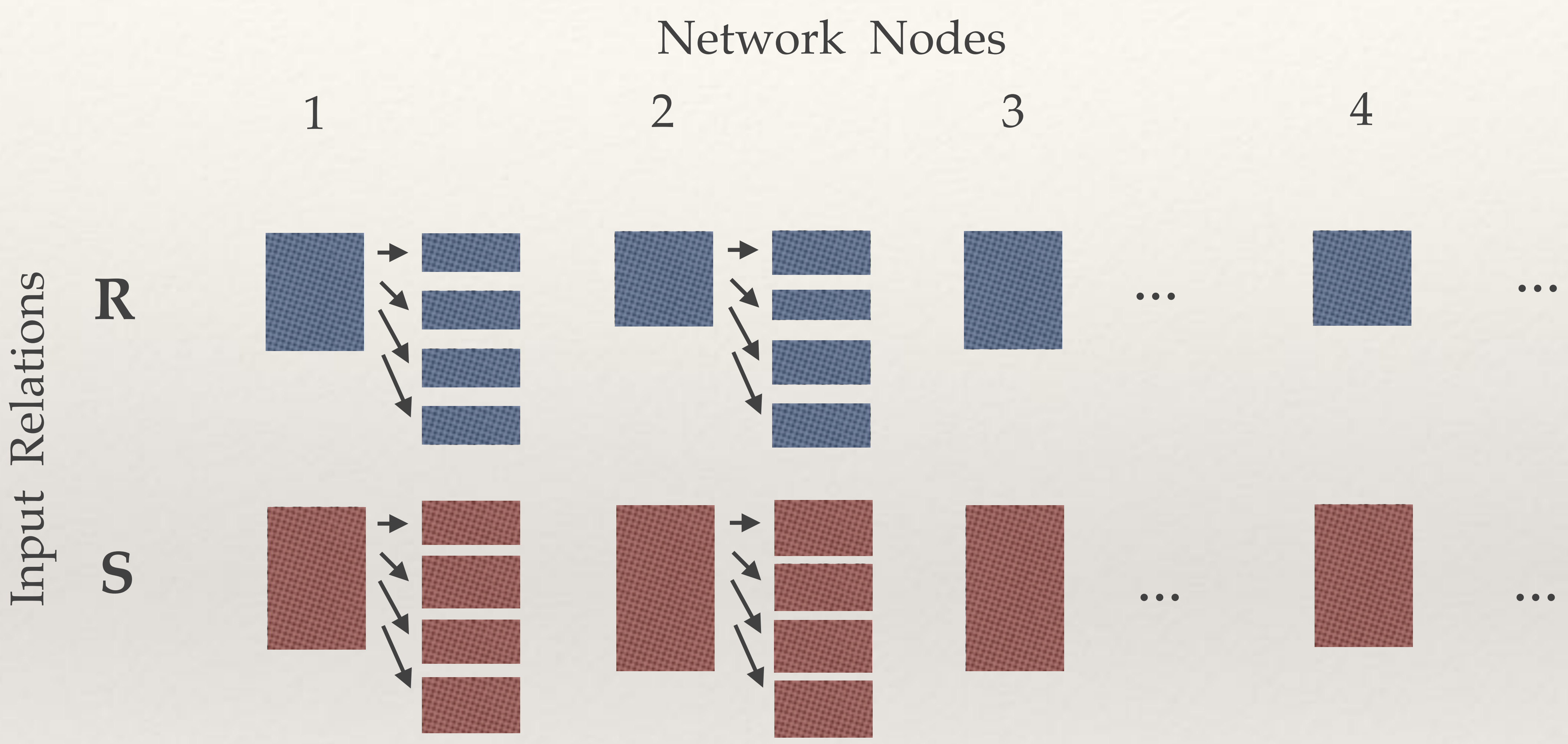


Previous Work: Broadcast Join



- Good if one table is small
- Bad for large number of nodes

Previous Work: Hash Join

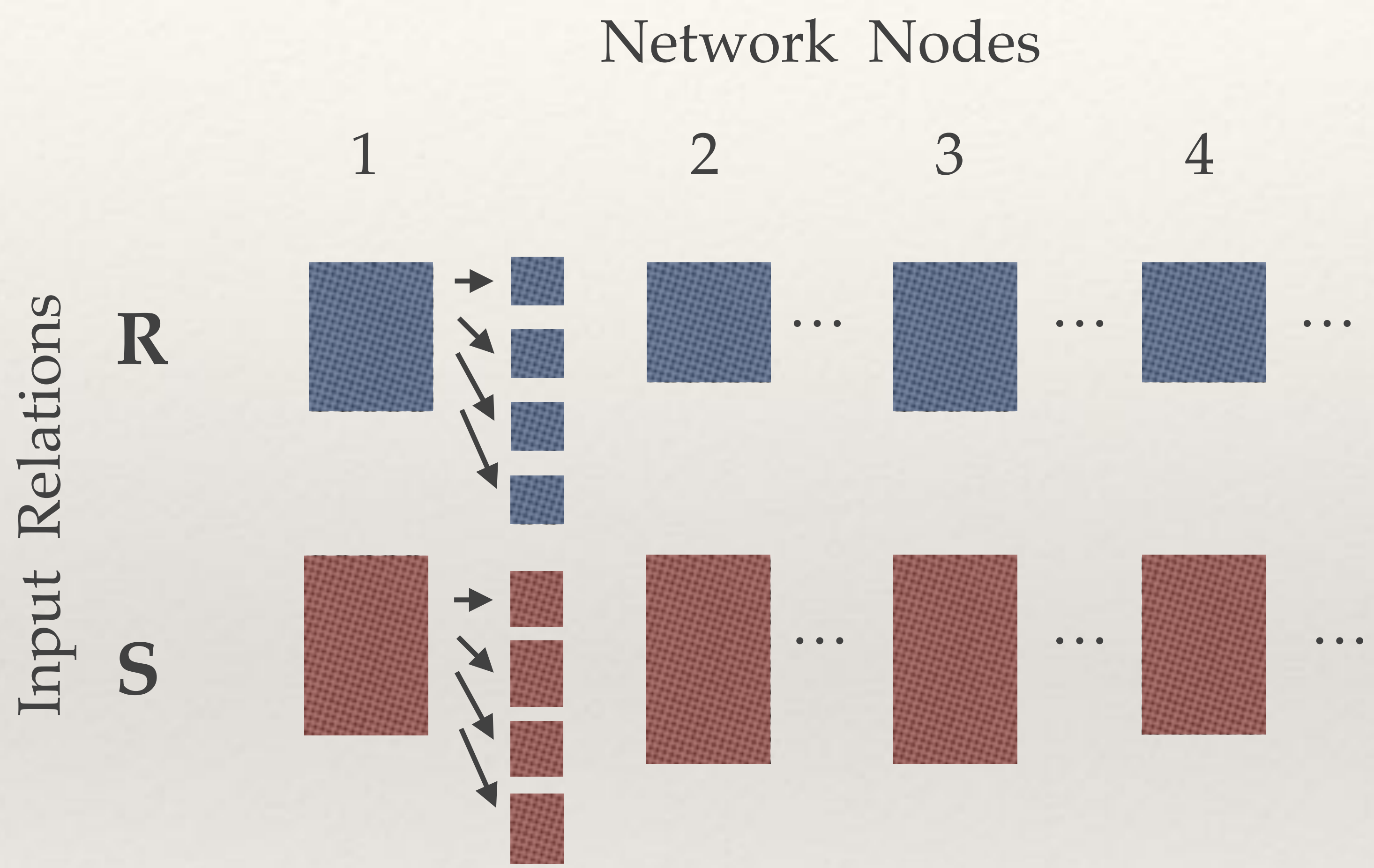


- Good if both tables are large
- Bad if one table is small

Track Join: Minimize Network Traffic

- Basic idea
  - Logical decomposition into **Cartesian product** joins
  - **Optimize the transfers** for each Cartesian product
- Basic steps
  - **Track** tuple locations per unique join key
  - Generate **optimal** transfer schedule per key
  - Transfer data and execute join
- Multiple variants
  - 2-phase, 3-phase, 4-phase

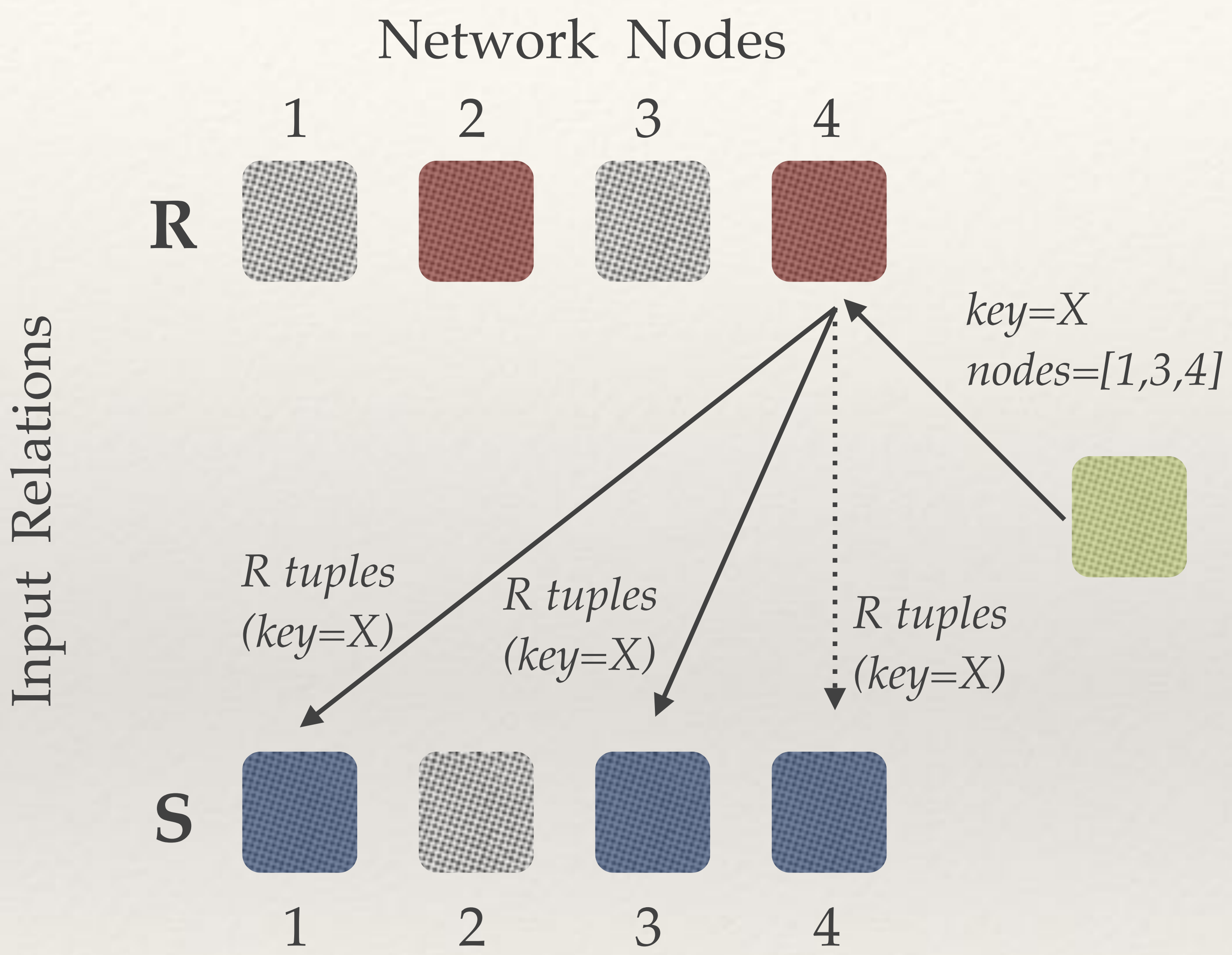
Track Join



Hash partition ***unique*** *keys*

- Tracking phase
  - Like a hash join of **keys only**

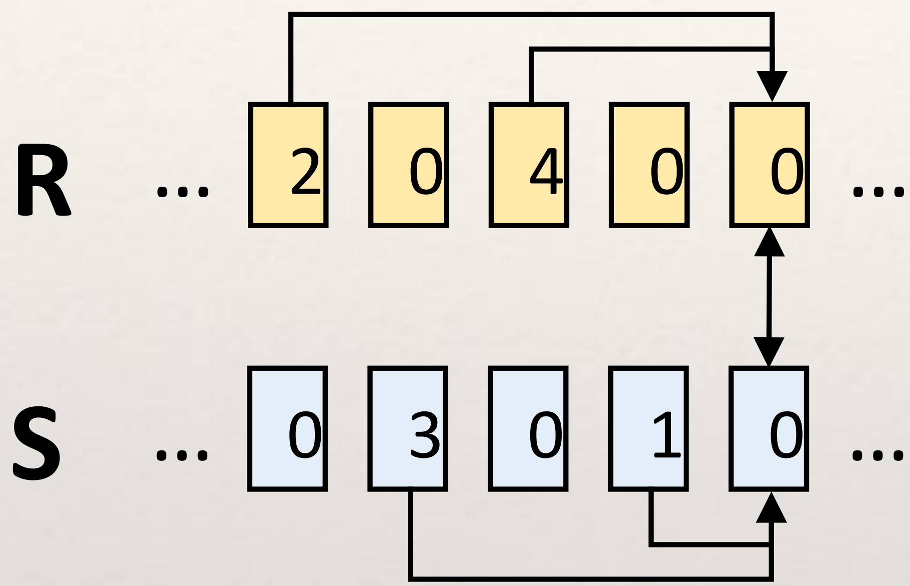
Track Join (2-phase)



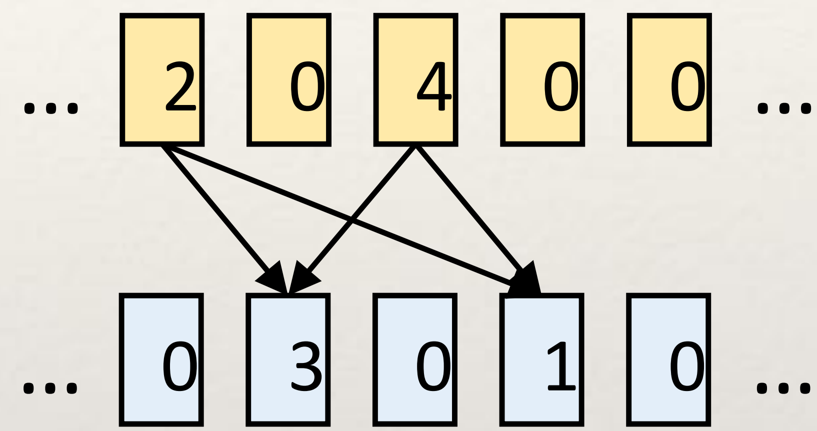
- Selective broadcast
    - On locations that have **at least one** tuple

Hash Join & Track Join

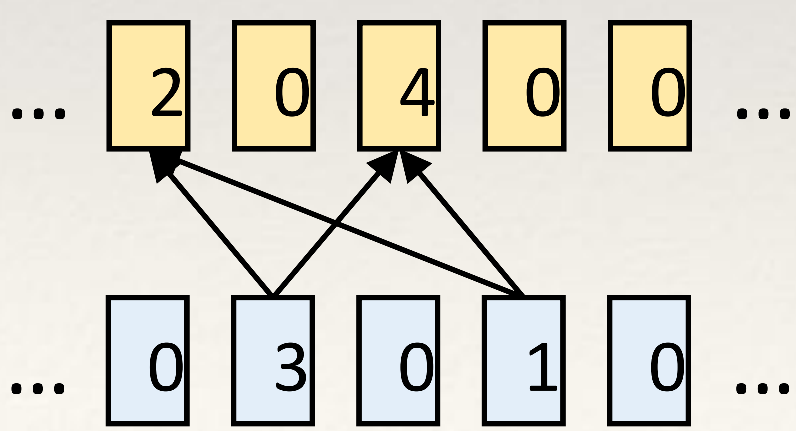
- Hash Join (network cost = 10)



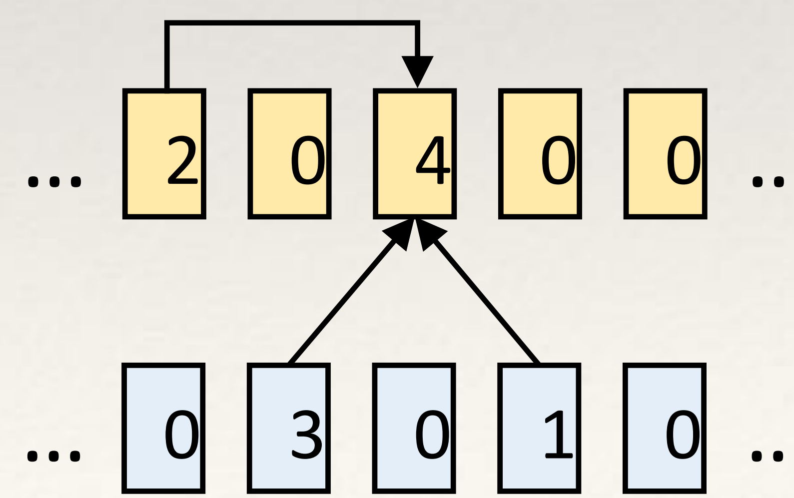
- 2-phase Track Join (network cost = 12)



- 3-phase Track Join (network cost = 8)

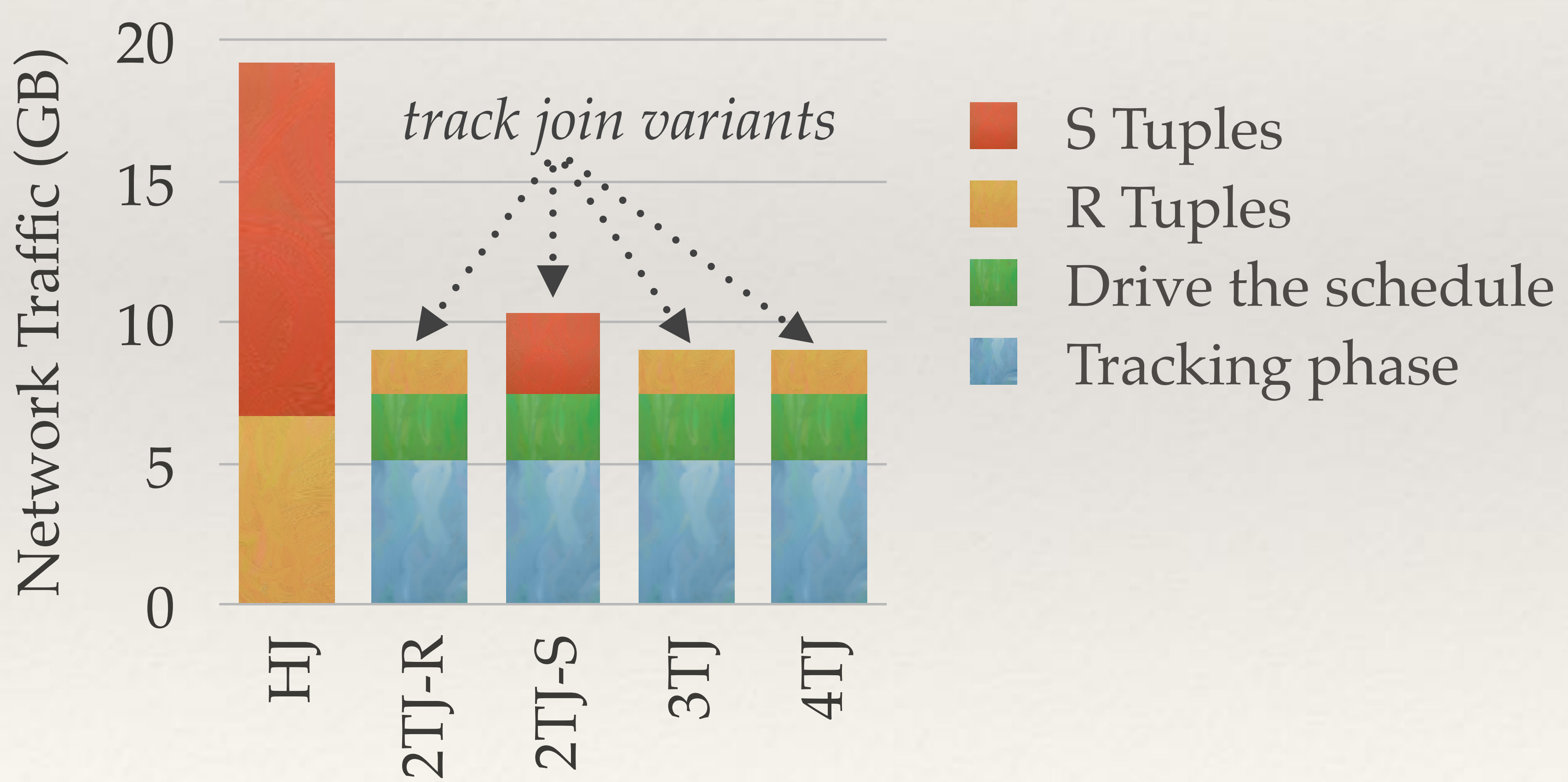


- 4-phase Track Join (network cost = 6)



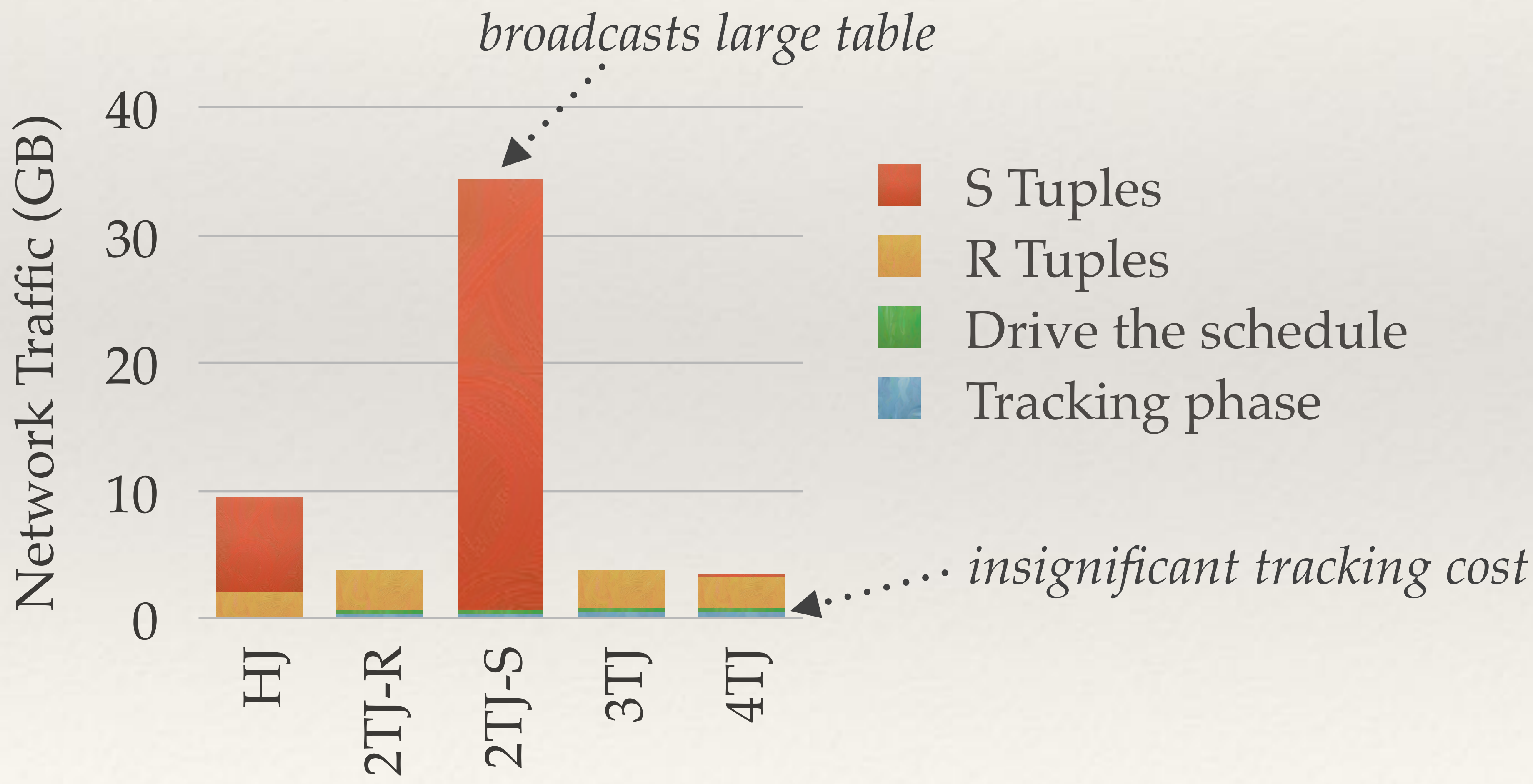
Real Workload 1-1

- Real workload 1
  - 1–1 join
  - Pre-existing **locality**



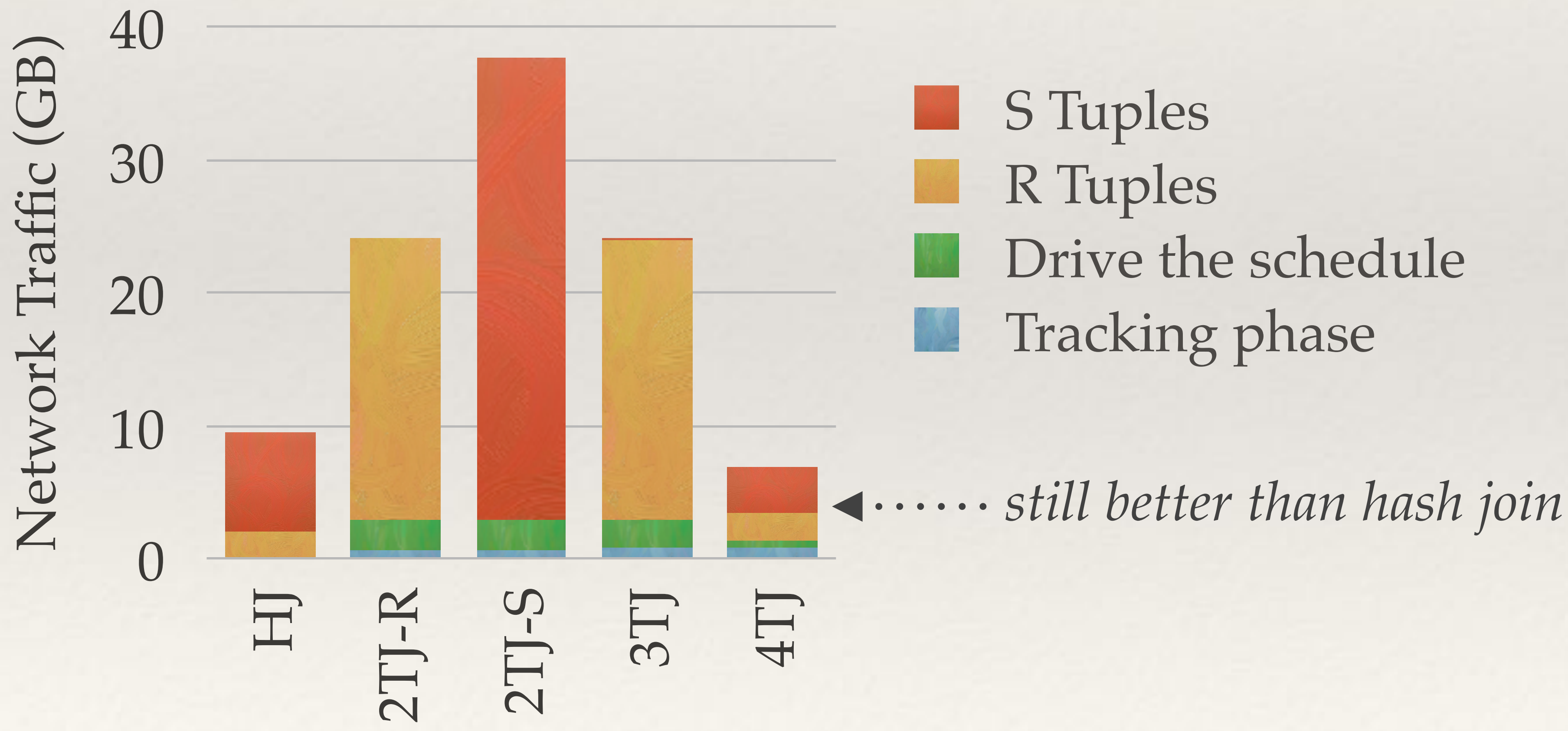
Real Workload M-N

- Real workload 2
  - M—N join i.e., output = **~5X** inputs
  - Pre-existing **locality**



