Impact of Traditional Sparse Optimizations on a Migratory Thread Architecture

Thomas B. Rolinger, Christopher D. Krieger

SC 2018

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

LPS
The Laboratory for Physical Sciences

Outline

1. Motivation
2. Emu Architecture
3. SpMV Optimizations
4. Experiments and Results
5. Conclusions & Future Work

1.) Motivation

1.) Motivation

- Sparse linear algebra kernels
  - Present in many scientific/big-data applications
  - Achieving high performance is difficult
    - irregular access patterns and weak locality
  - Most approaches target today's architectures: deep-memory hierarchies, GPUs, etc.

1.) Motivation

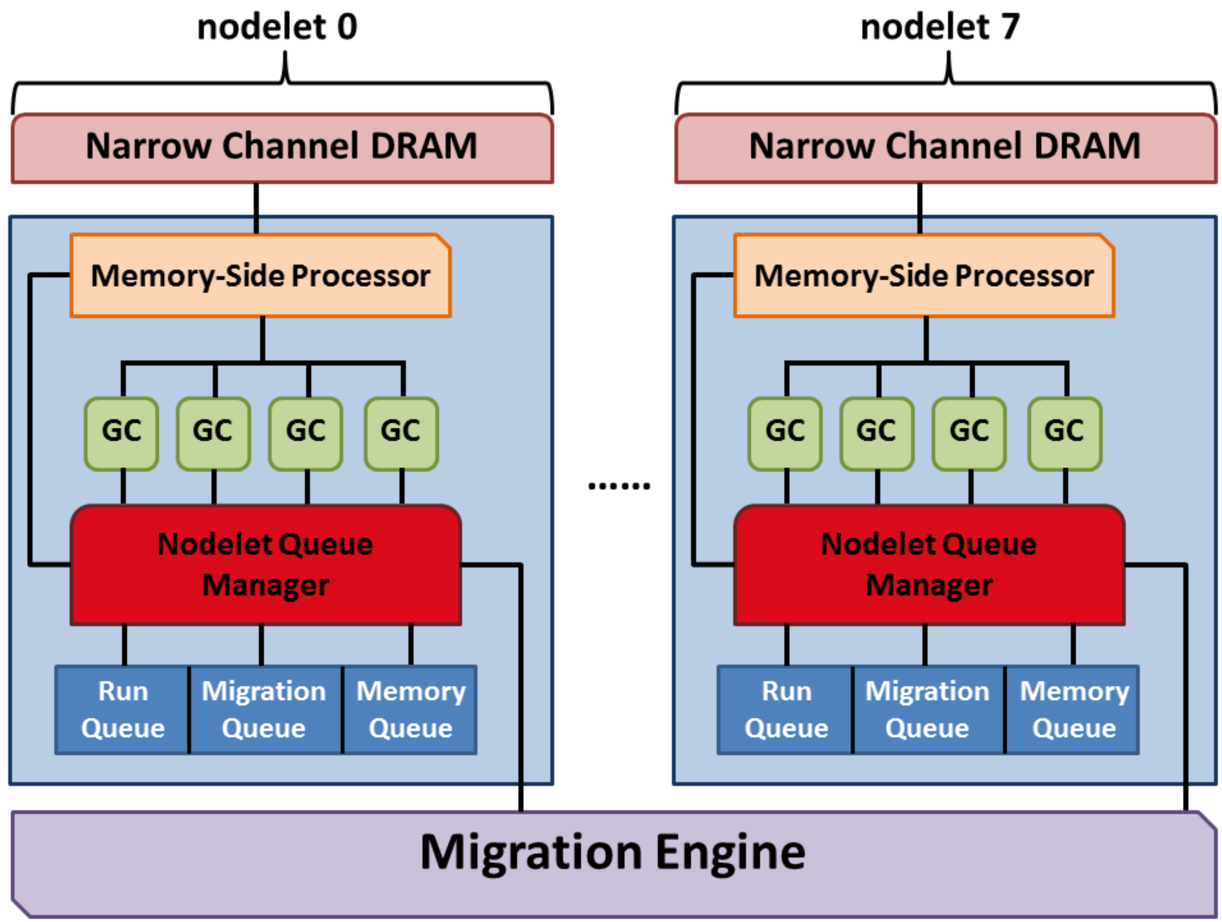
- Sparse linear algebra kernels
  - Present in many scientific/big-data applications
  - Achieving high performance is difficult
    - irregular access patterns and weak locality
  - Most approaches target today's architectures: deep-memory hierarchies, GPUs, etc.
- Novel architectures for sparse applications
  - Emu: light-weight migratory threads, narrow memory, near-memory processing