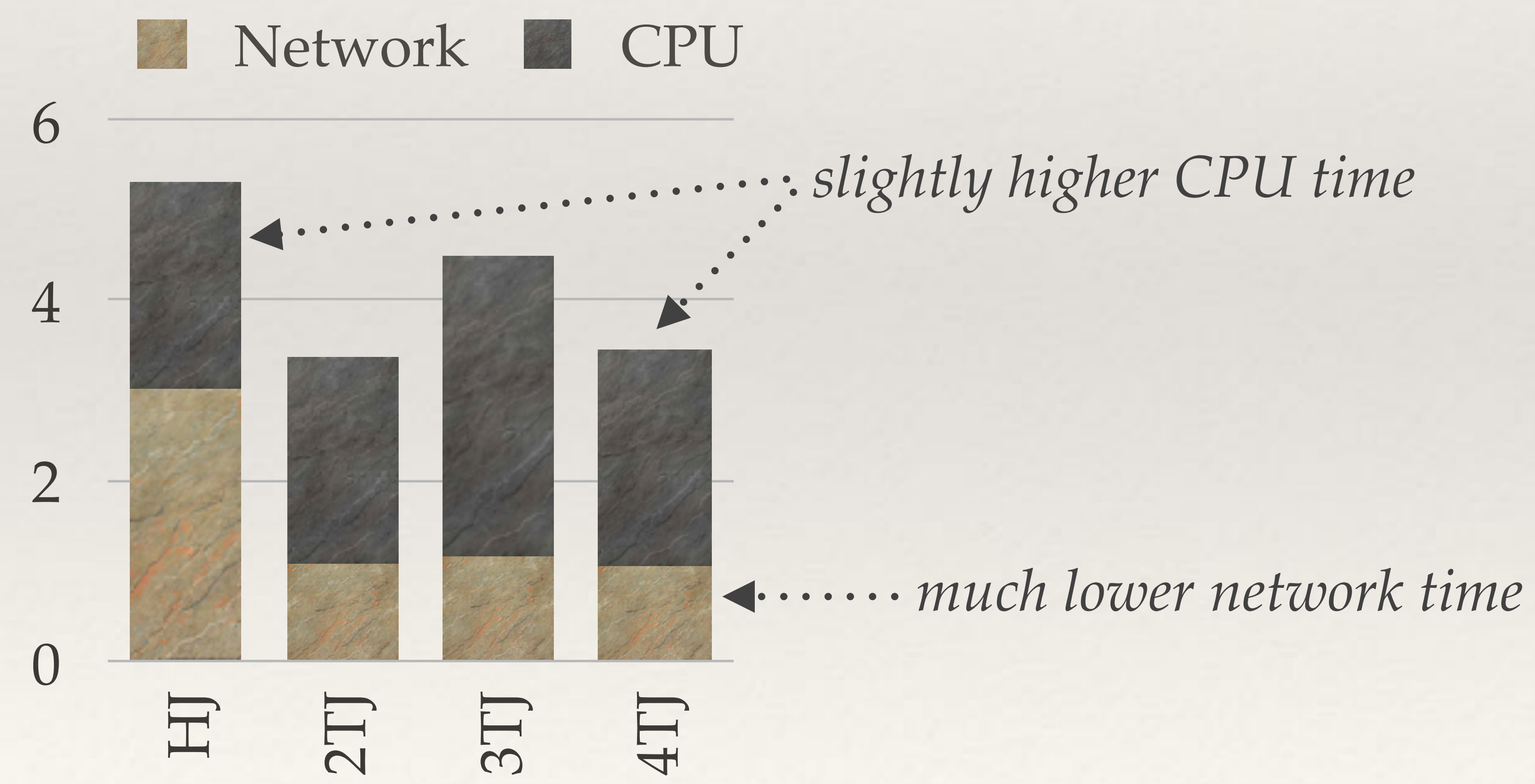
No locality? Use 4TJ

- Real workload 2
  - M—N join i.e., output = **~5X** inputs
  - Shuffle to **remove** locality

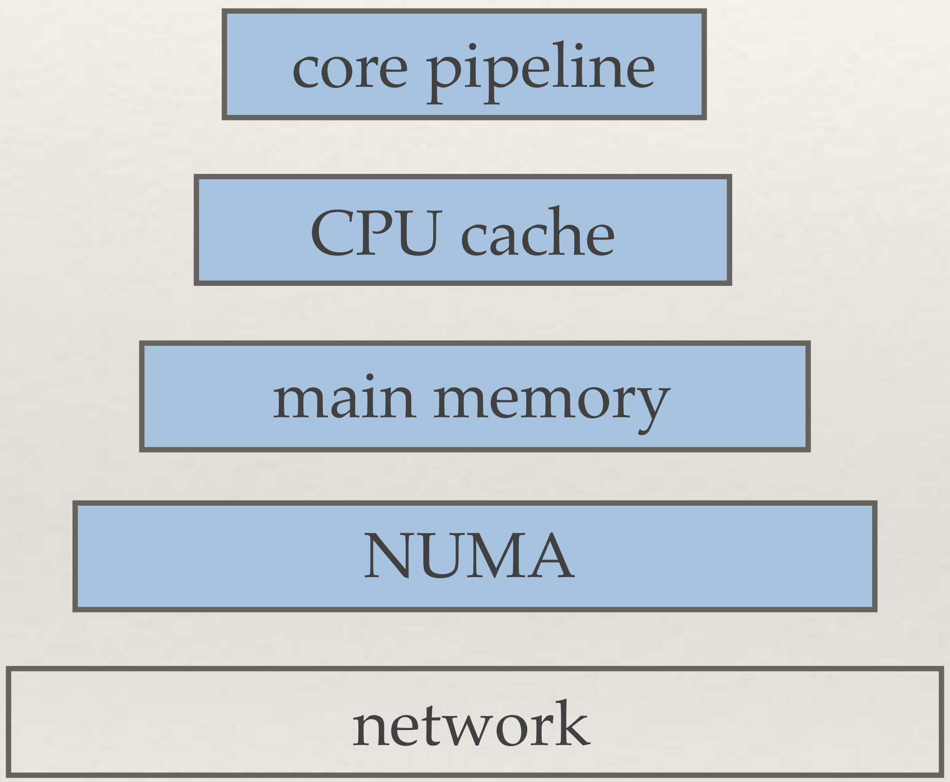


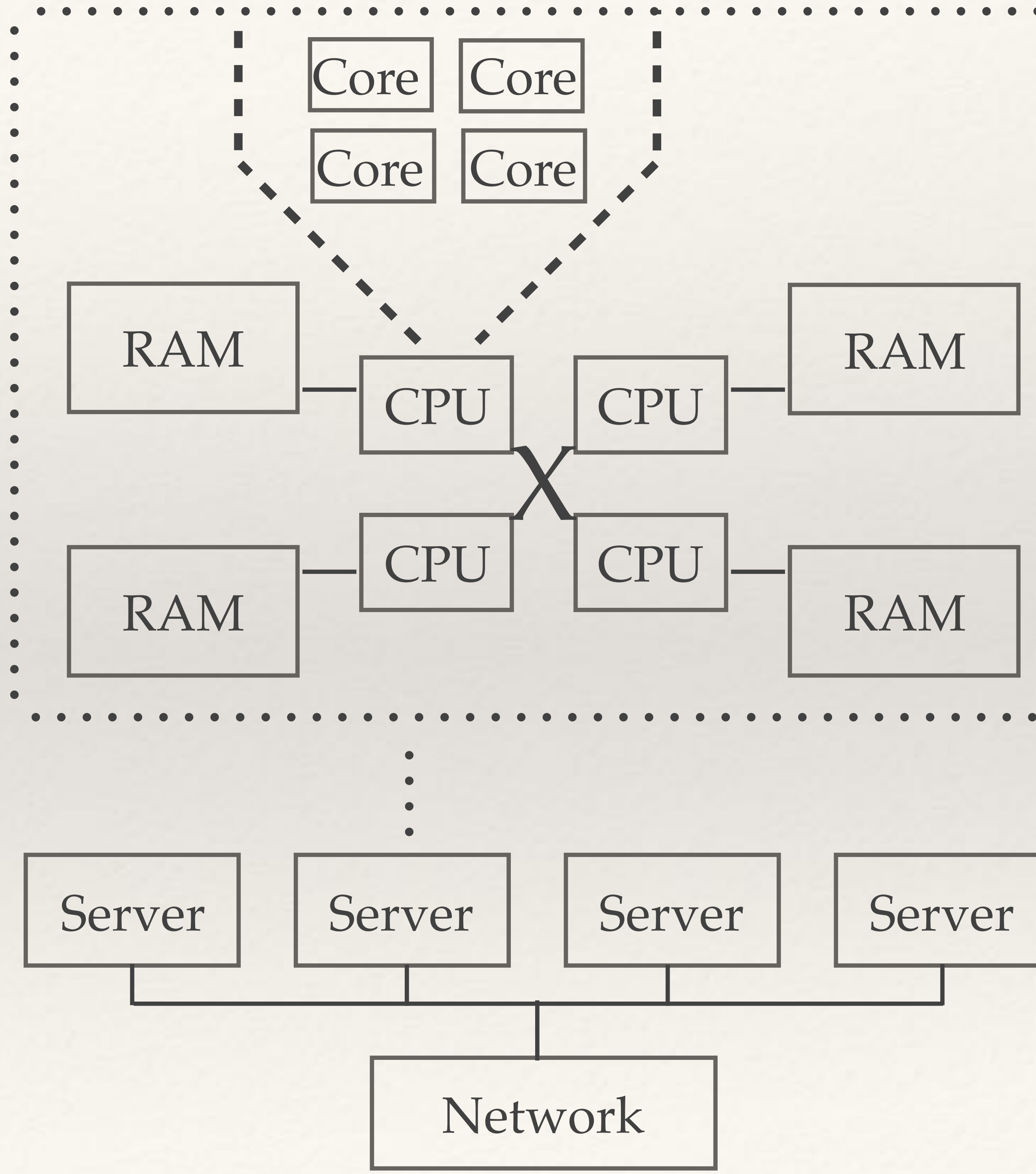
CPU+Network Time

- Non-pipelined implementation
  - 4 servers **x** 2 CPUs/sever **x** 4 cores/CPU **x** 2 threads/core
  - 10 Gbit Ethernet projected from 1 Gbit (Columbia CLIC lab)



Part 2: Memory-bound (random)

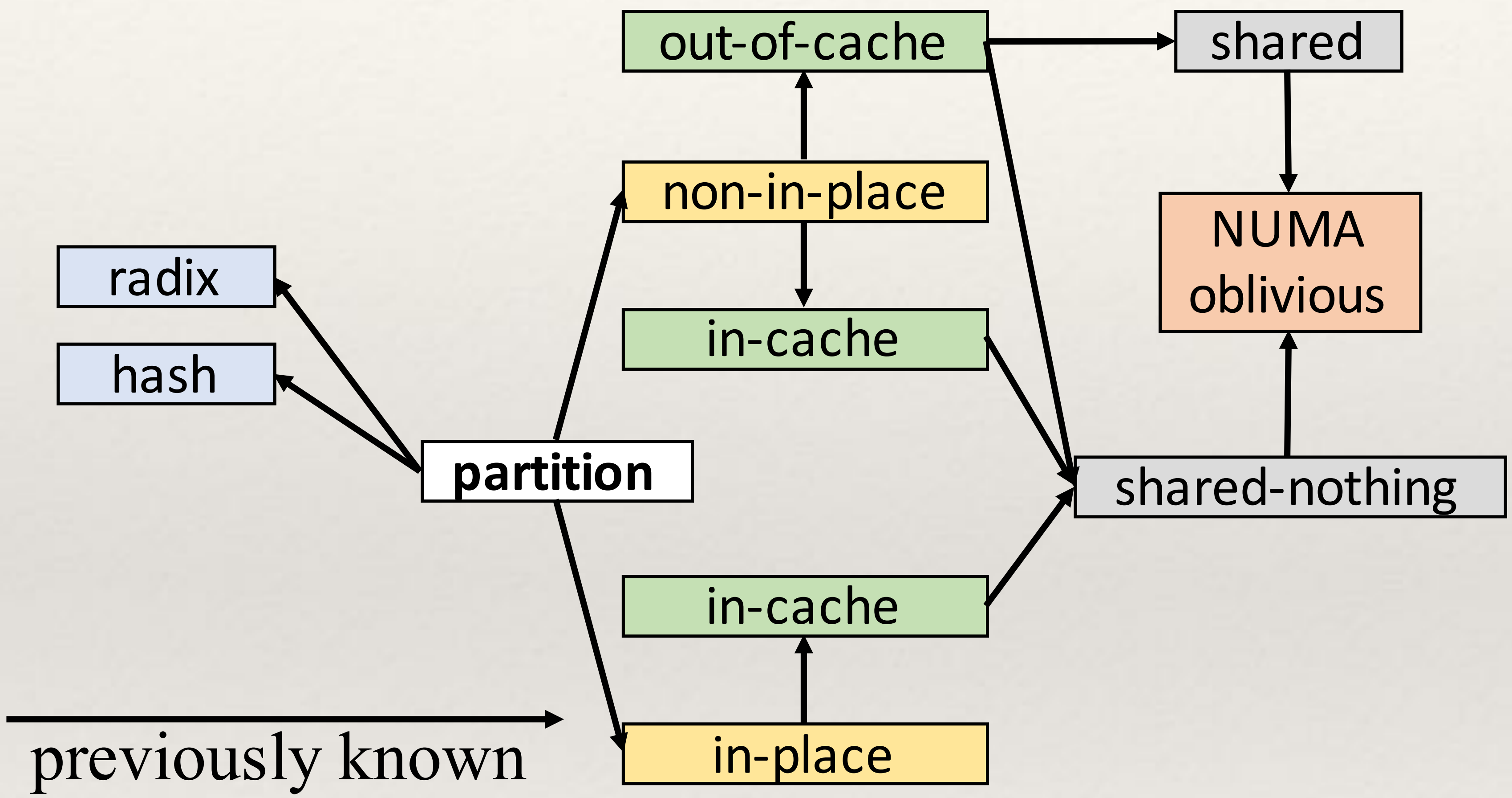




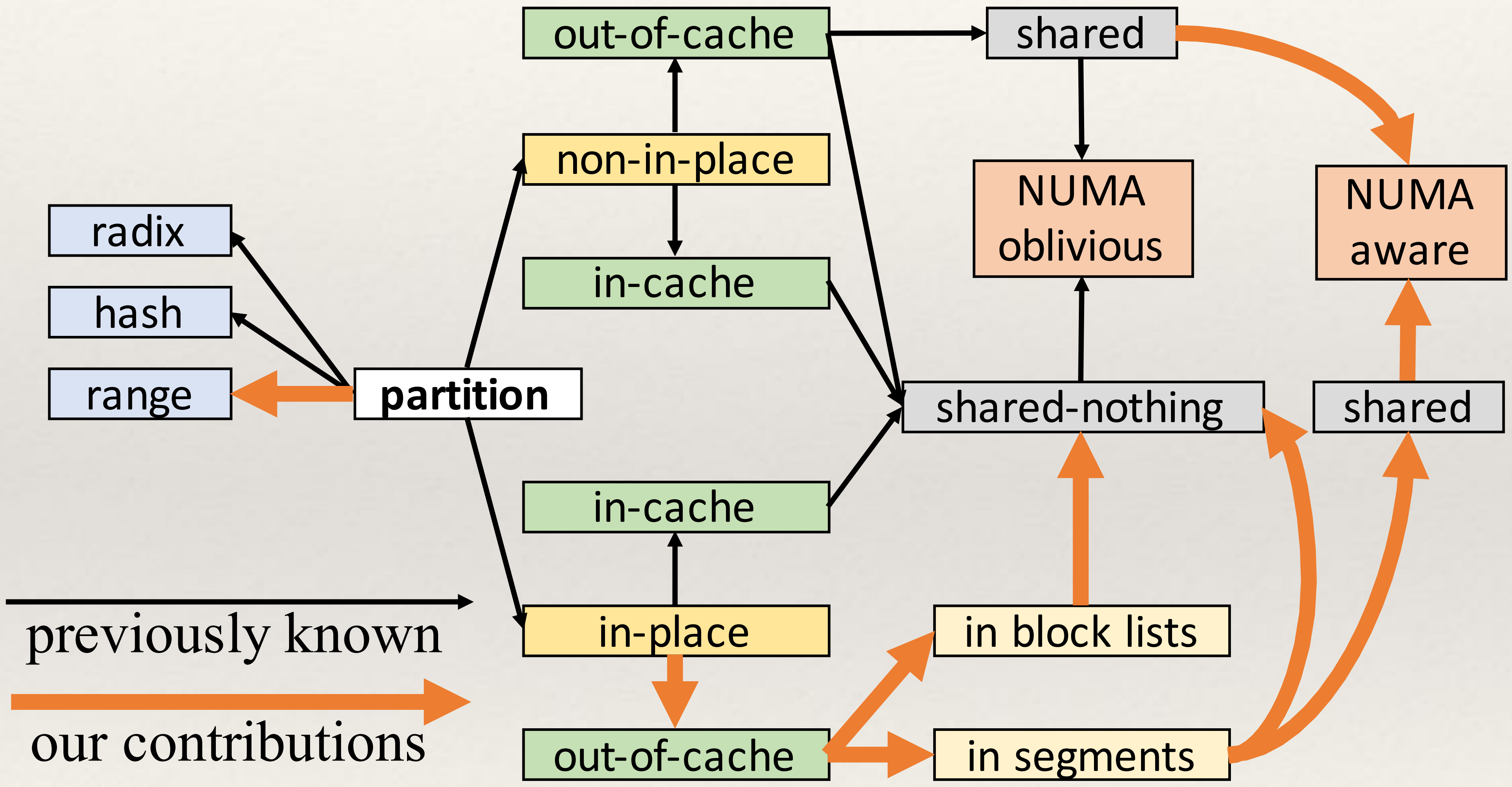
Why partitioning?

- Random accesses << sequential accesses
  - Cache misses
  - TLB misses
- **Where** to use partitioning
  - **Sorting**
  - Joins
  - Group-by aggregation
  - Materialization

Variants of Partitioning

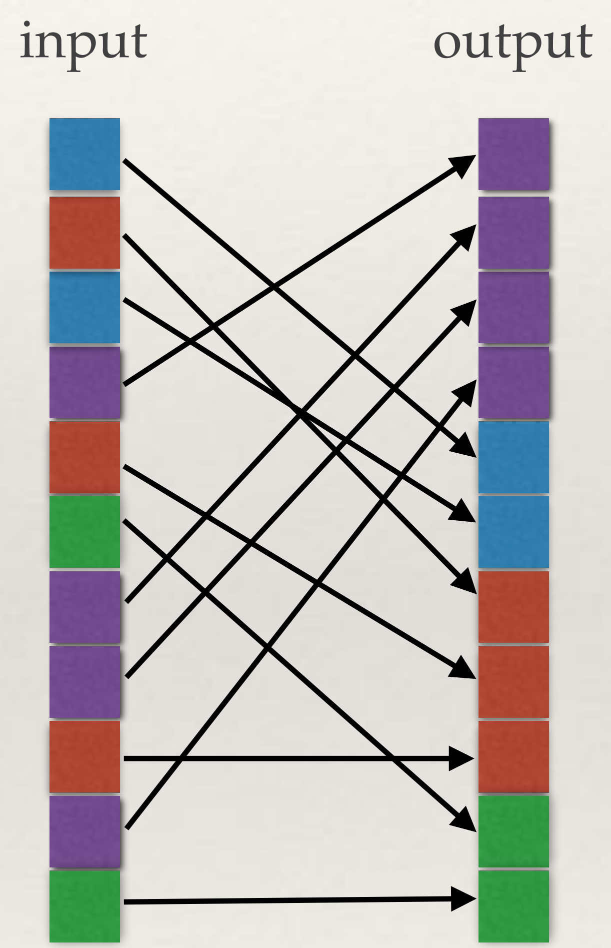


Variants of Partitioning



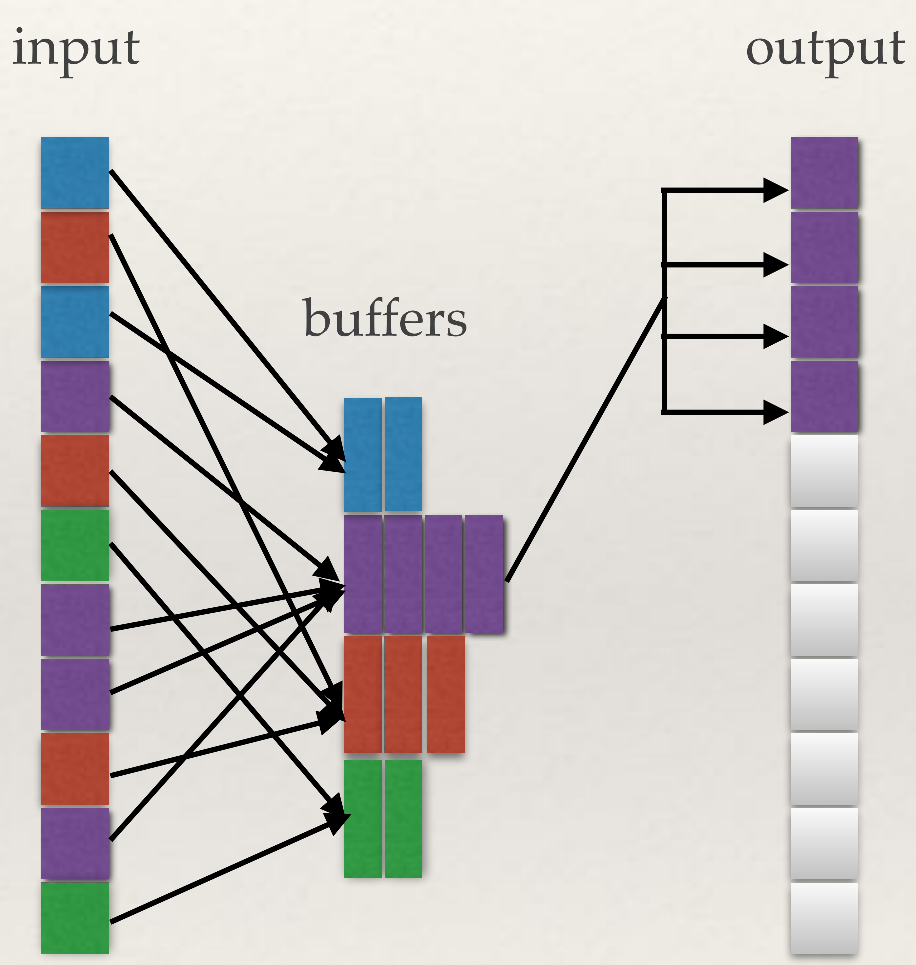
Previous Work: Partitioning small arrays

- Compute **histogram**
  - Contiguous arrays
  - Prefix sum of histogram
- Shuffle the data
  - **Copy** from input to output



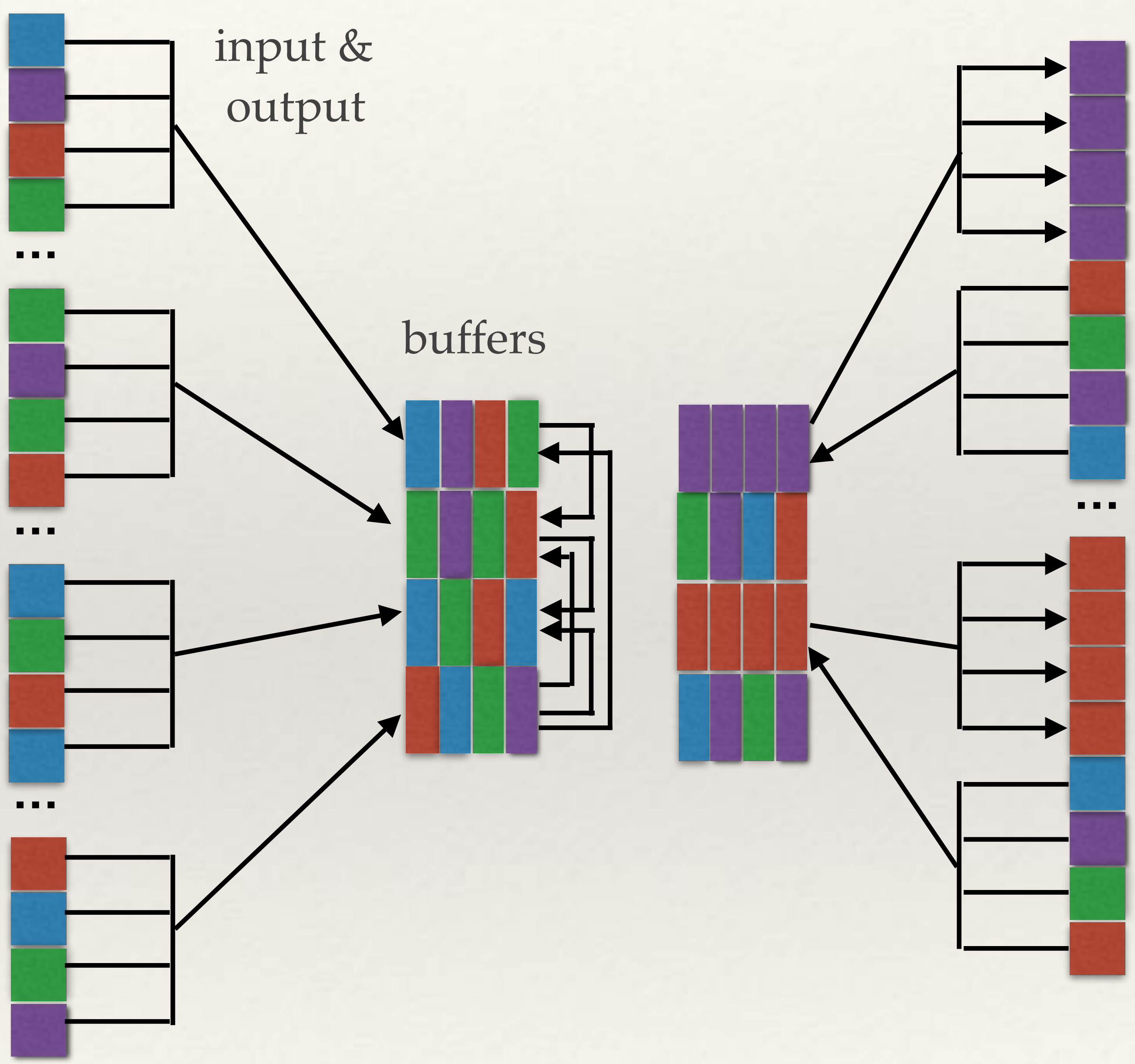
Previous Work: Partitioning large arrays

- If size of array >> size of cache:
  - TLB thrashing
  - Cache conflicts
  - Cache pollution
- Use **buffering**
  - Store tuples in **cache-resident** buffers
  - **Write-combine** full buffers to output
- Parallel
  - **Interleave** histograms during prefix sum

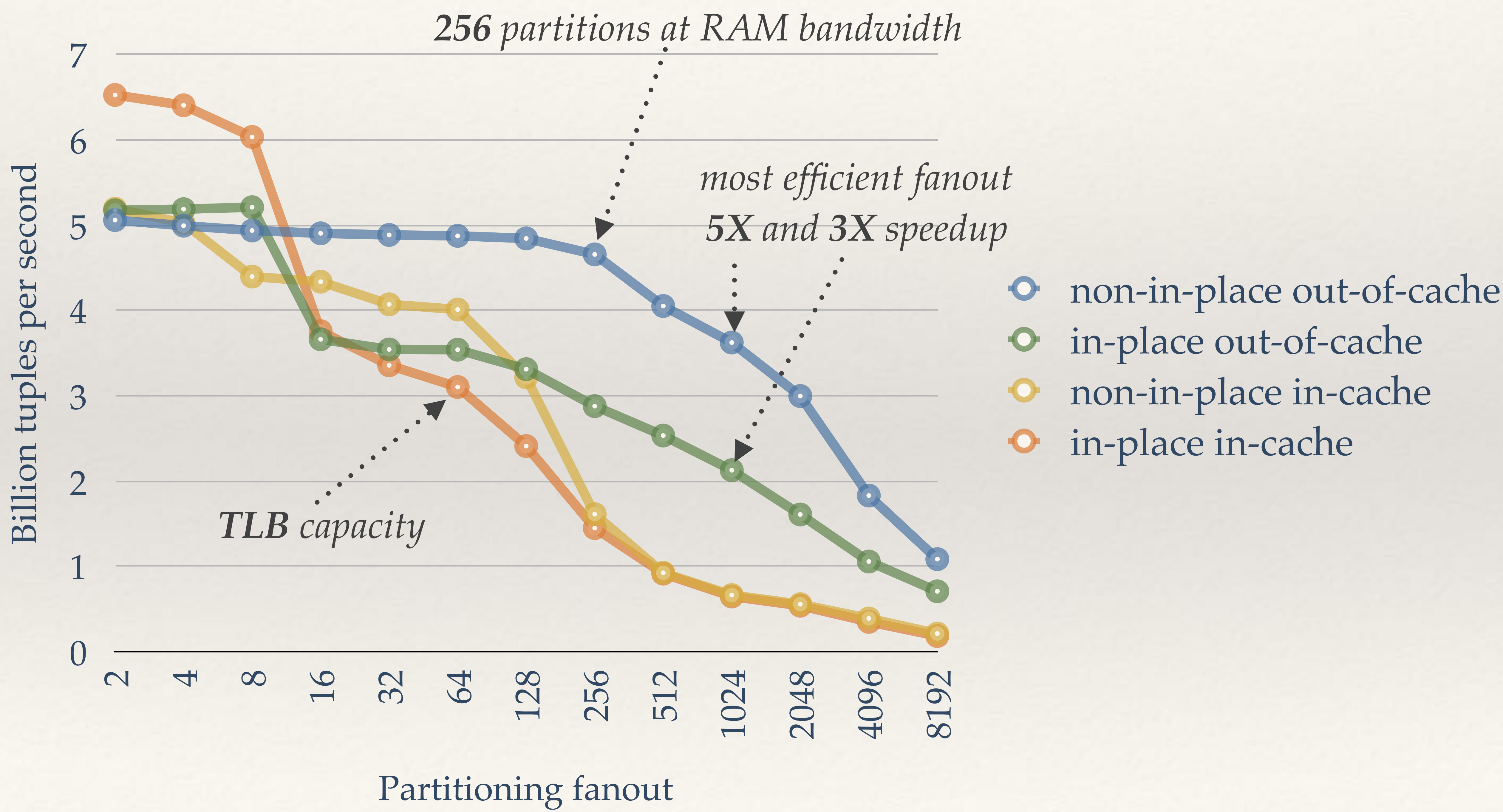


Partitioning large arrays in place

- Transfer data in cache lines
  - **Amortize** out-of-cache accesses
  - RAM <—> CPU cache
- “Work” on the cached buffers
  - Similar to in-cache (“swap cycles”)
  - Data transferred across buffers
- Recycle buffers when done
  - Flush buffer when filled
  - Refill buffer with next data

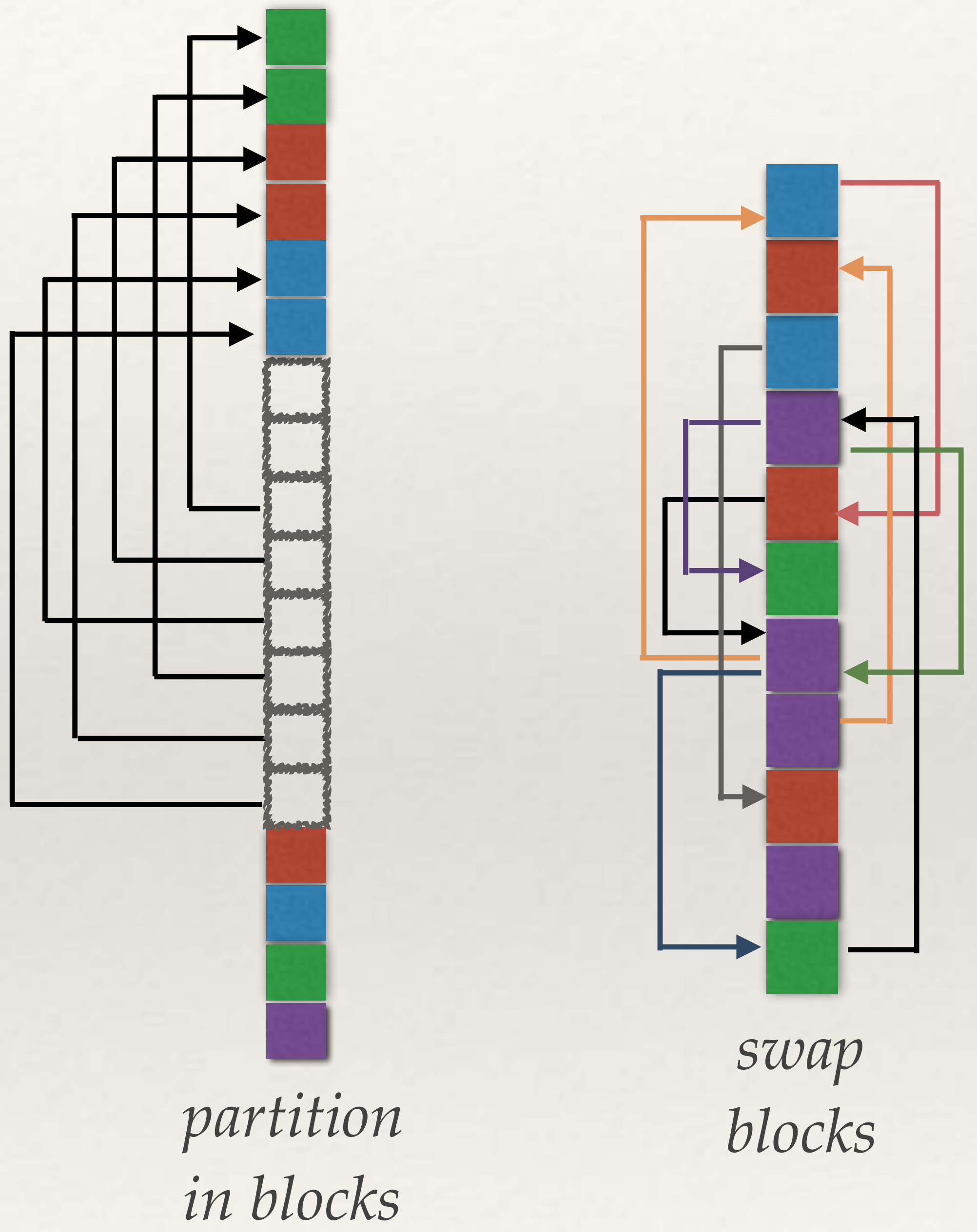


Partitioning large arrays



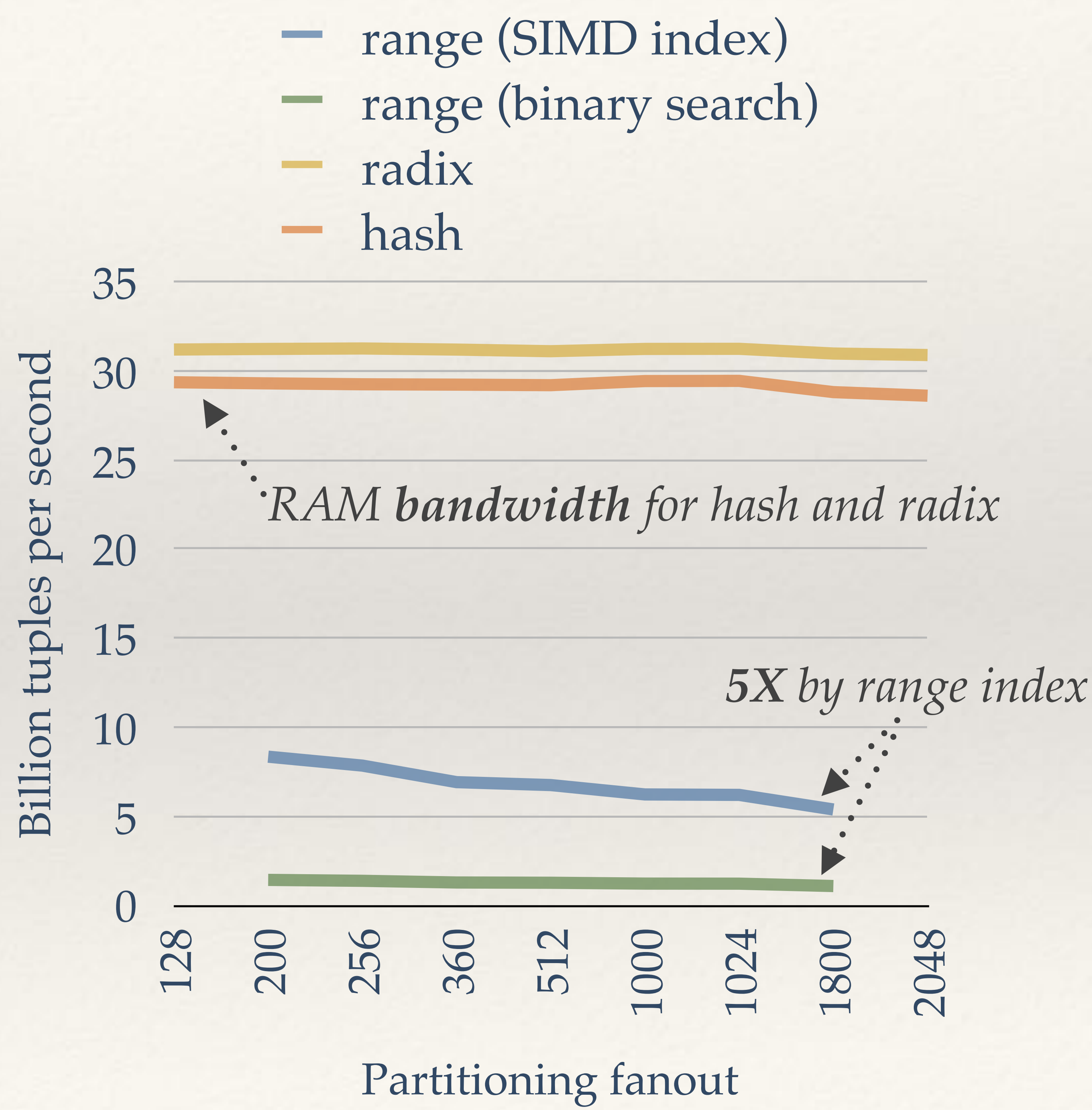
Parallel partitioning in-place

- Swap tuples in-place
  - Using atomics
  - Extreme **synchronization** cost
- Swap blocks of tuples in-place
  - Partition to **list of blocks** in-place
  - Swap **blocks** of tuples
  - **Amortize** synchronization cost



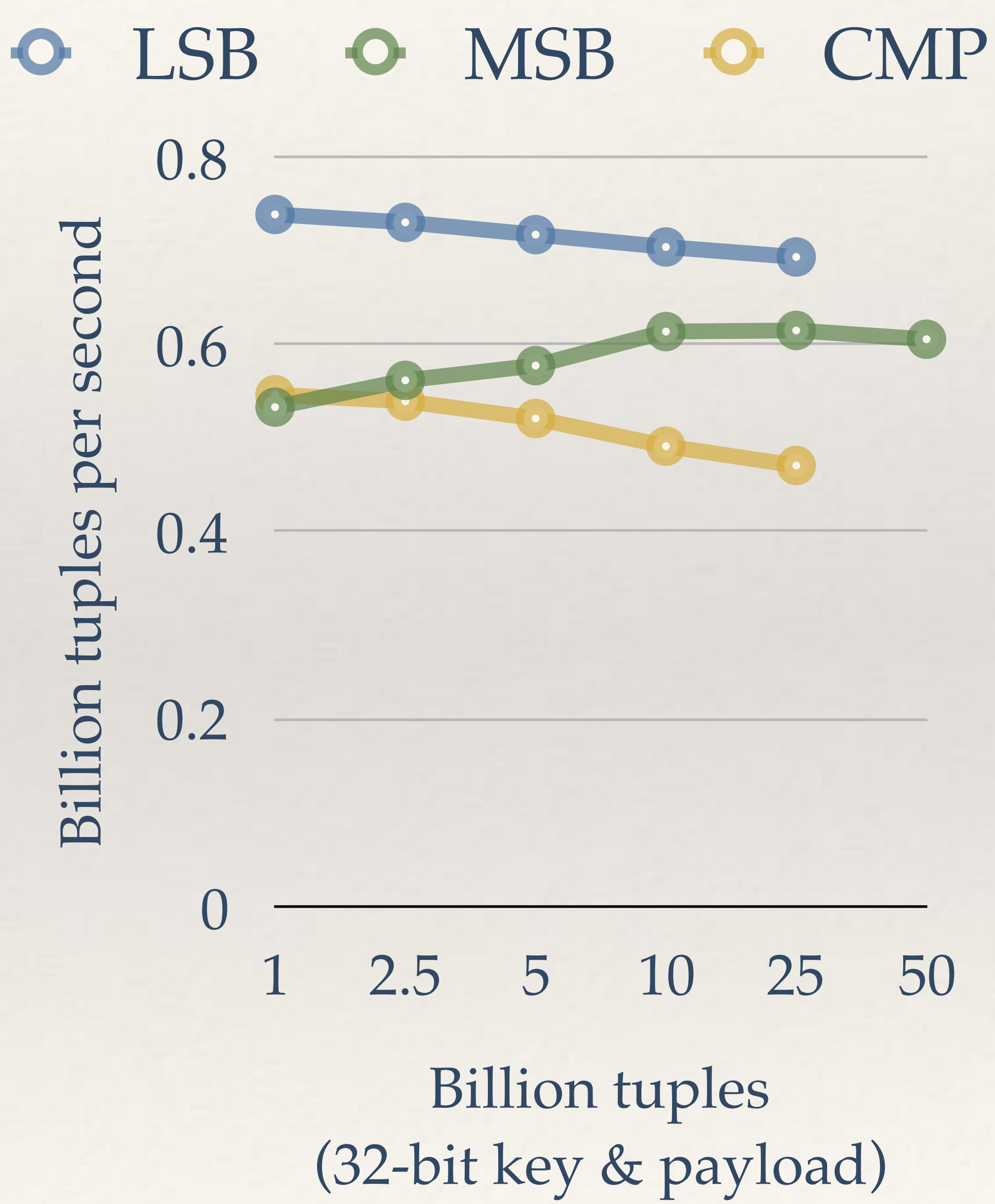
Partitioning Function

- Radix
  - Trivial
- Hash
  - Depends on hash function
- Range
  - **Slow** with binary search
  - **Fast** with range index



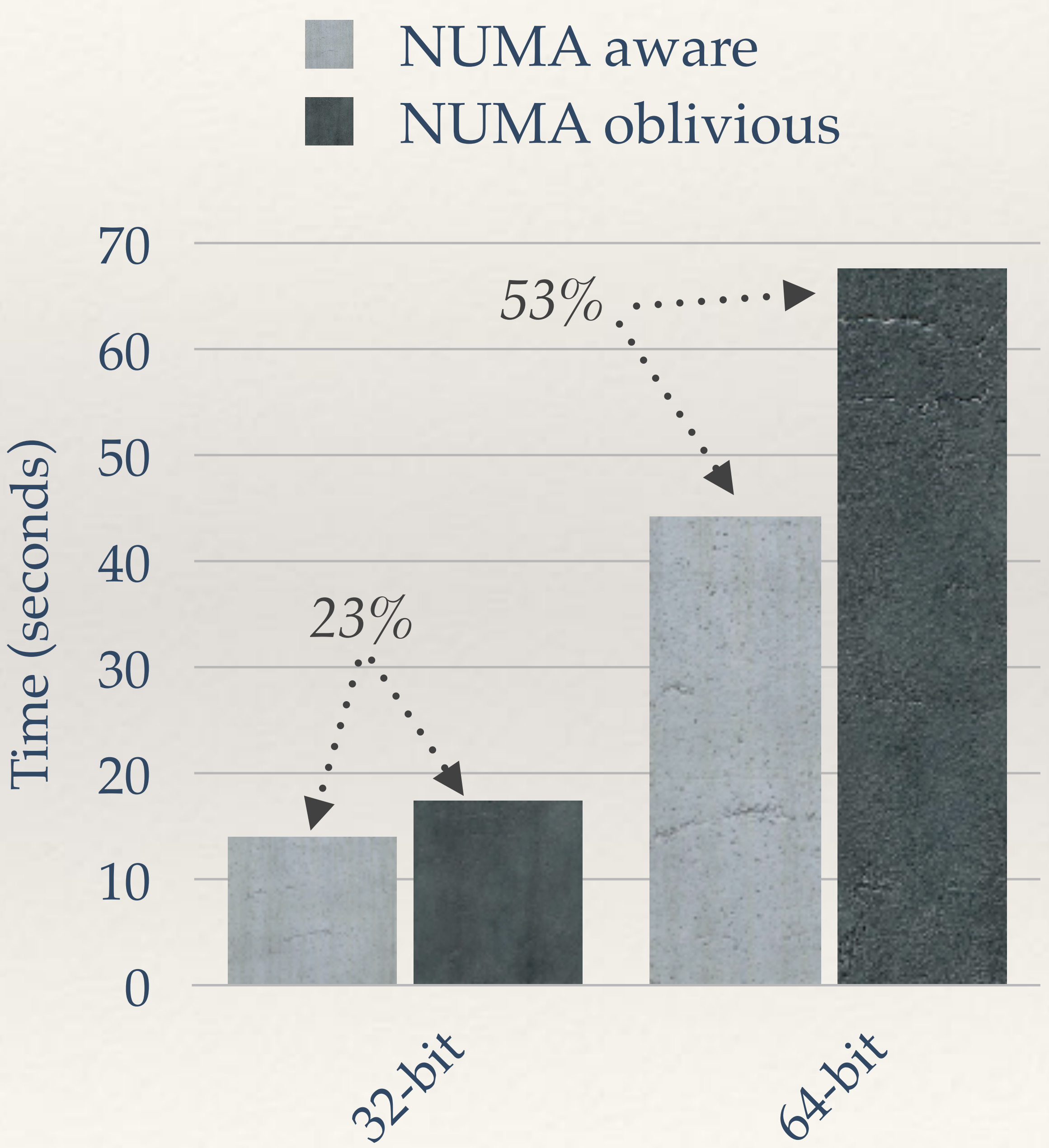
Large-scale Sorting

- Stable LSB radix-sort
  - Parallel radix partitioning (not in-place)
- In-place MSB radix-sort
  - Parallel in-place radix partitioning
  - In-place radix partitioning
- Comparison-sort (CMP)
  - Parallel range partitioning (not in-place)
  - SIMD comb-sort in the cache

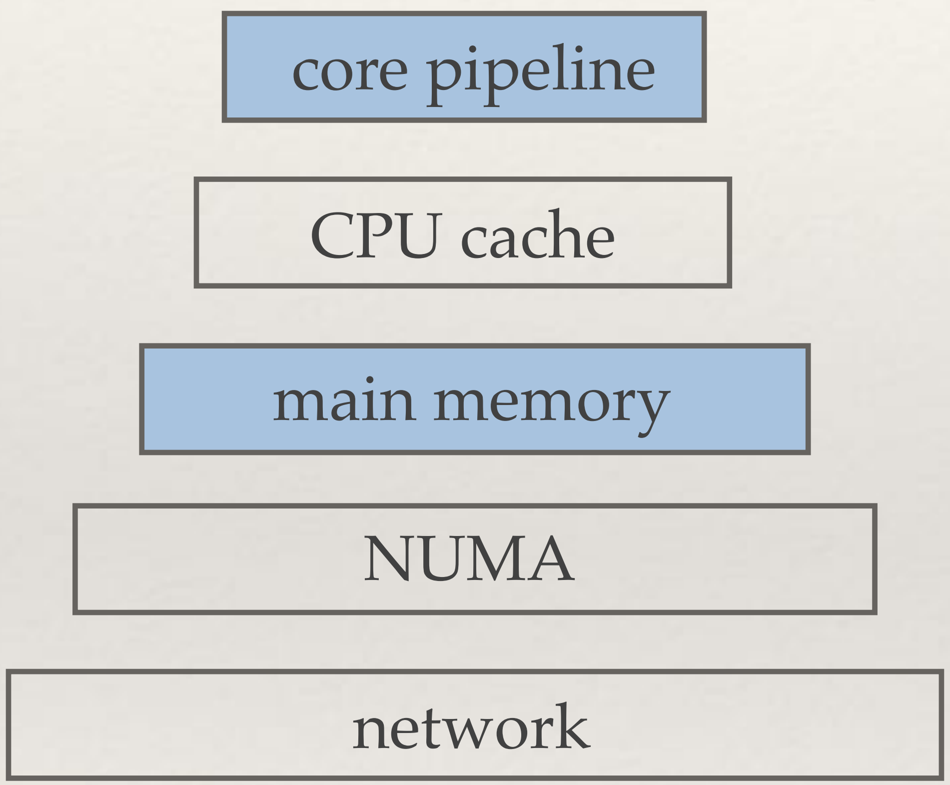


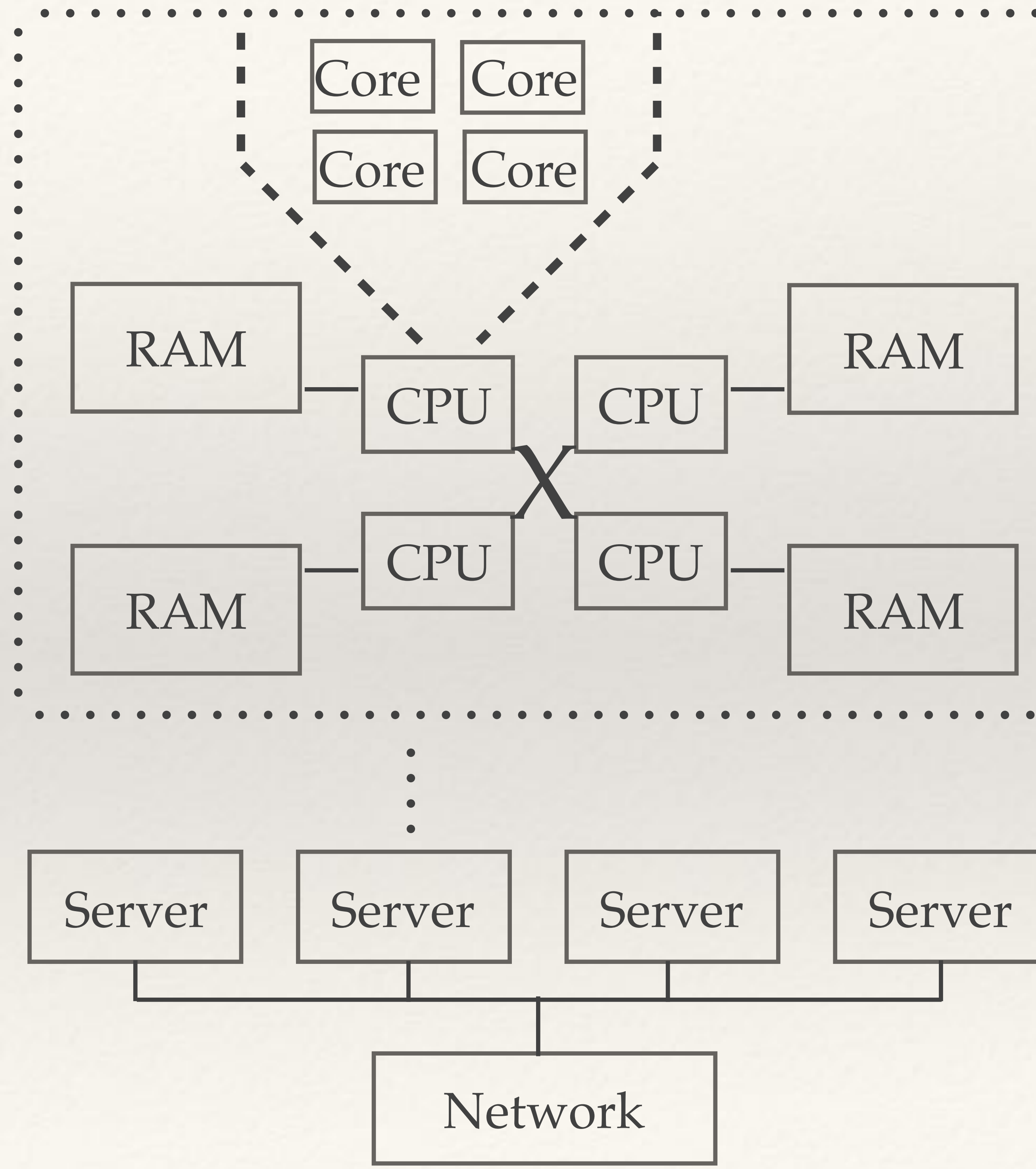
NUMA Awareness

- Optimize for NUMA
  - Use **local** RAM per CPU
  - **Minimize** NUMA transfers
- Transfers per sorting variant
  - LSB: up to 1 transfer
  - MSB: up to 2 transfers
  - CMP: up to 1 transfer