1.) Motivation

- Sparse linear algebra kernels
  - Present in many scientific/big-data applications
  - Achieving high performance is difficult
    - irregular access patterns and weak locality
  - Most approaches target today's architectures: deep-memory hierarchies, GPUs, etc.
- Novel architectures for sparse applications
  - Emu: light-weight migratory threads, narrow memory, near-memory processing
- Our work
  - Study impact of existing optimizations for sparse algorithms on Emu versus cache-memory based systems
  - Target algorithm: Sparse Matrix-Vector Multiply (**SpMV**)
    - Compressed Sparse Row (**CSR**)

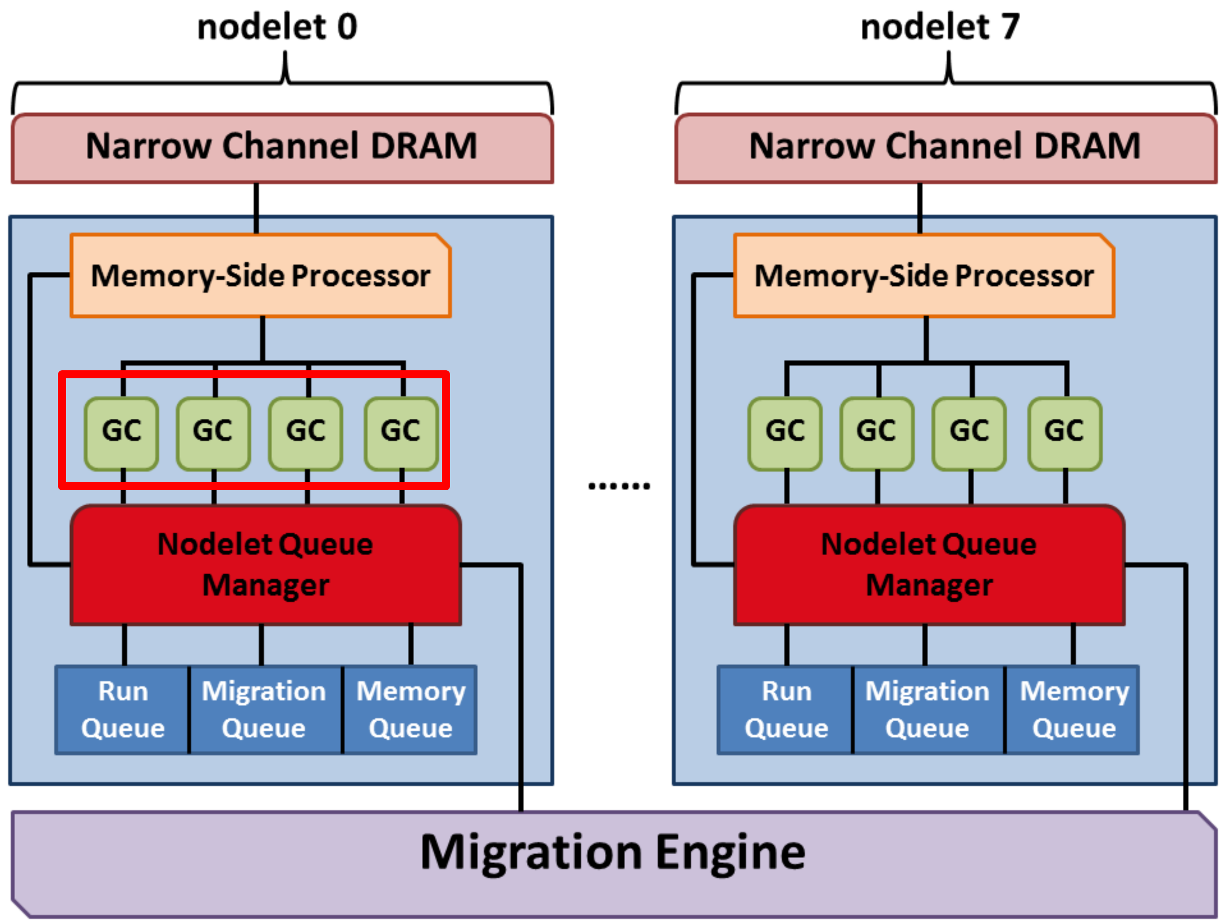
2.) Emu Architecture

2.) Emu Architecture



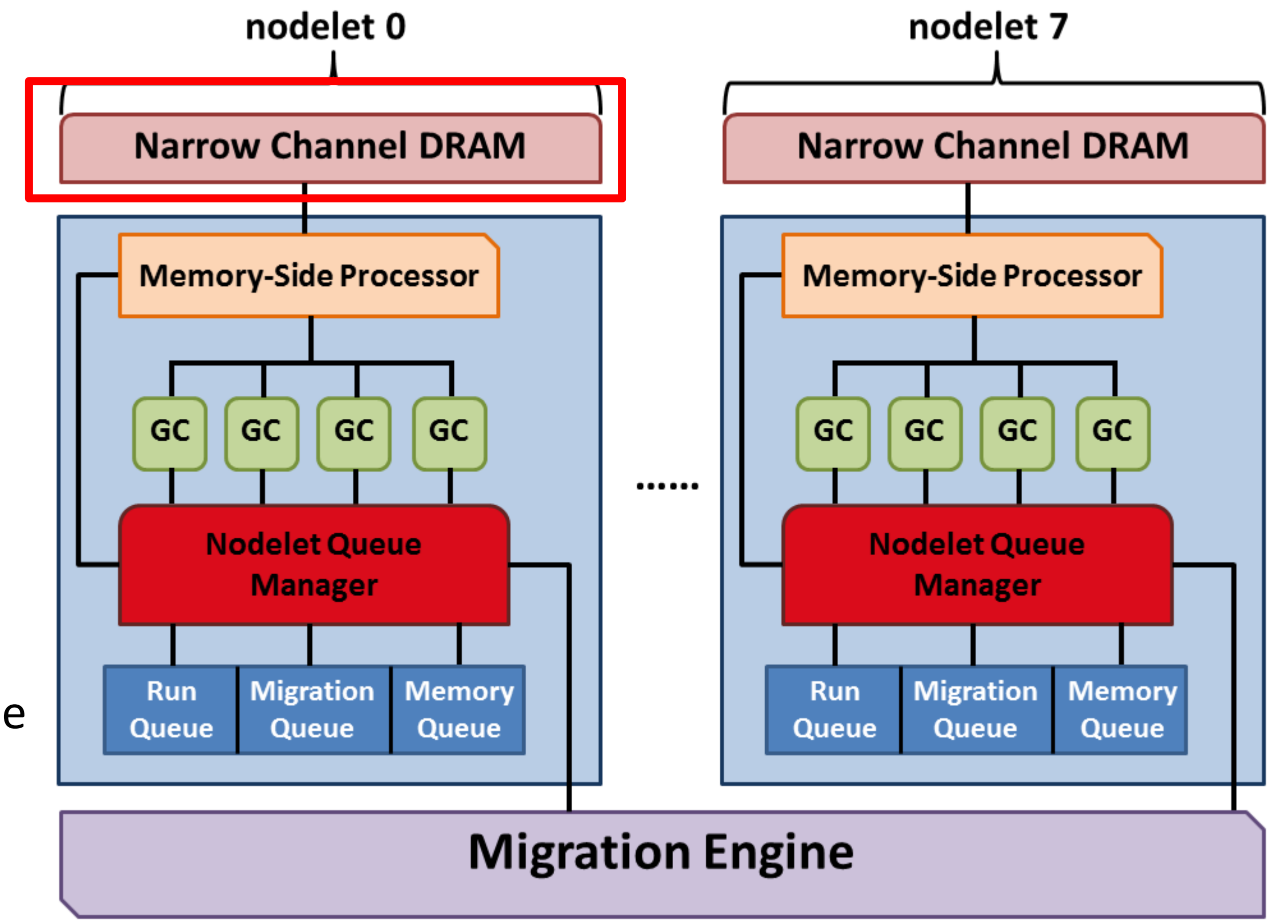
2.) Emu Architecture

- Gossamer Core (GC)
  - general purpose, cache-less
  - supports up to 64 concurrent light-weight threads