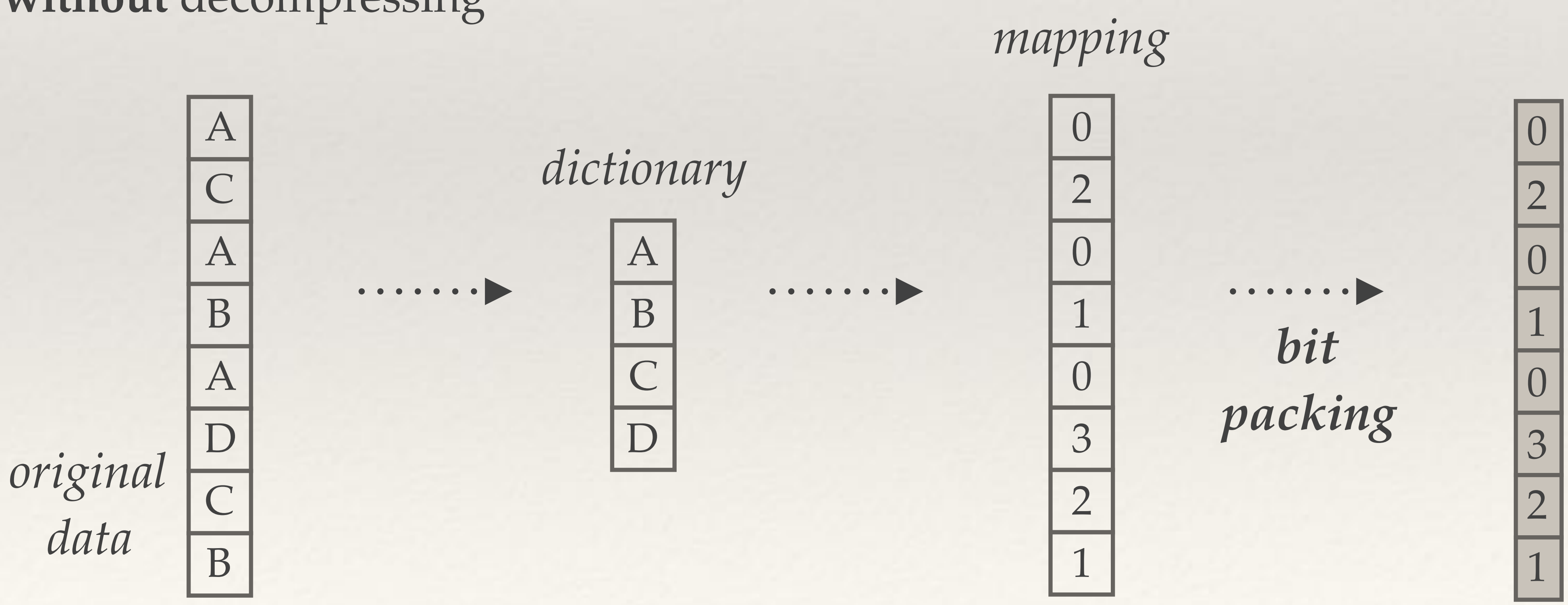
Part 3: Memory-bound (sequential)





Compression in Databases

- Why compress?
  - Make dataset RAM resident
  - Process data **faster** than RAM bandwidth
- **Dictionary** encoding
  - Process **without** decompressing



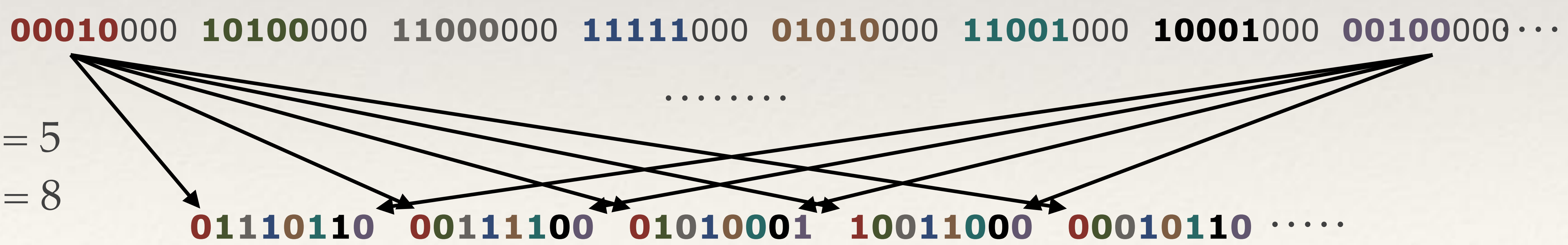
Bit Packing Layouts

- Horizontal Bit Packing

$b = 5$

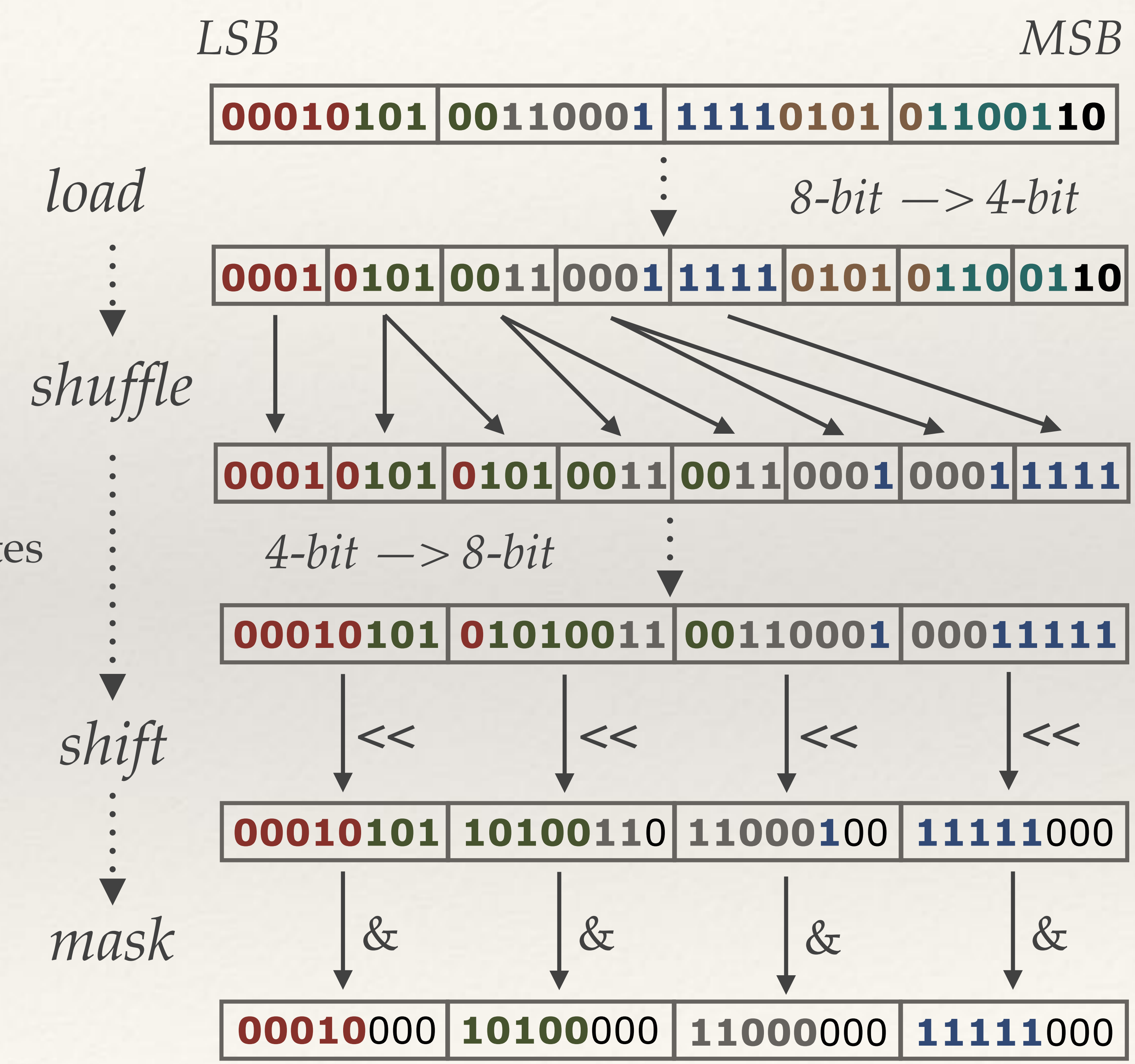
- Vertical Bit Packing

$b = 5$
$k = 8$



Previous Work: Scan Horizontal

- Fully packed
  - No bits wasted
  - Unpack **before** evaluating predicates
  - Unpack in **SIMD**



Previous Work: Scan Horizontal

- Word aligned
  - Scan **without** unpacking
  - Using scalar code
  - Bits **wasted**
  - Parallel bit extraction

select … where column < C …

set constant C

01

010 **01**0 00

invert code

010 **10**0 00 ^ 110 110 00 = **10**0 **01**0 00

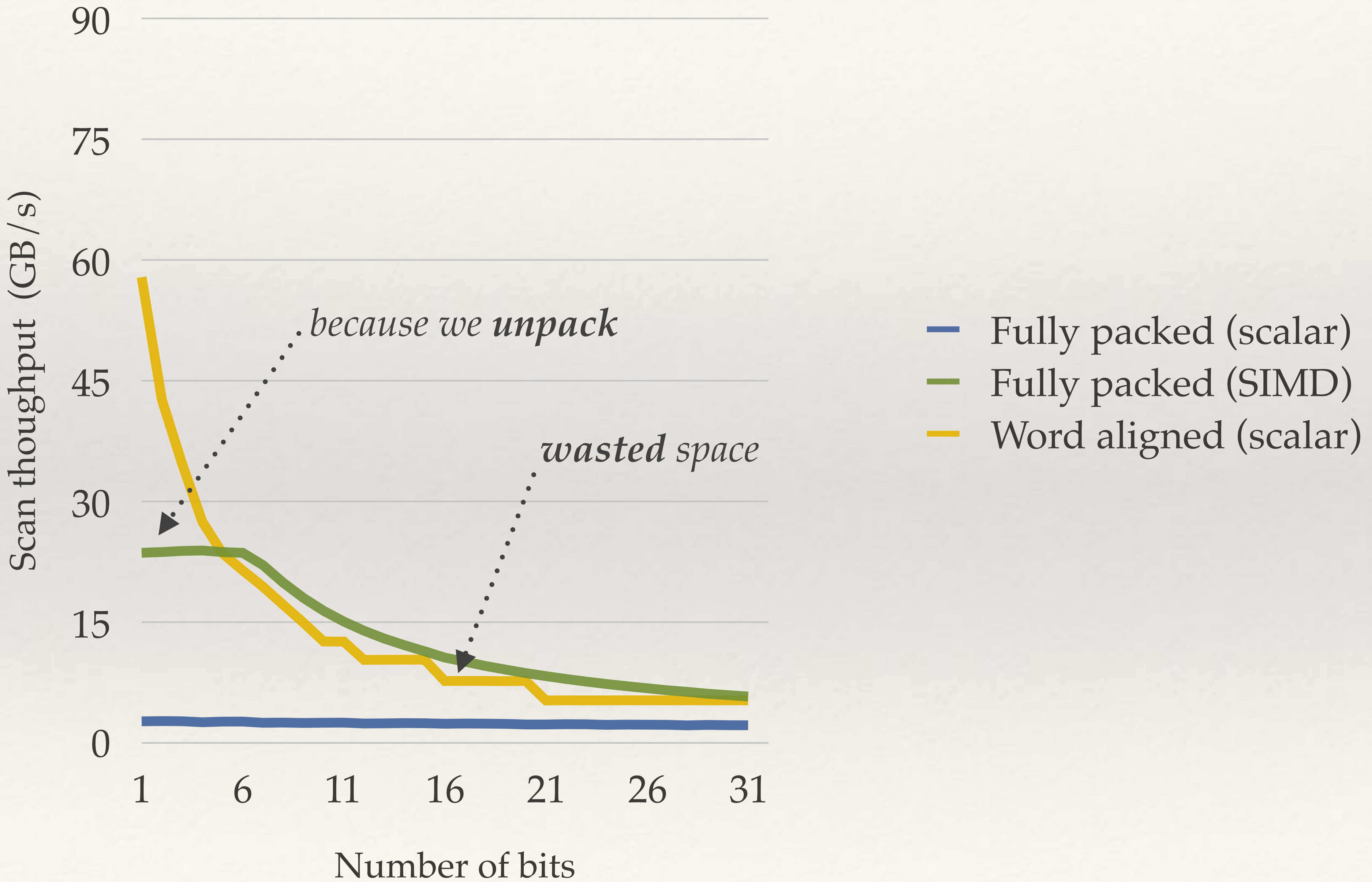
add constant C

100 **01**0 00 + **01**0 **01**0 00 = 11**0** 00**1** 00

extract bits

11**0** 00**1** 00 —> **01**

Previous Work: Scan Horizontal



Previous Work: Scan Vertical

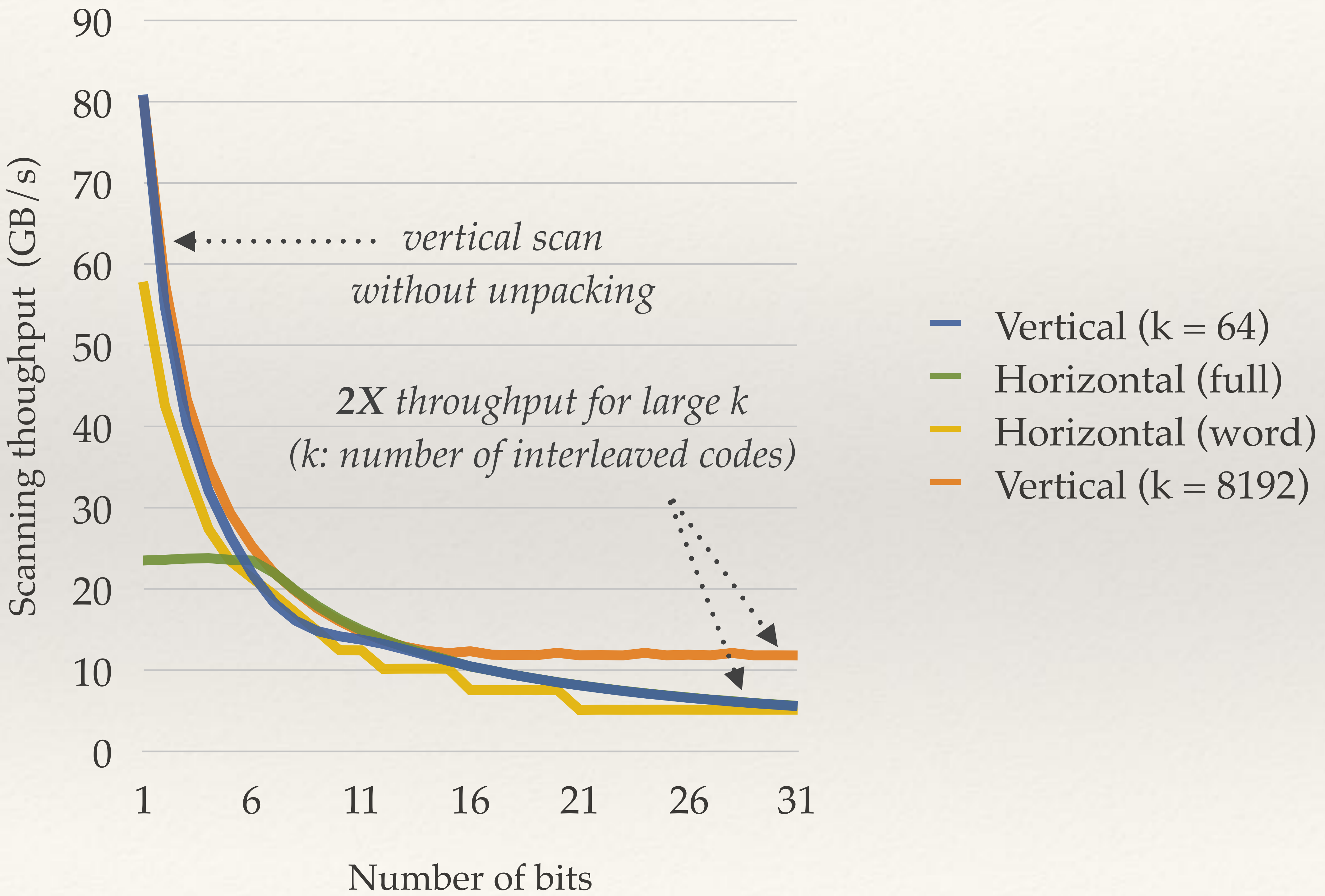
- Scan **without** unpacking
  - Bit-wise operations
  - Both scalar & SIMD
  - Can skip bits and **stop early**

select … where column < C …

result |= constant & ~data

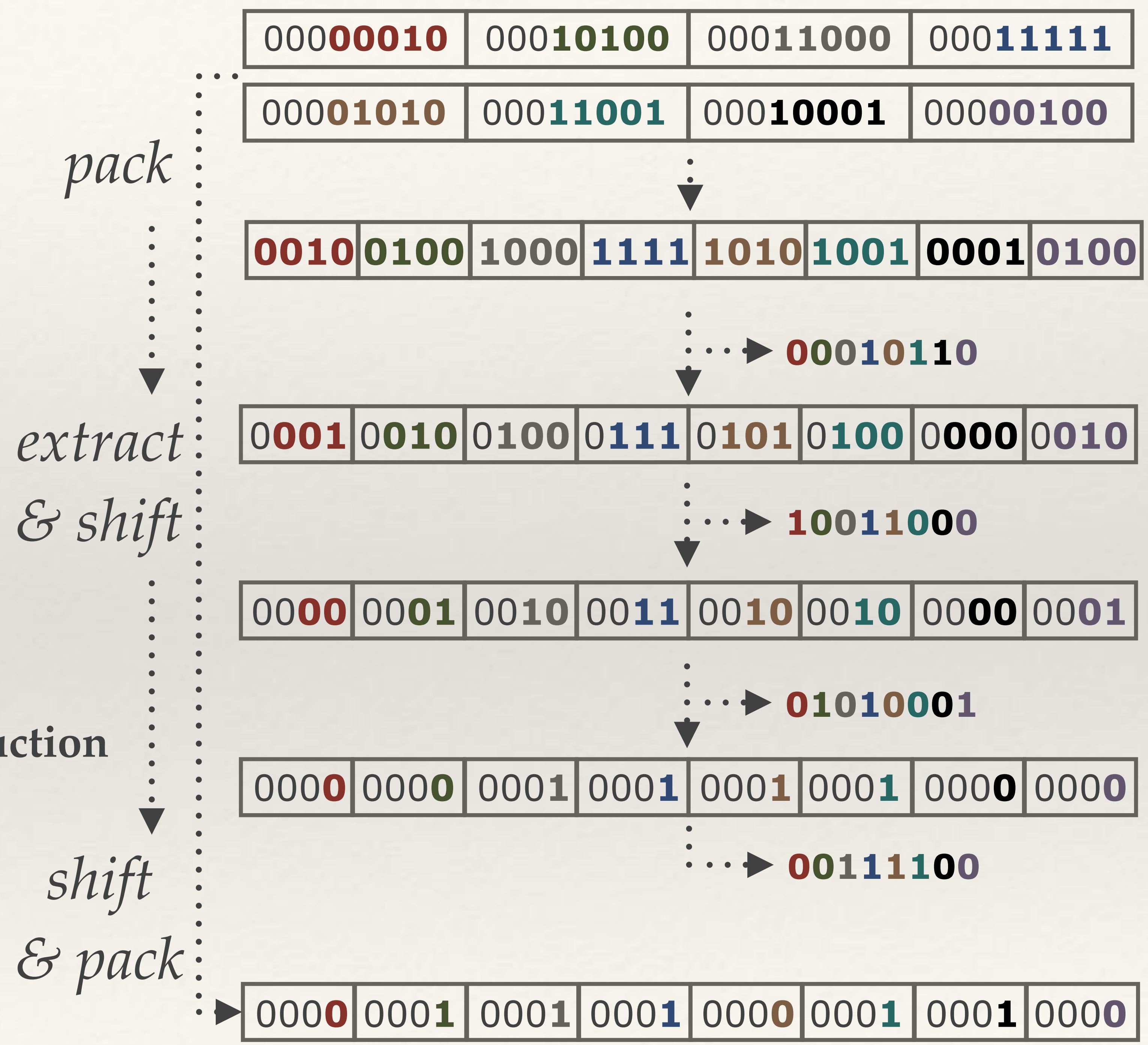
00010110	00000000	___0_00_
10011000	11111111	_110_001
01010001	11111111	11101001
00111100	00000000	*result*
01110110	00000000	
data	*constant*	

Previous Work: Scan Vertical



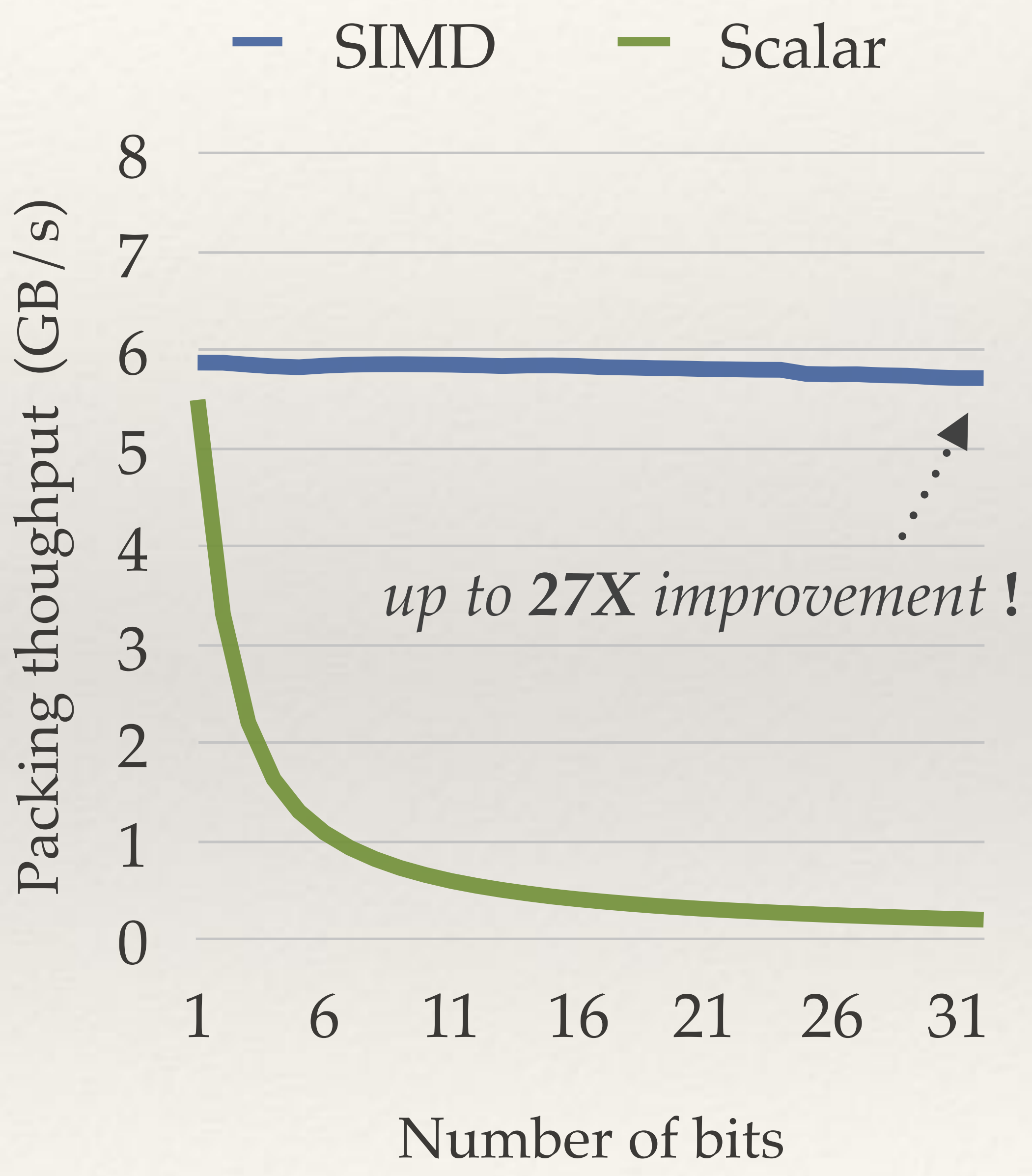
Pack Vertical Layout

- Scalar
  - **Extract** 1 bit per instruction
  - O(n * b)
- SIMD
  - Keep codes in SIMD registers
  - Maximize **bits per SIMD instruction**



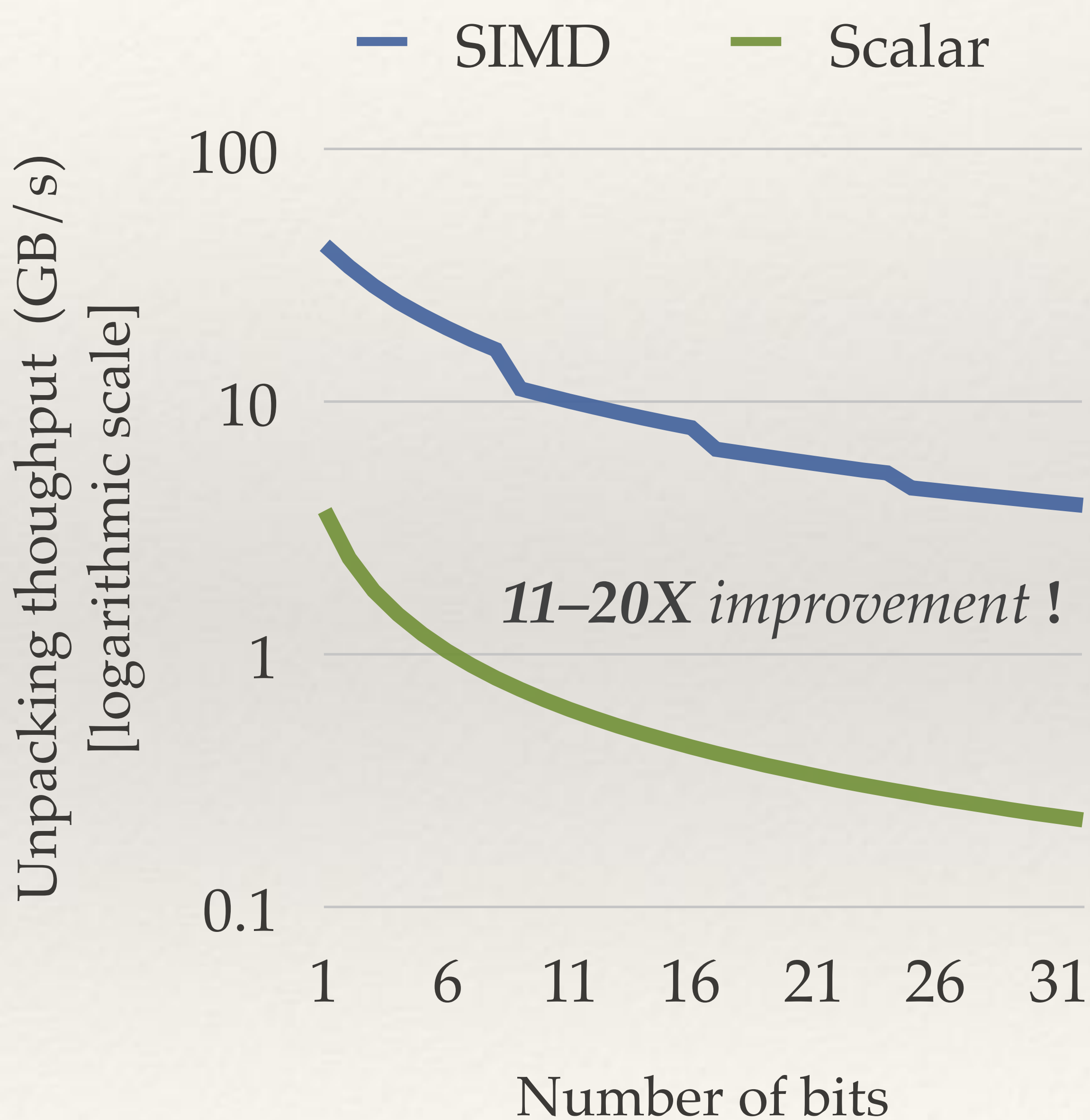
Pack Vertical Layout

- Scalar
  - **Extract** 1 bit per instruction
  - O(n * b)
- SIMD
  - Keep codes in SIMD registers
  - Maximize **bits per SIMD instruction**



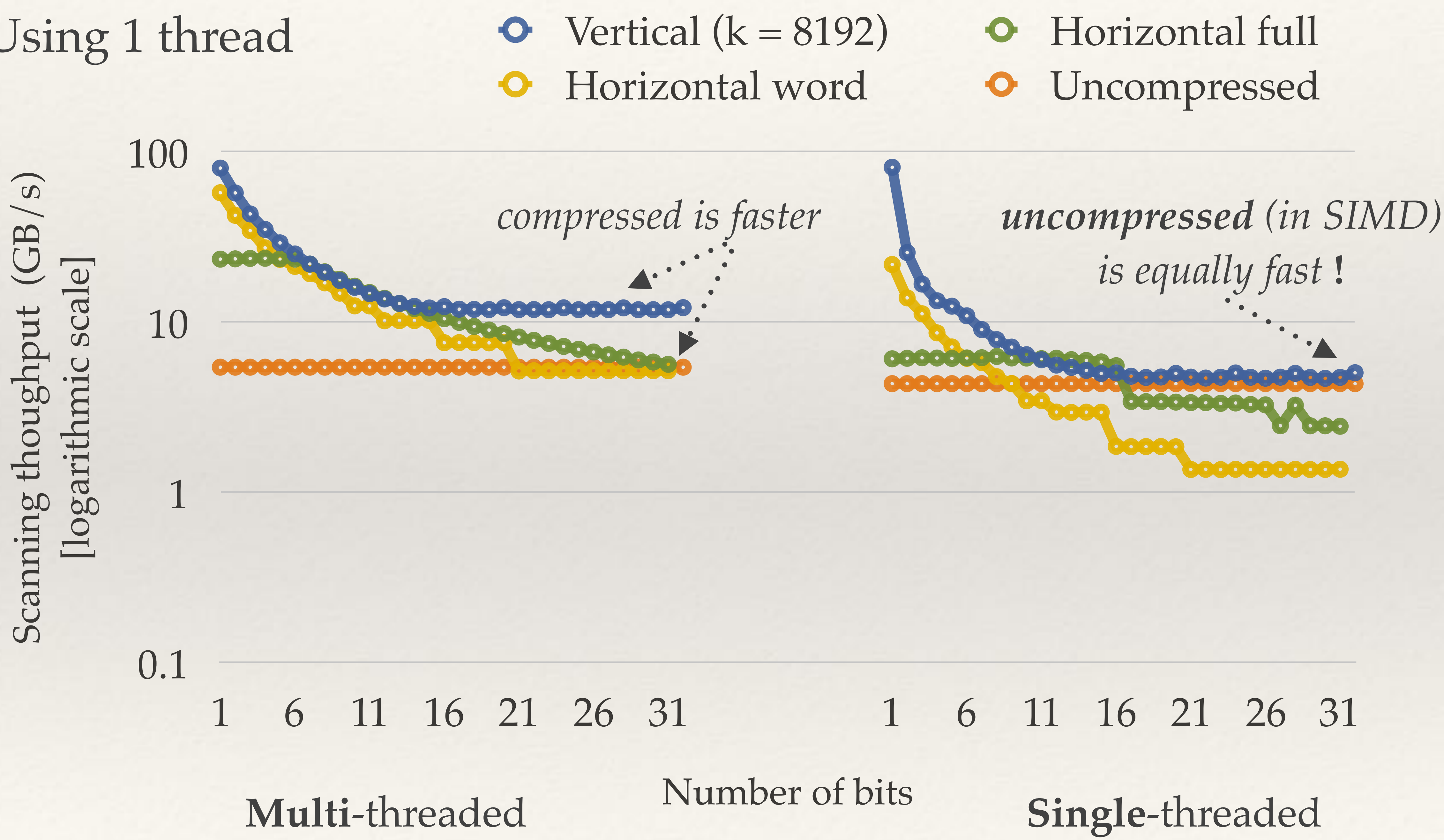
Unpack Vertical Layout

- Scalar
  - **Insert** 1 bit per instruction
  - O(n * b)
- SIMD
  - Keep codes in SIMD registers
  - Maximize **bits per SIMD instruction**

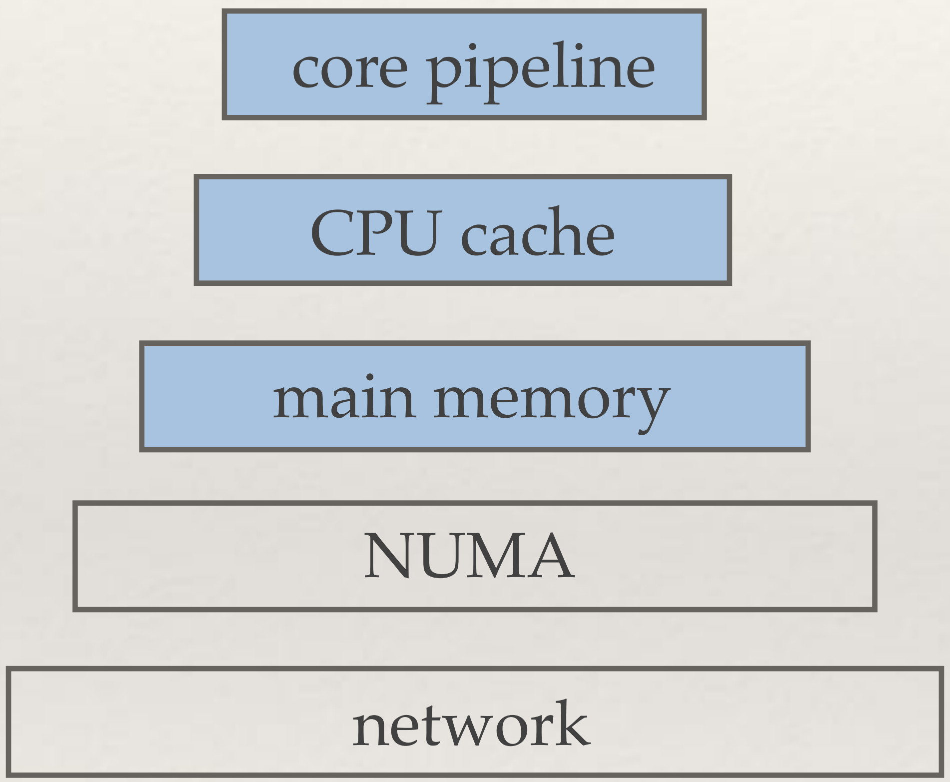


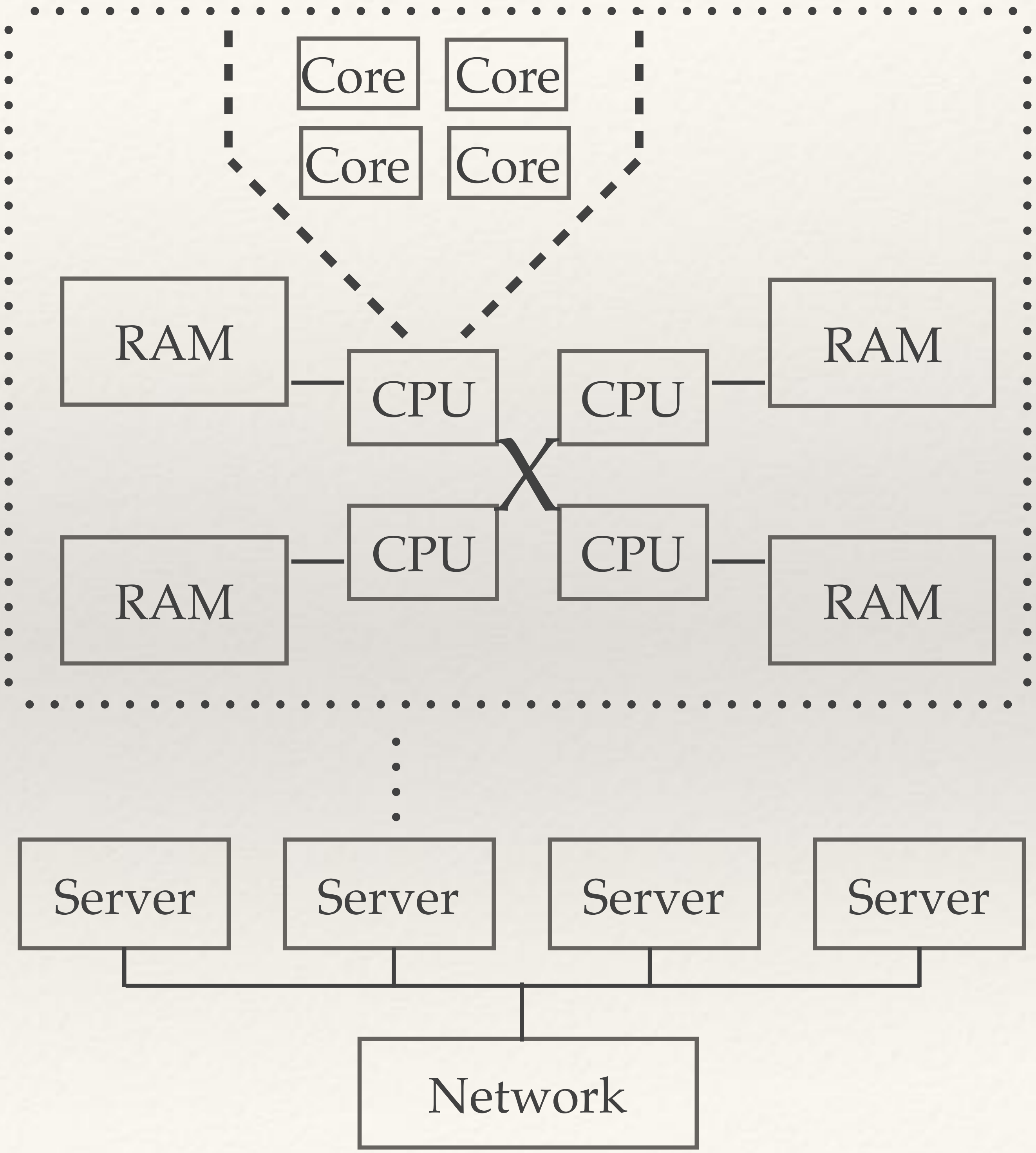
What if not memory-bound?

- Using 1 thread



Part 4: Compute-bound

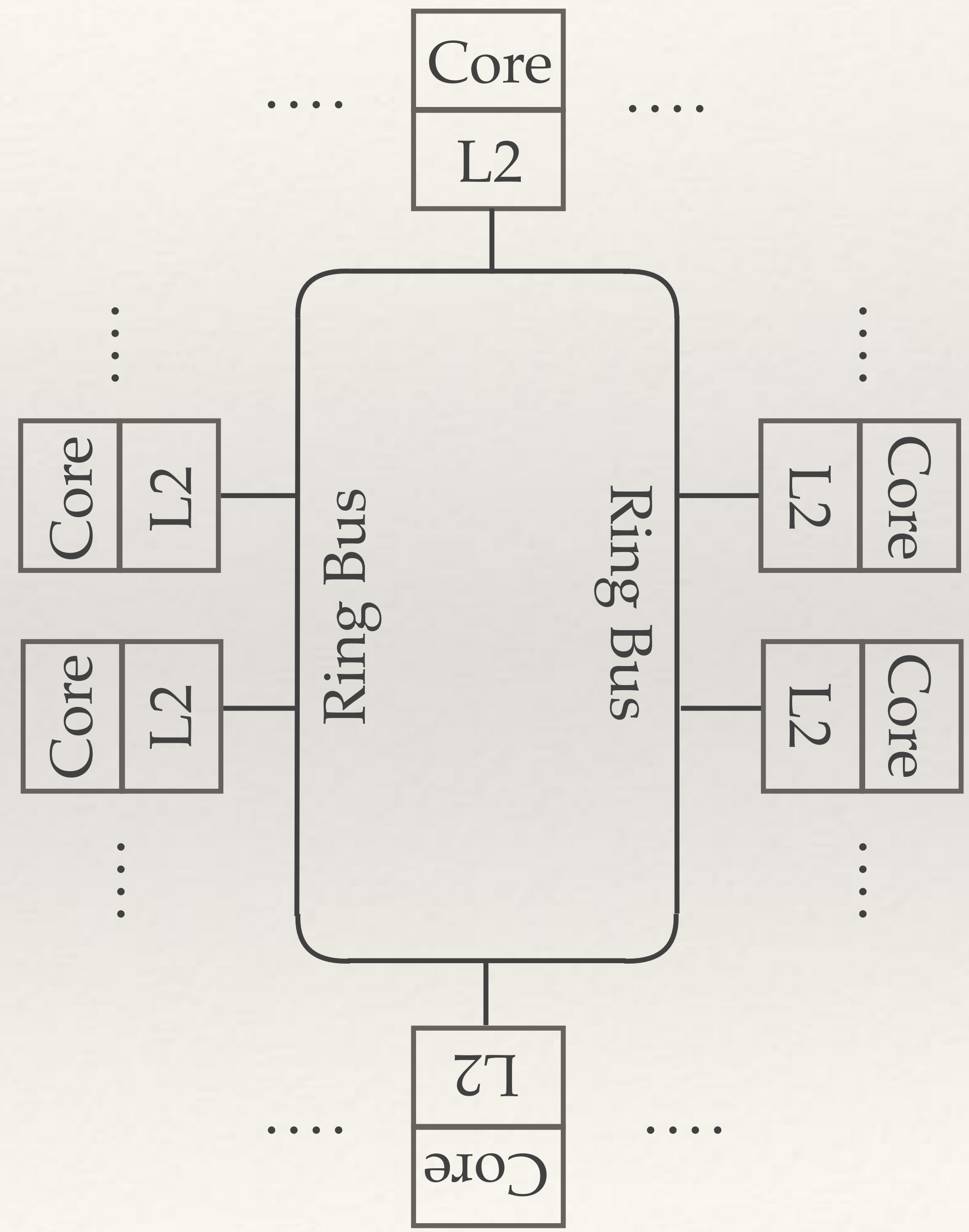




Many-Core (MIC) Platforms

- Mainstream CPUs
  - Aggressively **out-of-order**
  - Massively **super-scalar**
  - ~20 cores ($$$$)
- **Many-core** co-processors
  - 1st generation
  - In-order
  - Not super-scalar
  - 16 GB of fast RAM
  - ~60 cores

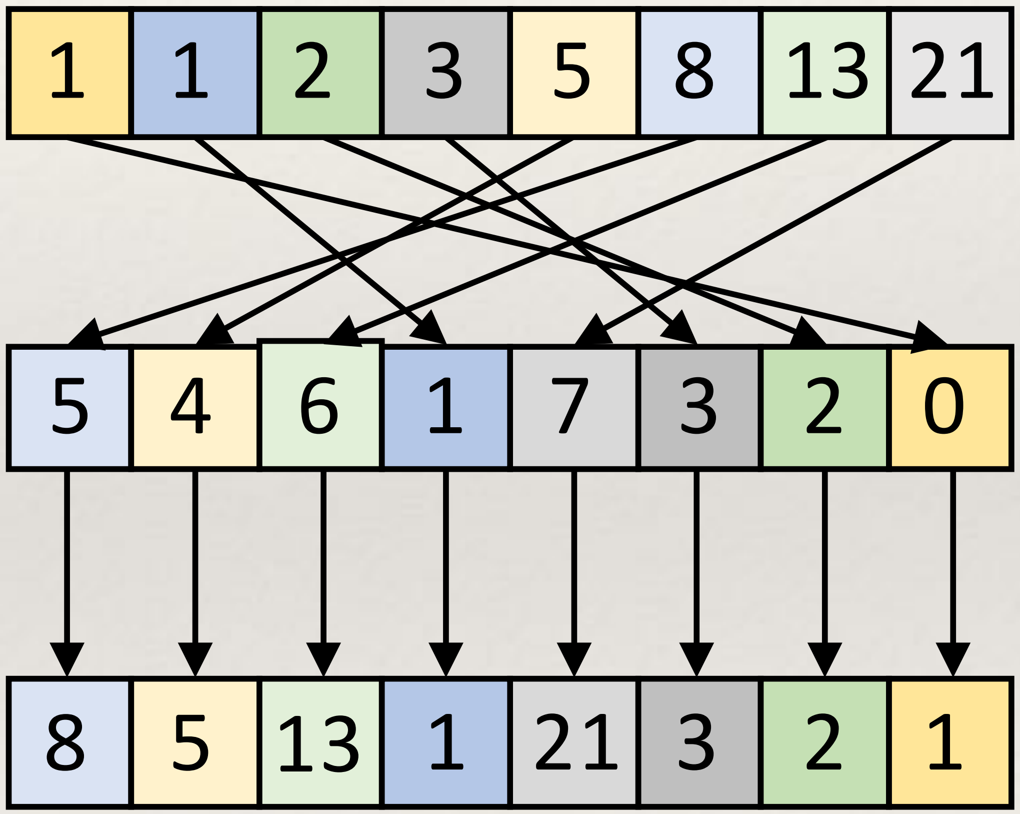
~ *60* cores



Advanced SIMD Vectorization

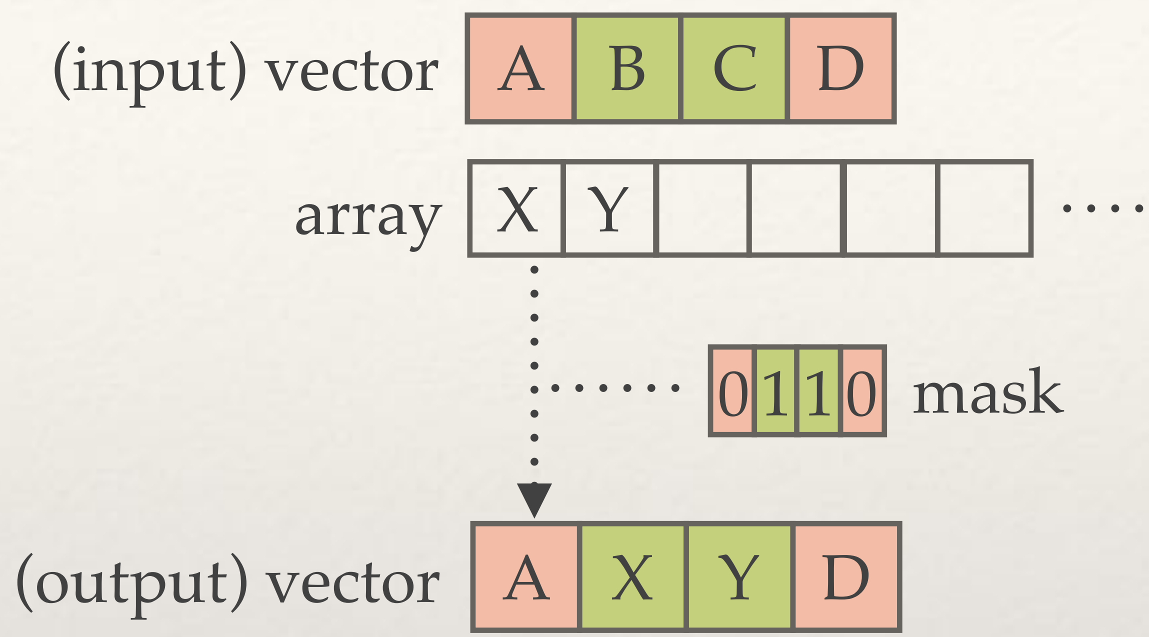
- Baseline operator
  - O(f(n)) complexity in scalar code
- **Fully** vectorized
  - O(f(n) / **W**) complexity in SIMD code
  - Excluding **random** memory accesses
- **Reusable** vectorization techniques
  - Reuse fundamental operations

*8-way **SIMD** permutation*

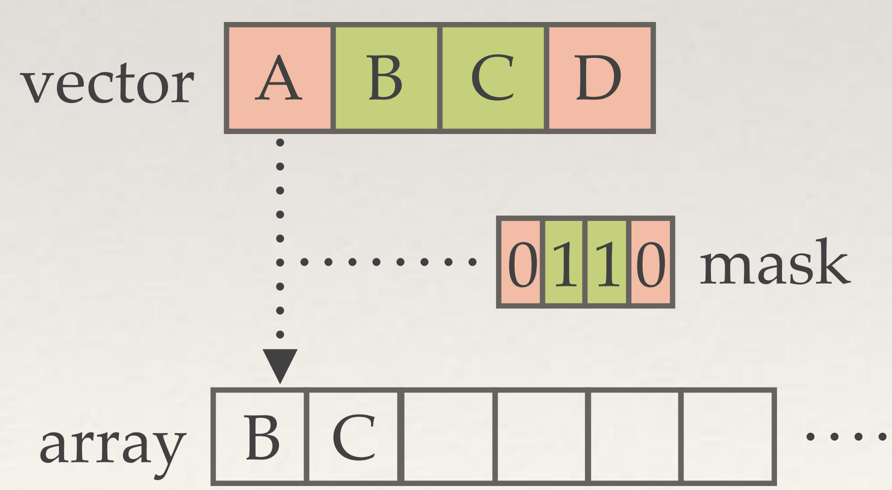


Fundamental Operations

- Selective load

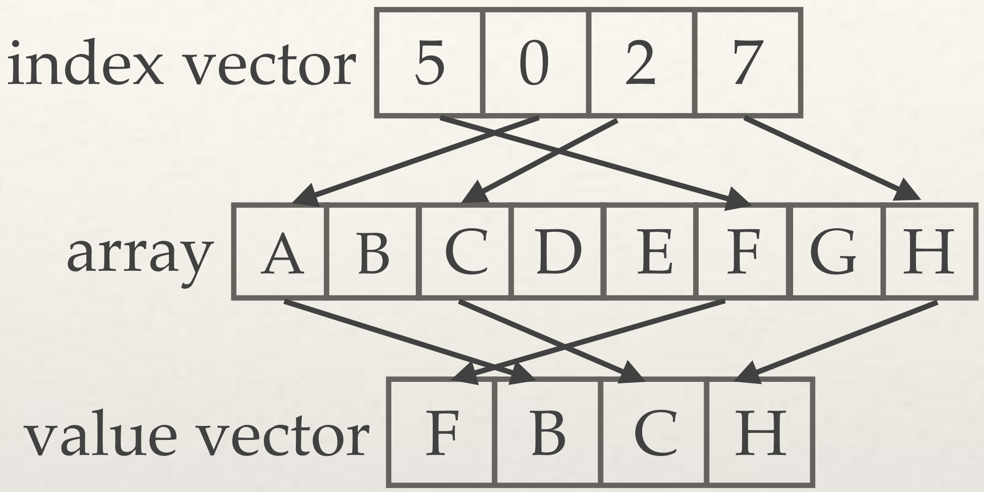


- Selective store

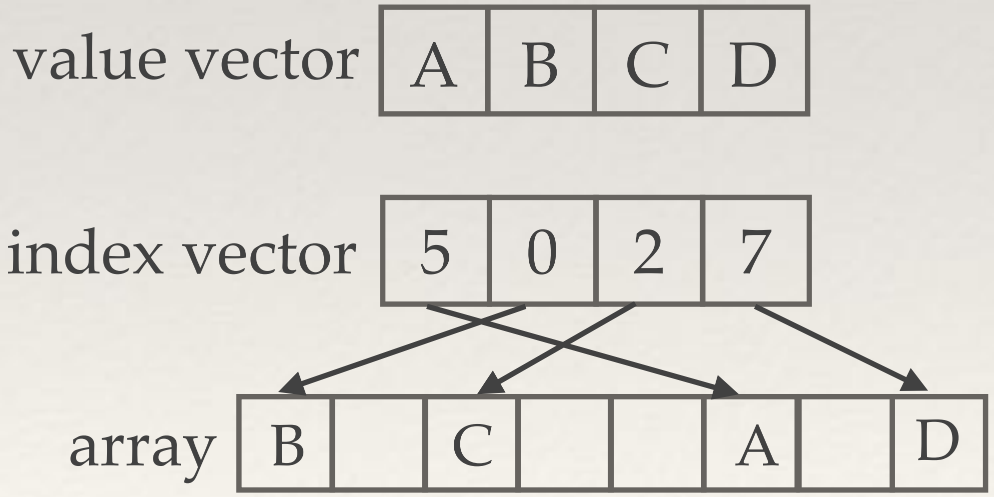


Fundamental Operations

- (Selective) gather

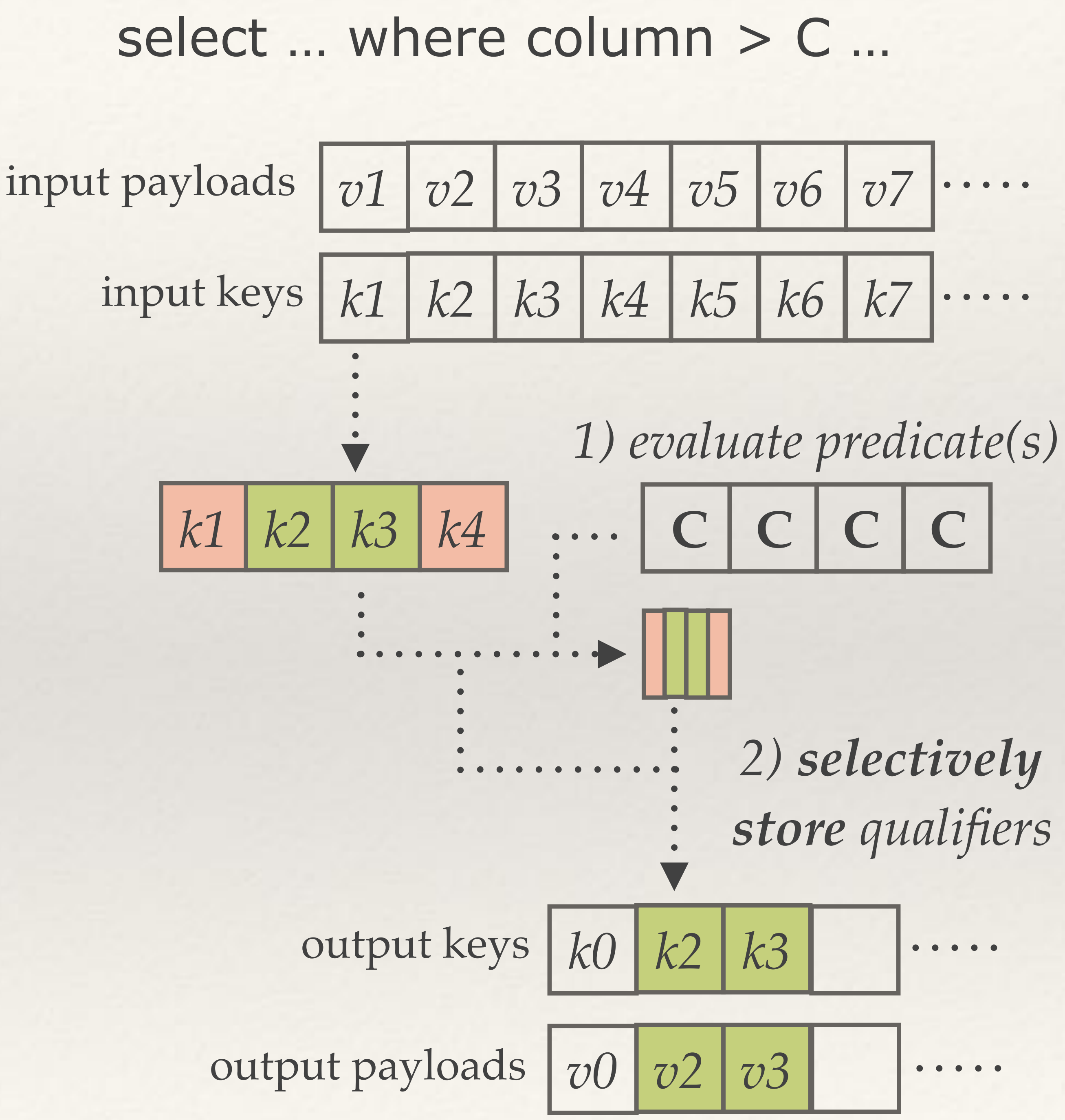


- (Selective) scatter

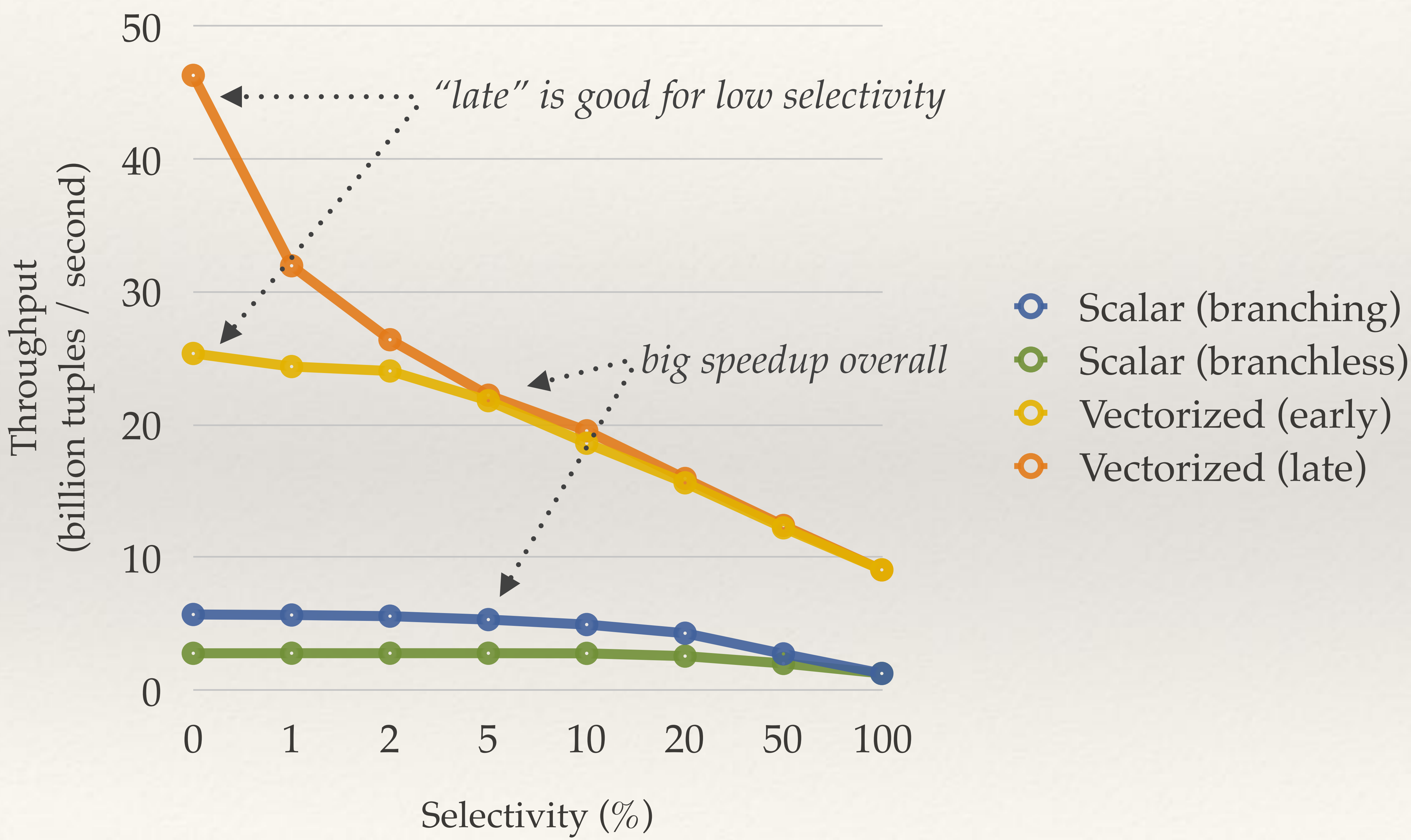


Vectorized Selection Scans

- Scalar
  - Branching
  - Branchless
- Vectorized
  - Using **selective stores**
  - Store tuples (early)
  - Store rids & dereference (late)

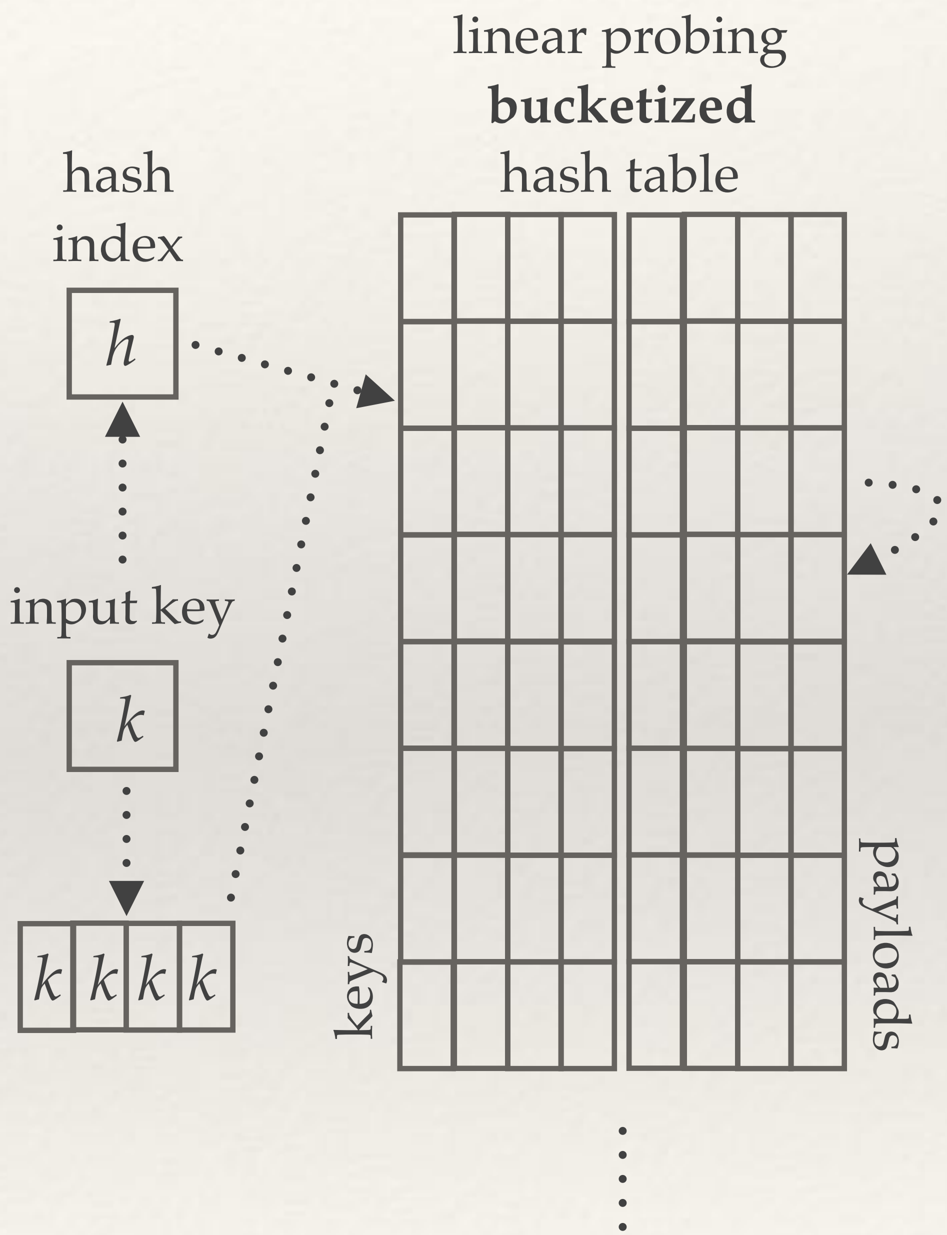


Vectorized Selection Scans



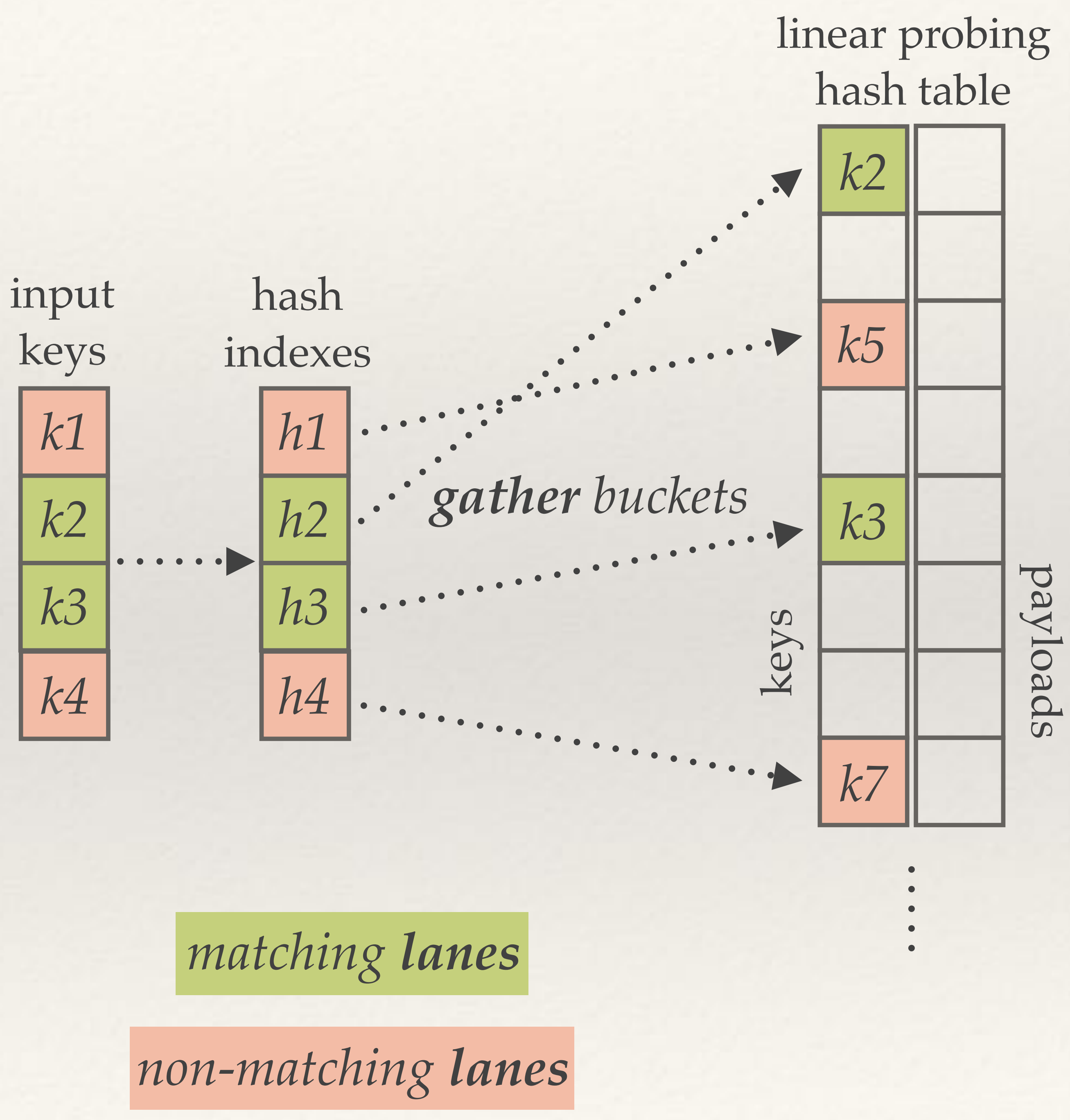
Previous Work: Vectorized Hash Probing

- Scalar
  - 1 input key at a time
  - 1 table key per input key
- **Horizontal** vectorization
  - 1 input keys at a time
  - W table keys per input key



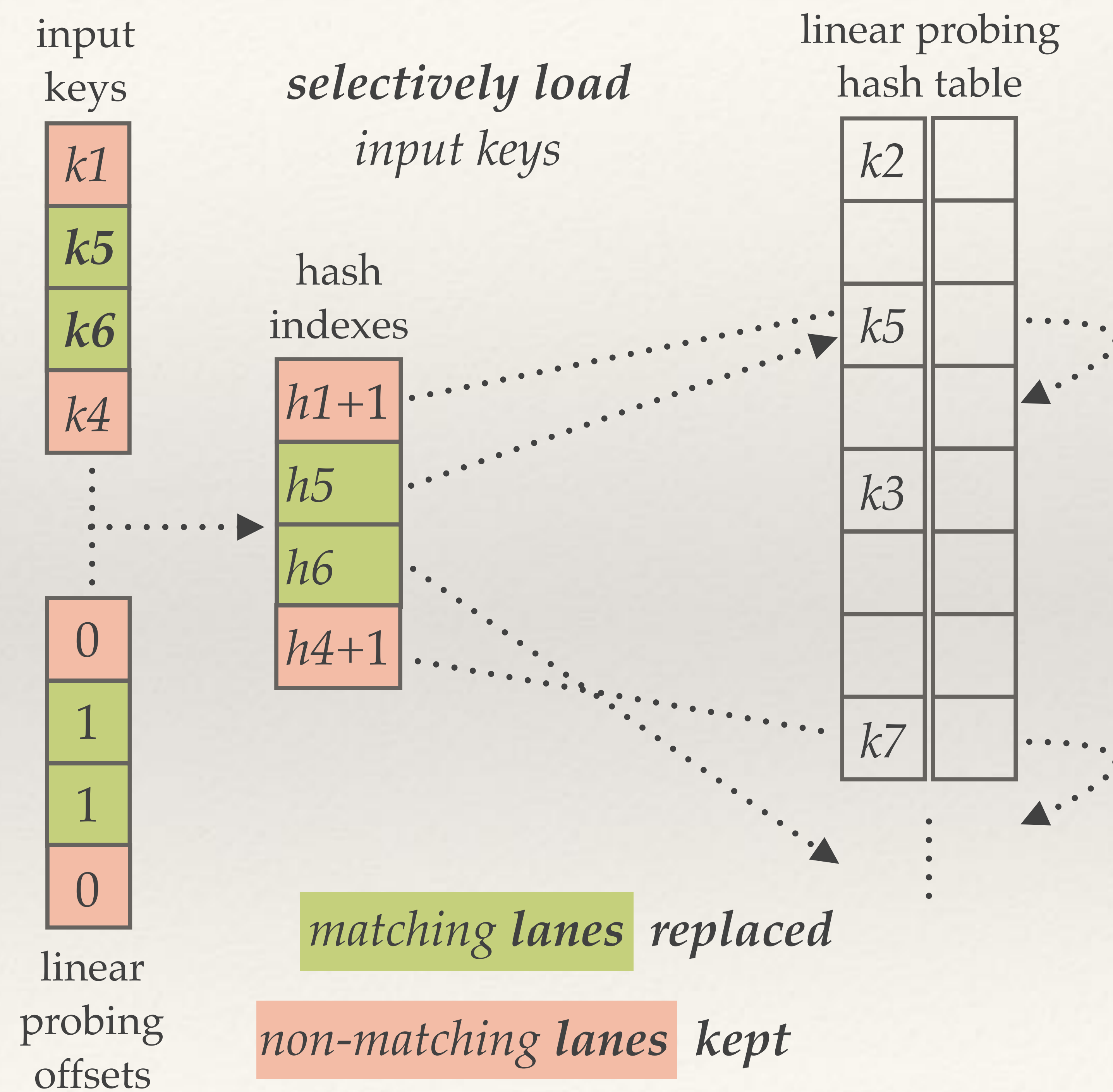
Vectorized Hash Probing

- **Vertical** vectorization
  - W input keys at a time
  - 1 table keys per input key

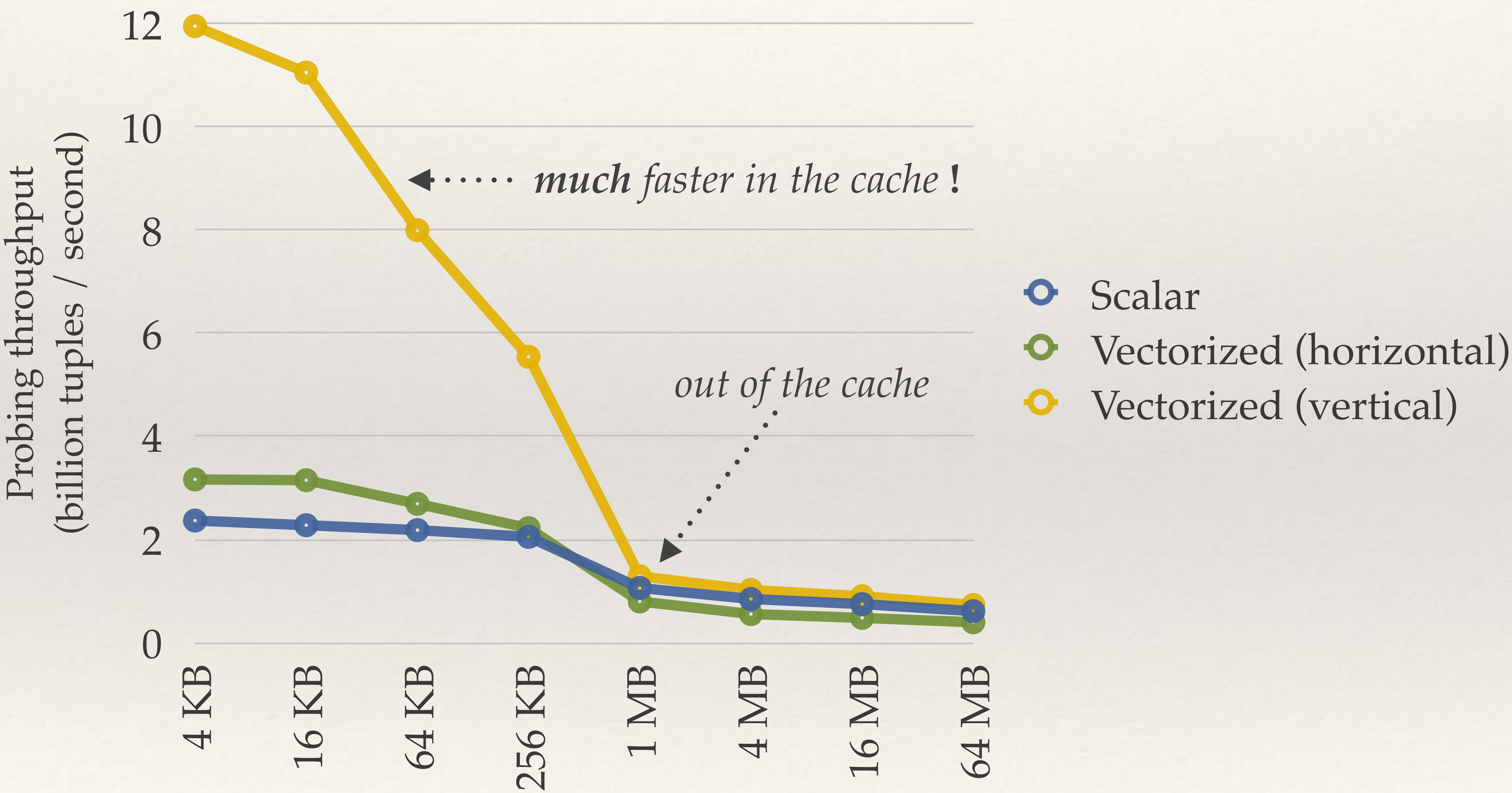


Vectorized Hash Probing

- **Vertical** vectorization
  - W input keys at a time
  - 1 table keys per input key

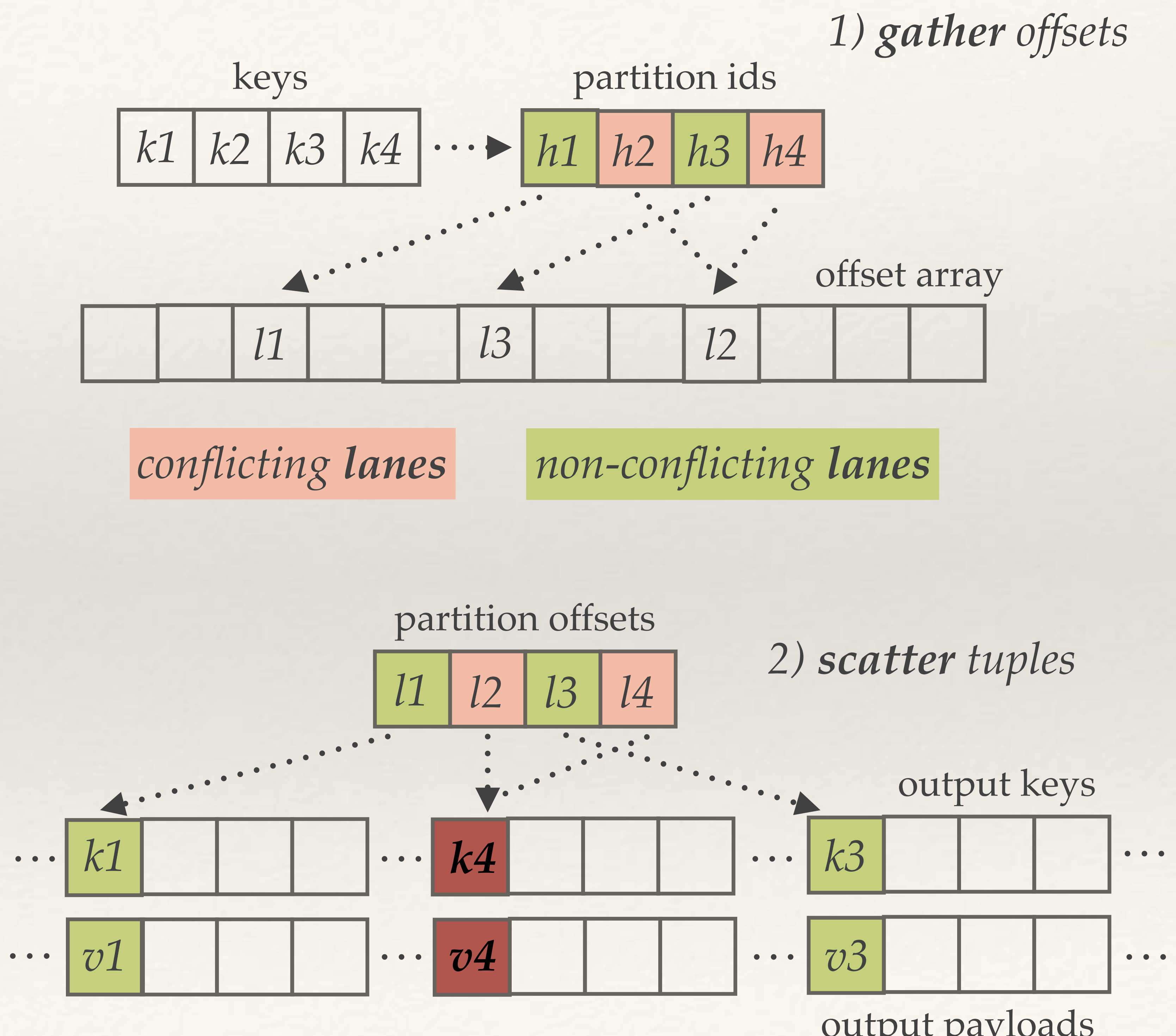


Vectorized Hash Table Probing



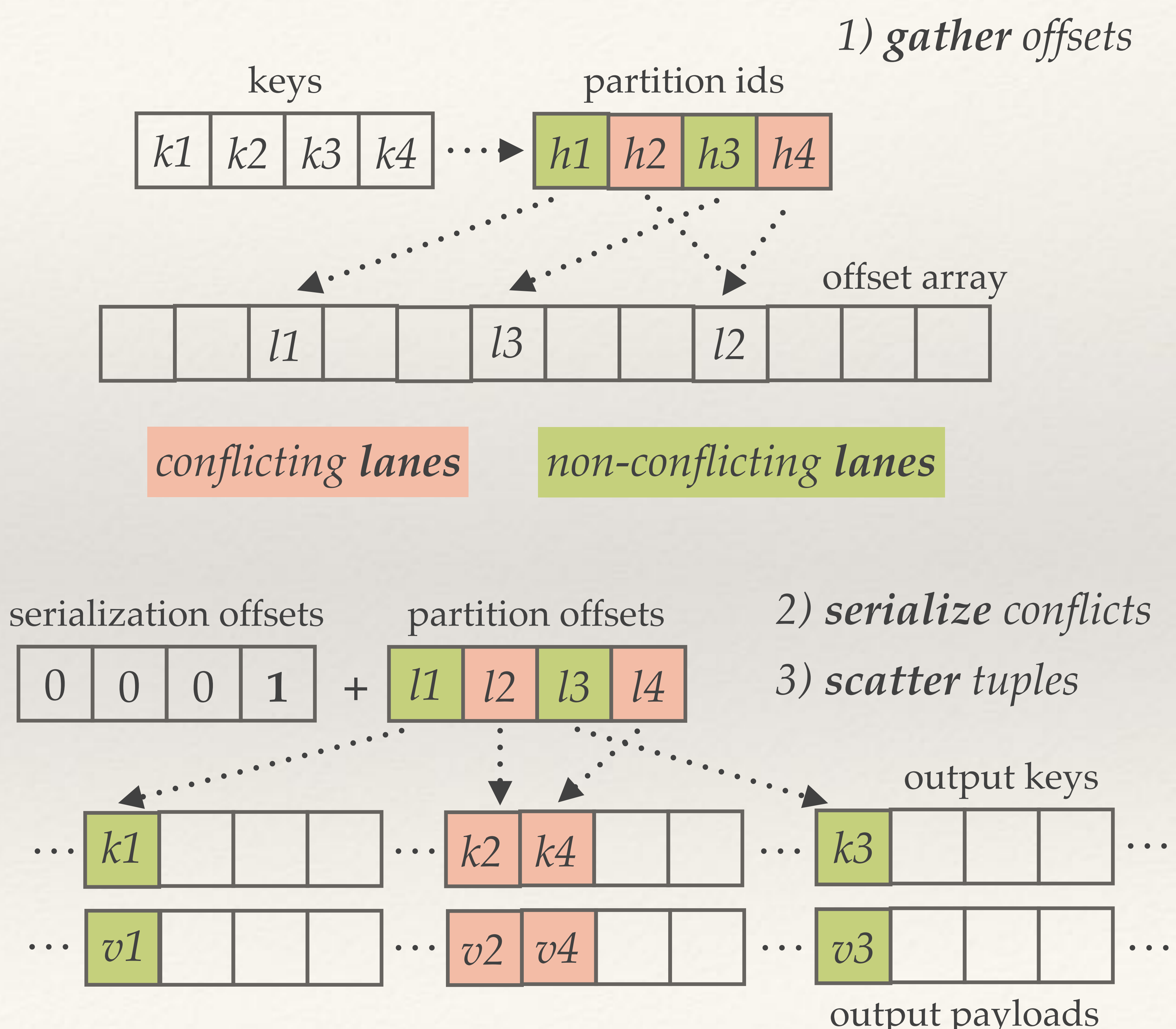
Vectorized Data Shuffling

- Scalar
  - Move 1 tuple at a time
- Vectorized
  - **Scatter** tuples to output
  - **Serialize** conflicts



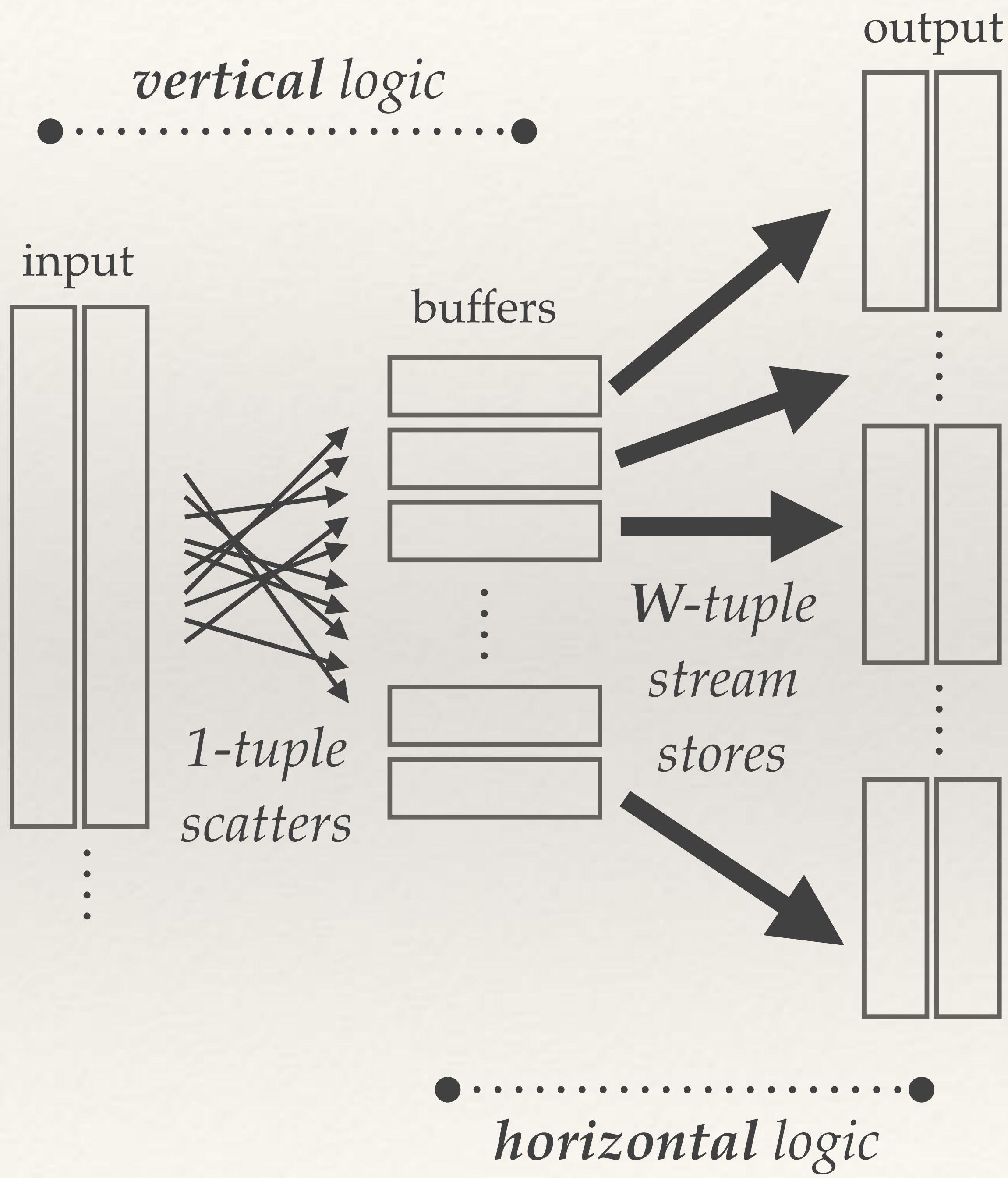
Vectorized Data Shuffling

- Scalar
  - Move 1 tuple at a time
- Vectorized
  - **Scatter** tuples to output
  - **Serialize** conflicts

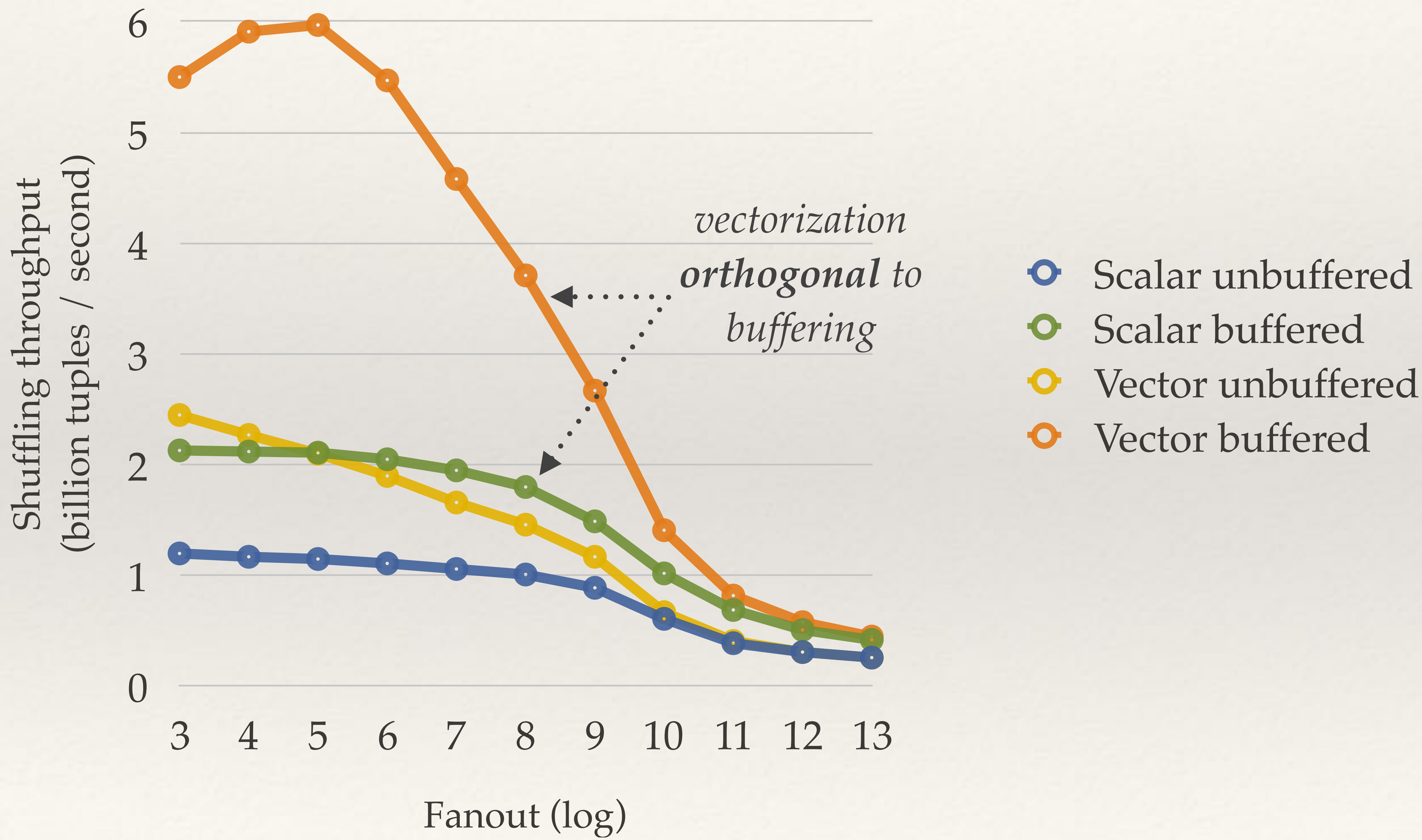


Vectorized Buffered Data Shuffling

- Scalar
  - Move 1 tuple at a time
- Vectorized
  - **Scatter** tuples to **buffers**
  - **Serialize** conflicts



Vectorized Partitioning



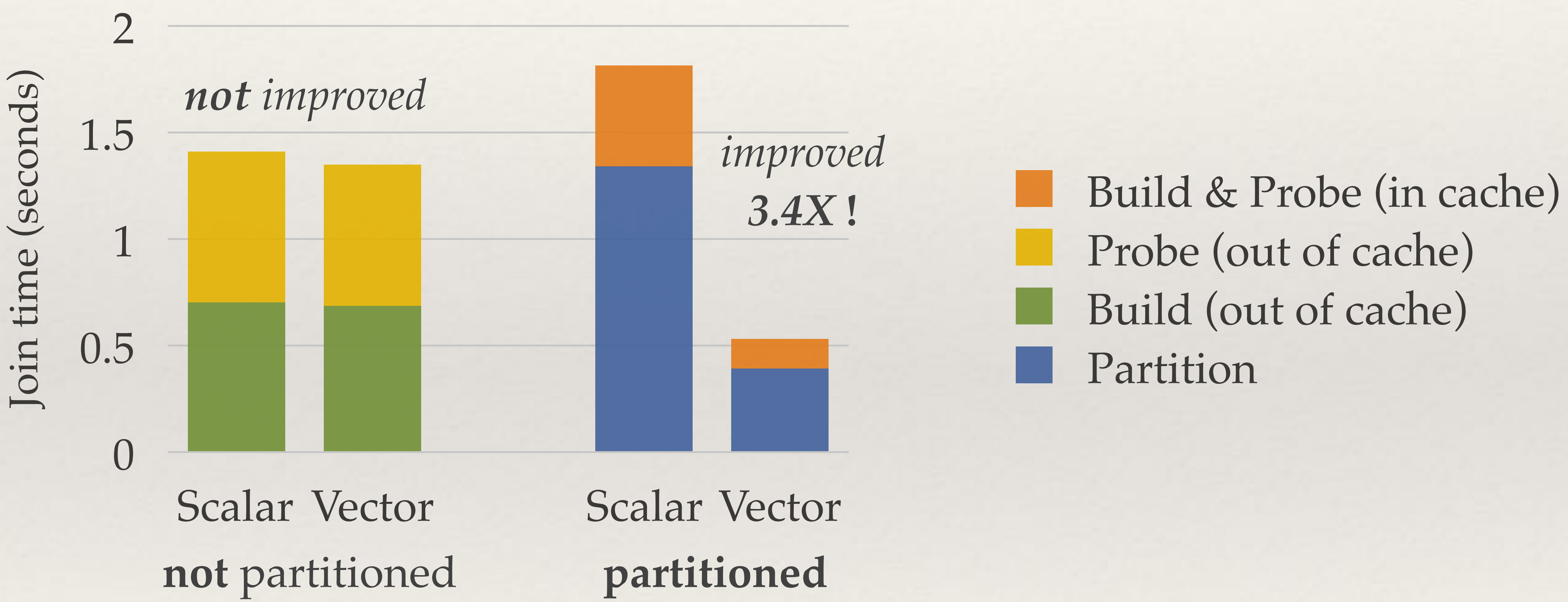
Vectorized Operators

- Selection scans
- Partitioning
  - Histogram
  - Data shuffling
- Hash table building & probing
  - Linear probing
  - Double hashing
  - Cuckoo hashing
- Bloom filter probing
- Regular expression matching

- Sorting
  - LSB radix-sort
- Hash joins
  - Non-partitioned
  - Partitioned

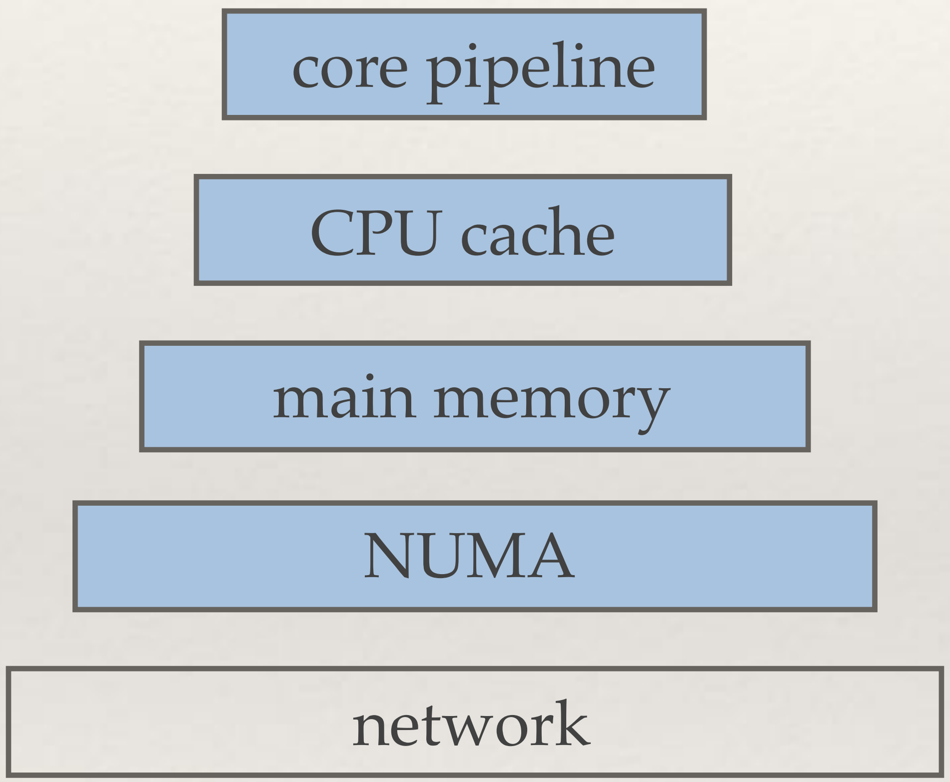
Hash Join Performance

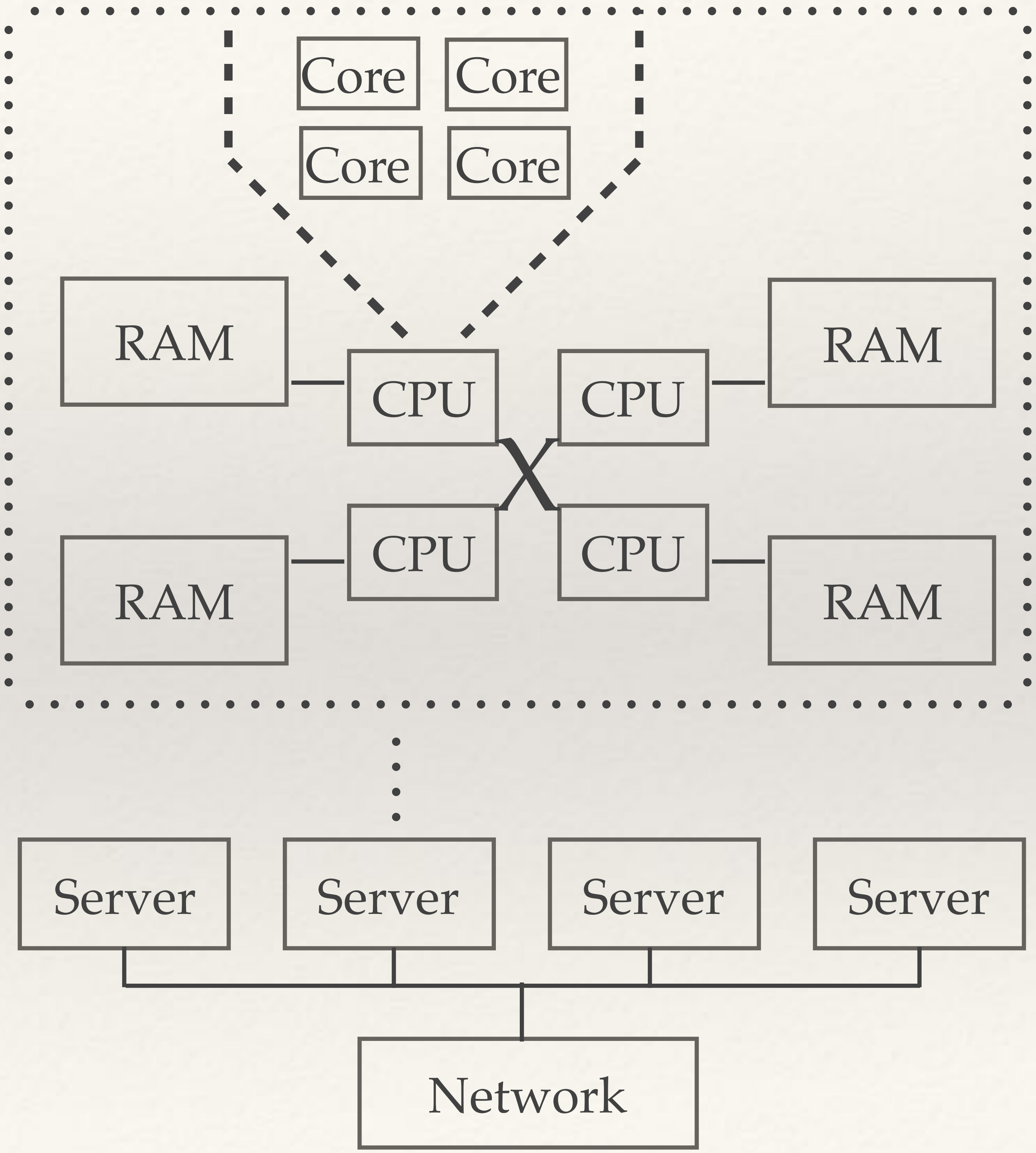
- Hash **join** 200 with 200 million tuples (2X 32-bit key & payload)



- Being **cache-conscious** matters !

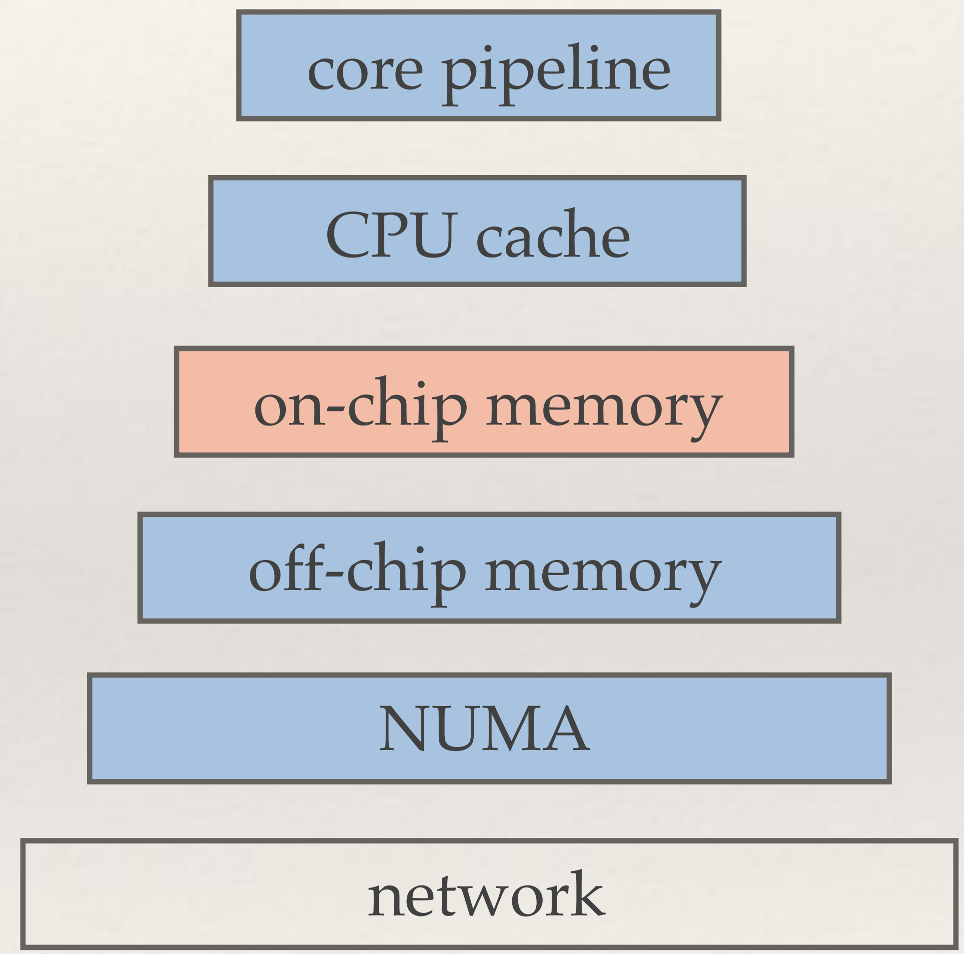
Part 5: An Engine for Many-Cores





Why many-core CPUs?

- More **complex** cores
  - Super-scalar out-of-order cores
  - Core size: 1st-gen << 2nd-gen << mainstream
- Additional layer of on-chip ***MCDRAM***
  - **~4X** higher **bandwidth** than DDR4 DRAM
  - **Larger** than the caches (16 GB)
- Advanced **SIMD**: AVX-512
  - **Same** as upcoming mainstream CPUs



Baseline: Code Generation

- Code generation
  - Generate code **per query** at runtime
  - **Pipelined** operators
  - **Specialized** data structures

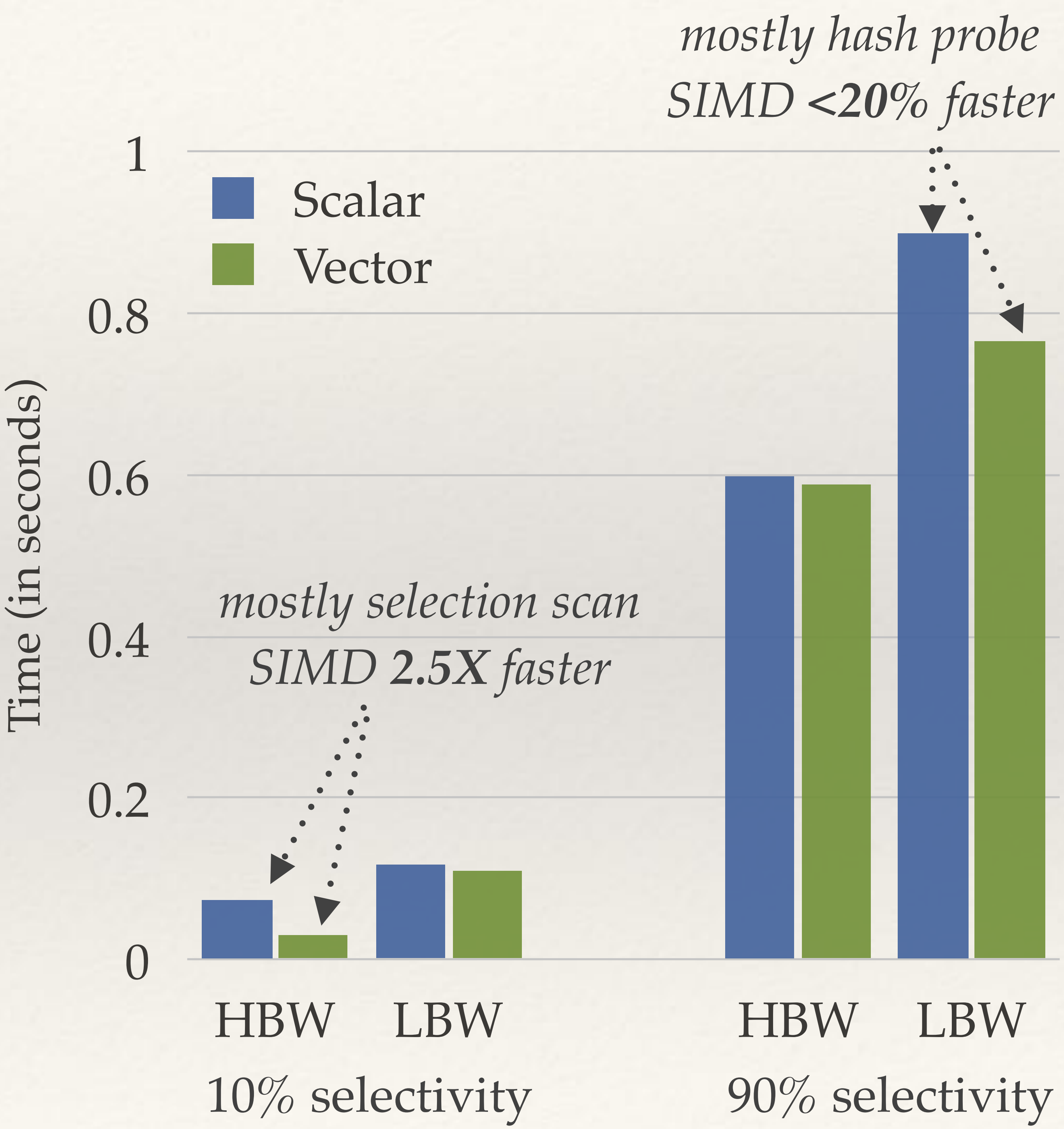
```
typedef struct {
   int set:1;
   A_key_t key;
   A_val_t val;
} A_key_val_t;
```

```
select sum(F.val * A.val * B.val)
from F, A, B
where F.key_A = A.key
   and F.key_B = B.key
   and F.val between x0 and y0
   and A.val between x1 and y1
   and B.val between x2 and y2;
```

```
for (size_t i = 0; i != F_tuples; ++i) {
  if (F_val[i] >= x0 && F_val[i] <= y0) {

    size_t h1 = hash(F_key_A[i], buckets_HJT_a);
    while (HJT_a[h1].set) {
      if (HTJ_a[h1].key == F_key_A[i]) {

        size_t h2 = hash(F_key[i], buckets_HJT_b);
        while (HJT_b[h2].set) {
          if (HJT_b[h2].key == F_key[i]) {

            sum += F_val[i] * HJT_a[h1].val
               * HJT_b[h2].val; }

          if (++h2 == buckets_HJT_b) h2 = 0; }}

      if (++h1 == buckets_HJT_a) h1 = 0; }}}
```

Baseline + SIMD Vectorization

- Maximize data parallelism
  - Written entirely in SIMD
  - No **register-resident** execution
  - Move data in cache-resident **buffers**
- SIMD can **hurt** performance
  - Due to **cache** & **TLB misses**
  - Fast RAM does **not** help



VIP Engine

- Based on “sub-operators” that …
  - Process a **block** of tuples at a time
  - Process one **column** at a time within that block
  - Designed to be **data-parallel**
  - Implemented **entirely** in SIMD
- **Why** is the design fast ?
  - **Specialized** sub-operators can be extremely **optimized**
  - Block at a time execution reduces **materialization** & **interpretation** cost
  - Use **cache-conscious** execution to utilize both **SIMD** and **fast RAM**

Sub-operators: An Example

- Hash a composite key <A, B> (of types X, Y)
  - Hash one **block** at a time
  - Hash one **column** at a time per block
  - Call **hash_X()** on column **A** of type **X** for a block of tuples
  - Call **hash_Y()** on column **B** of type **Y** for a block of tuples
  - Keep working set (block of hash values) **cache-resident**
  - **Amortize** interpretation cost

32-bit integer prototype

```
void hash_int32(const int32_t* data, uint32_t* hash, size_t tuples);
```

Selection Scans in VIP

- Based on sub-operators
  - Combine results using **bitmaps**
  - **Skip** tuples already determined
  - Process **W** items in SIMD
- Built-in **compression**
  - **Horizontal** dictionary compression
  - **Skip** tuples if determined
  - Decompress in 5 SIMD instructions

```
select * from T
where x = 9
and y > 1;
```

```
x = [1, 9, 8, 9]
y = [_, 0, _, 7]
```

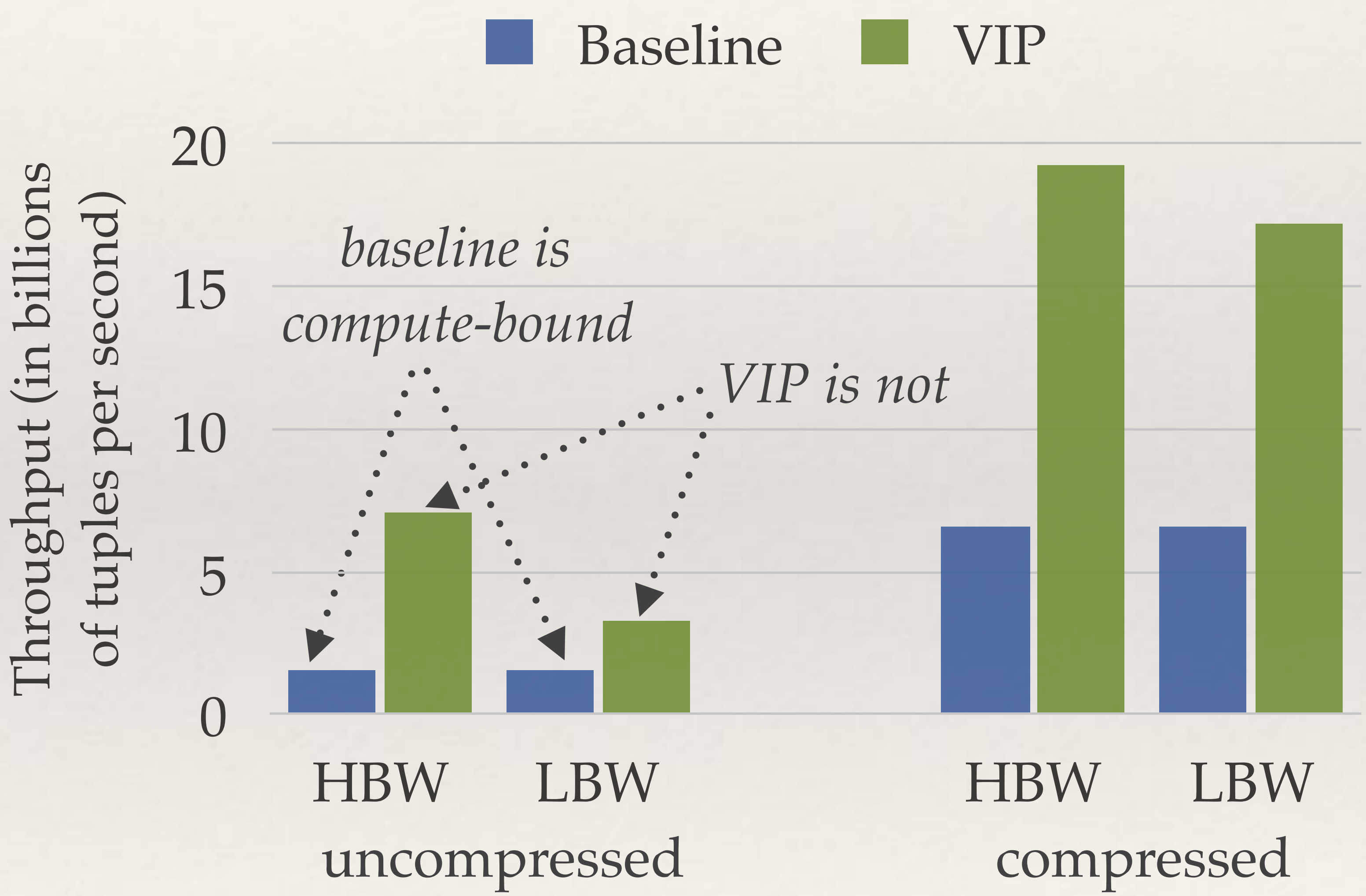
ignore or ***skip***

```
select * from T
where x = 9
or y > 1;
```

```
x = [1, 9, 8, 9]
y = [2, _, 1, _]
```

Selection Scans in VIP

- From TPC-H Q19 (SF = 1000)
  - Selection on **part** table
  - Neither CNF nor DNF
  - **0.24%** selectivity
  - **Skip** is essential here



Hash Joins in VIP

- Partition
  - **Inner** table must fit in the cache
- Hash join using **hash values**
  - **Specialized** data types & code
  - Generate rids lists
- Evaluate predicates
  - Use rid lists to access columns
  - Also evaluate non-equality predicates
  - Resolve hash **conflicts**

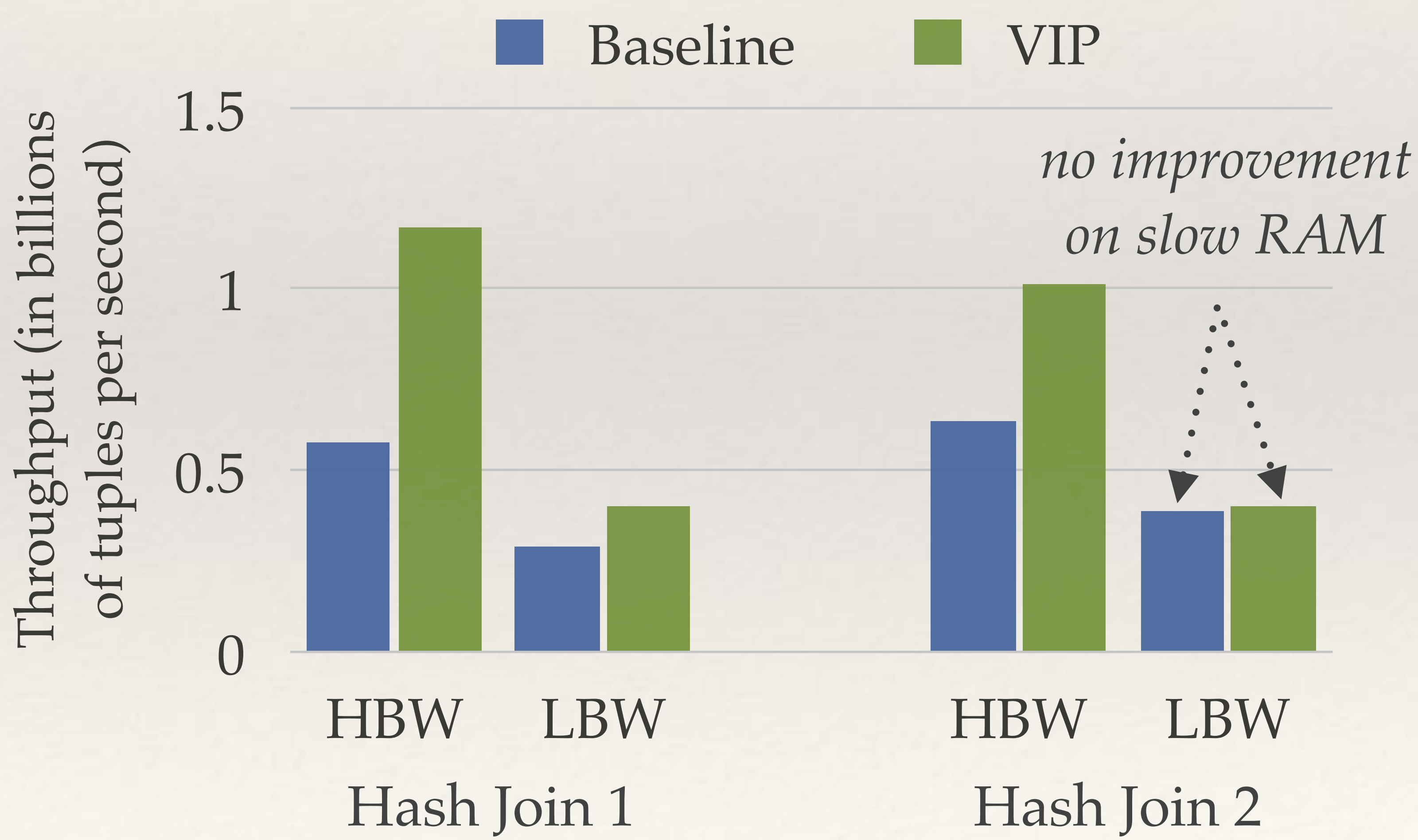
```
typedef struct {
  uint32_t hash;
  int32_t rid;
} join_bucket_t;

void build_hashes(
    const uint32_t* hashes,
    join_bucket_t* hash_table, [...]);

void probe_hashes(
    const uint32_t* hashes,
    const join_bucket_t* hash_table,
    int32_t* inner_rids,
    int32_t* outer_rids, [...]);
```

Hash Joins in VIP

```
select l_partkey, l_suppkey, o_custkey
from lineitem, orders
where l_orderkey = o_orderkey;
```

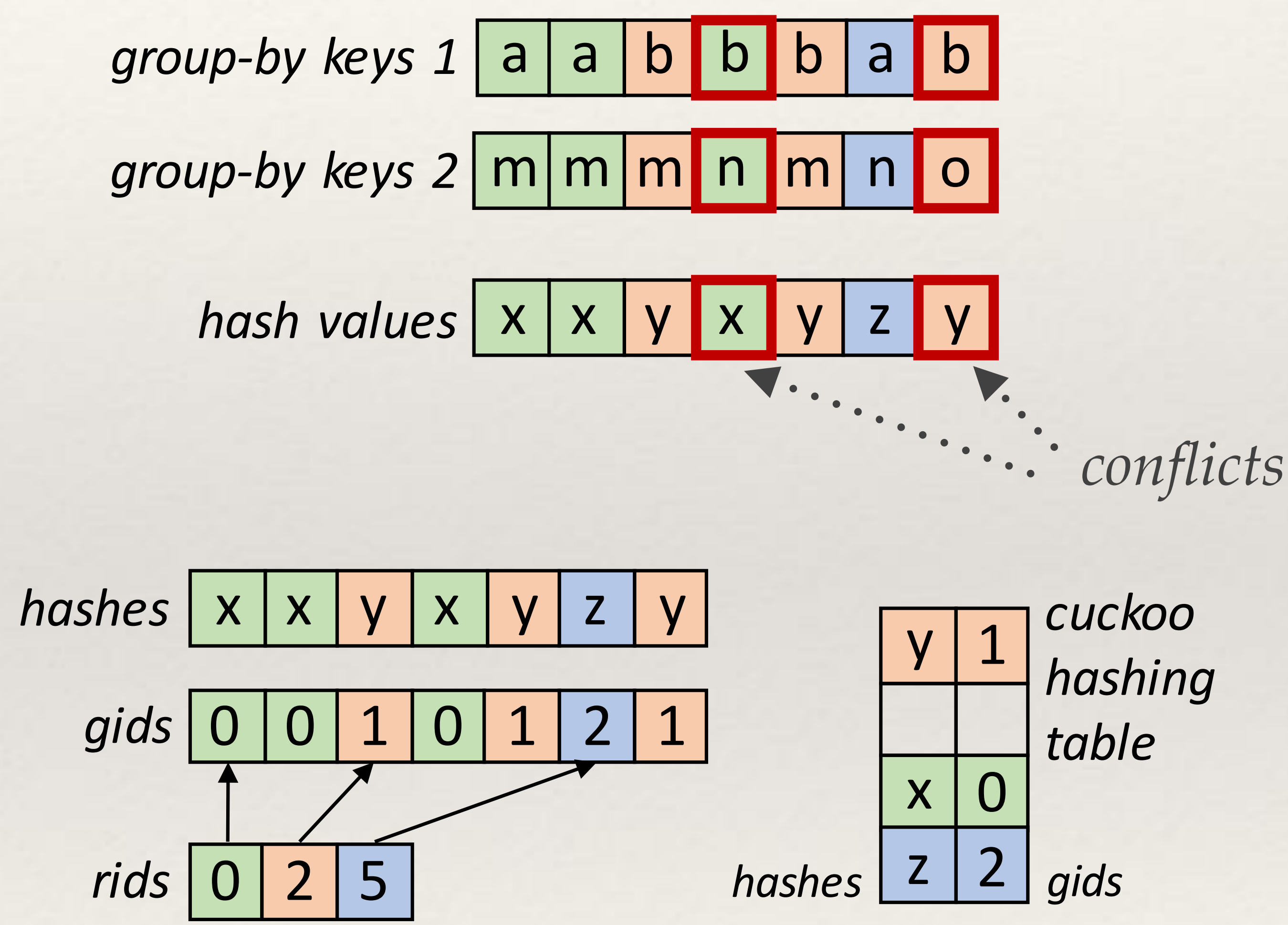
```
select l_orderkey, l_partkey, l_suppkey
from lineitem, partsupp
where l_partkey = ps_partkey
   and l_suppkey = ps_suppkey;
```

- From TPC-H (SF = 30)
  - **Largest** base tables
  - **Core** joins of TPC-H



Group-by Aggregation in VIP

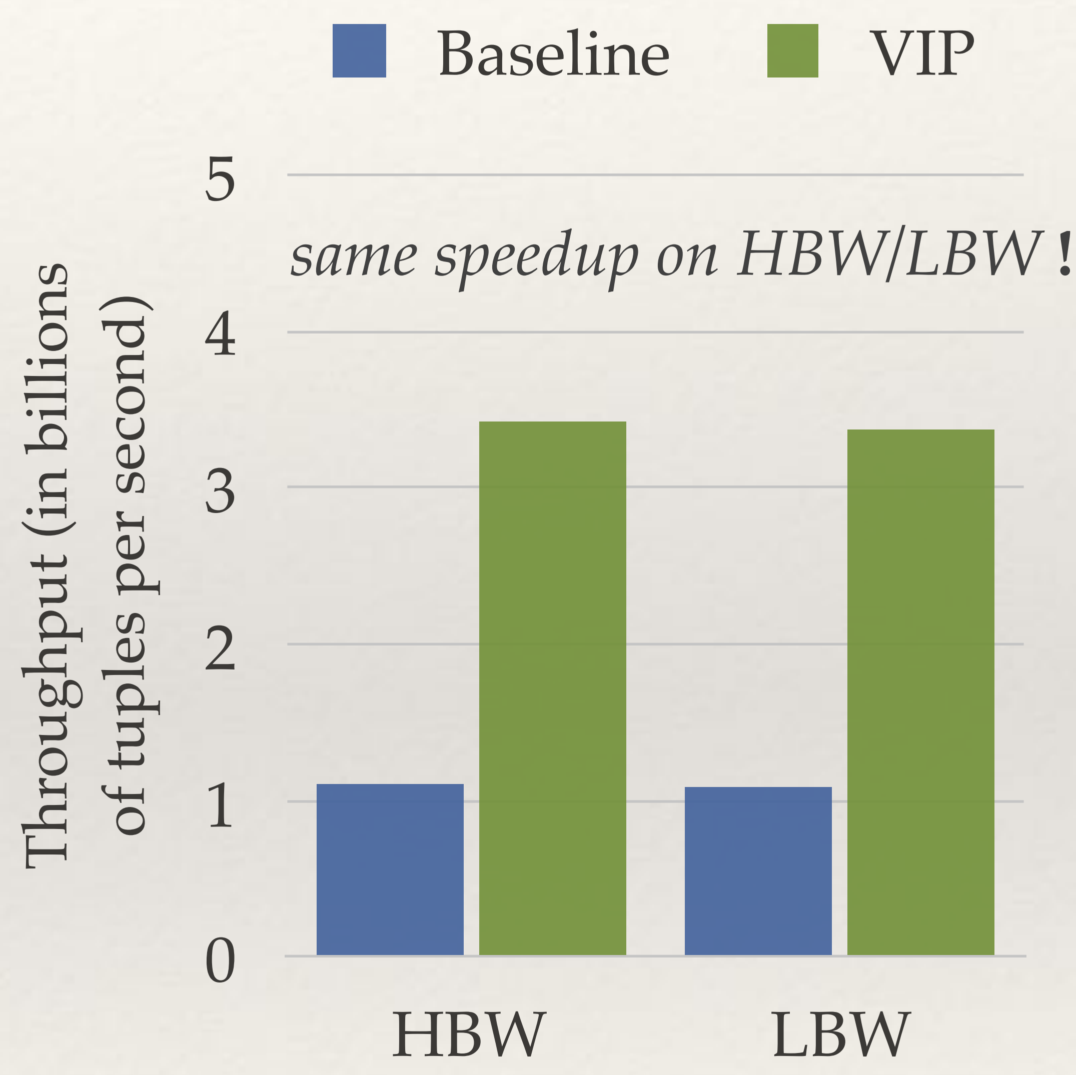
- Partition
  - Estimate number of groups
  - **Output** groups must fit in the cache
- Map hashes to group-ids
  - Using **specialized** sub-operator
  - Fix hash **conflicts**
- Compute **expressions**
  - Store in **cache-resident** buffers
- Update aggregates via group-ids
  - Keep partial aggregates **in the cache**



Group-by Aggregation in VIP

- From TPC-H Q1 (SF = 100)
  - **2** group-by attributes
  - **4** payload attributes
  - **8** aggregate functions
  - Reuse buffers for **sub-expressions**

```
sum(l_extendedprice),
sum(l_extendedprice * (1 - l_discount),
sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)
```



Future Work

- Track Join
  - Overlap CPU & network computation to reduce **end-to-end** time
  - Combine with **scheduling** algorithms for network transfers
- Compression
  - **Multiple** dictionaries or more complex schemes (e.g. Huffman encoding)
  - **Dynamic** dictionary encoding (e.g. add & update dictionary values)
- Vectorization
  - Evaluate new hardware **platforms** with better SIMD (e.g. AVX-512)
  - Design better **hardware** for database (e.g. better SIMD instructions)
- VIP engine
  - **Pipeline** operators when cache misses cannot occur
  - Evaluate **materialization** strategies & build operators in VIP

Published Papers

- SIMD-Accelerated Regular Expression Matching
  - At DaMoN ’16 with Eva Sitaridi, Kenneth A. Ross
- Rethinking SIMD Vectorization for In-Memory Databases
  - At SIGMOD ’15 with Arun Raghavan, Kenneth A. Ross
- Efficient Lightweight Compression Alongside Fast Scans
  - At DaMoN ’15 with Kenneth A. Ross
- Energy Analysis of Hardware and Software Range Partitioning
  - At TOCS with Lisa Wu, Raymond J. Barker, Martha A. Kim, Kenneth A. Ross
- A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scalar Comparison- and Radix-Sort
  - At SIGMOD ’14 with Kenneth A. Ross
- Track Join: Distributed Joins with Minimal Network Traffic
  - At SIGMOD ’14 with Rajkumar Sen, Kenneth A. Ross
- Vectorized Bloom Filters for Advanced SIMD Processors
  - At DaMoN ’14 with Kenneth A. Ross
- High Throughput Heavy Hitter Aggregation for Modern SIMD Processors
  - At DaMoN ’14 with Kenneth A. Ross

Acknowledgments

- My advisor Ken
- My PhD thesis committee
    - Martha, Luis, Eugene, and Stratos
- Friends and colleagues from the DB group
    - Fotis, John, Eva, Pablo, and Wangda
- Colleagues from Oracle and Amazon
    - Arun, Eric, Ippokratis, Michalis, and Raj
- More friends from Columbia & New York

Thank you!