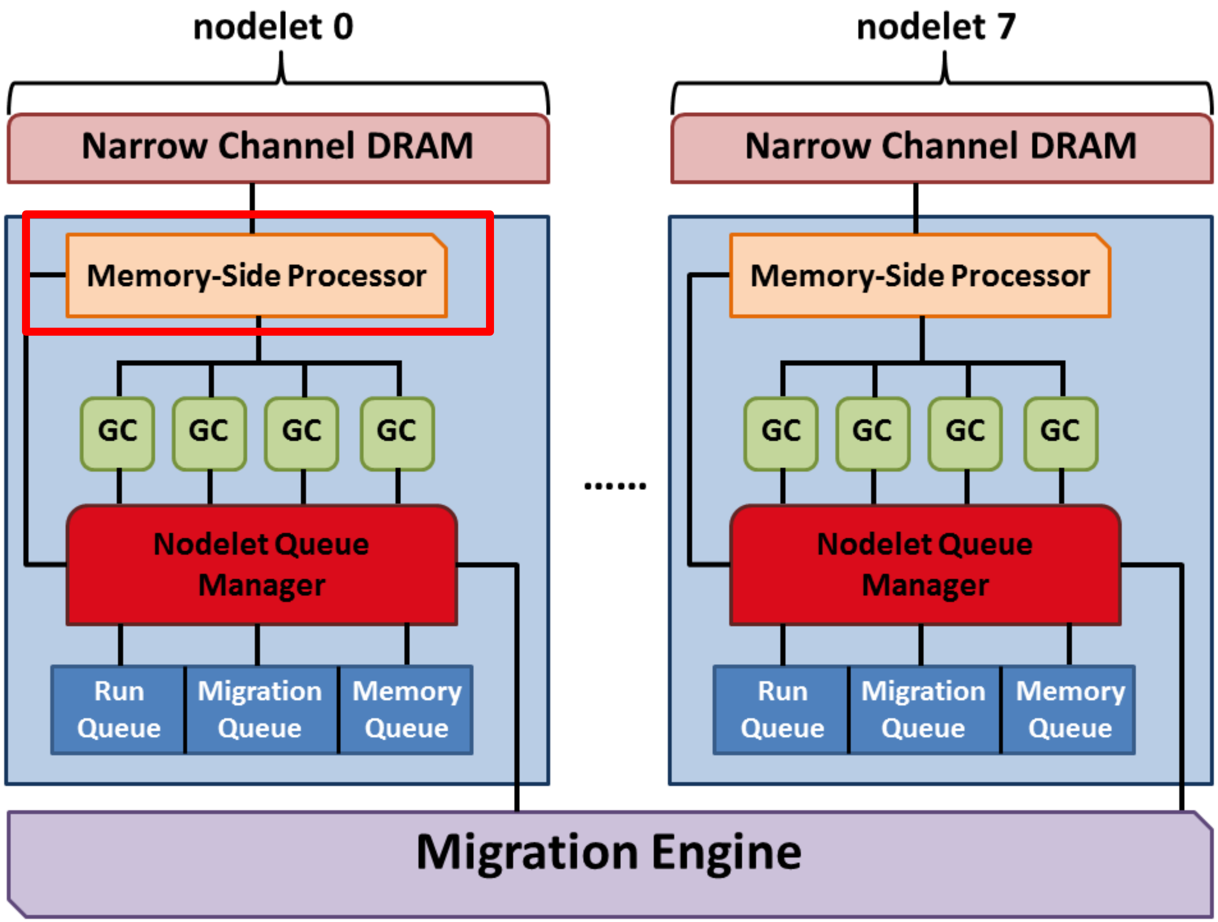
2.) Emu Architecture

- Gossamer Core (GC)
  - general purpose, cache-less
  - supports up to 64 concurrent light-weight threads
- Narrow Memory
  - eight 8-bit channels rather than a single, wider 64-bit interface



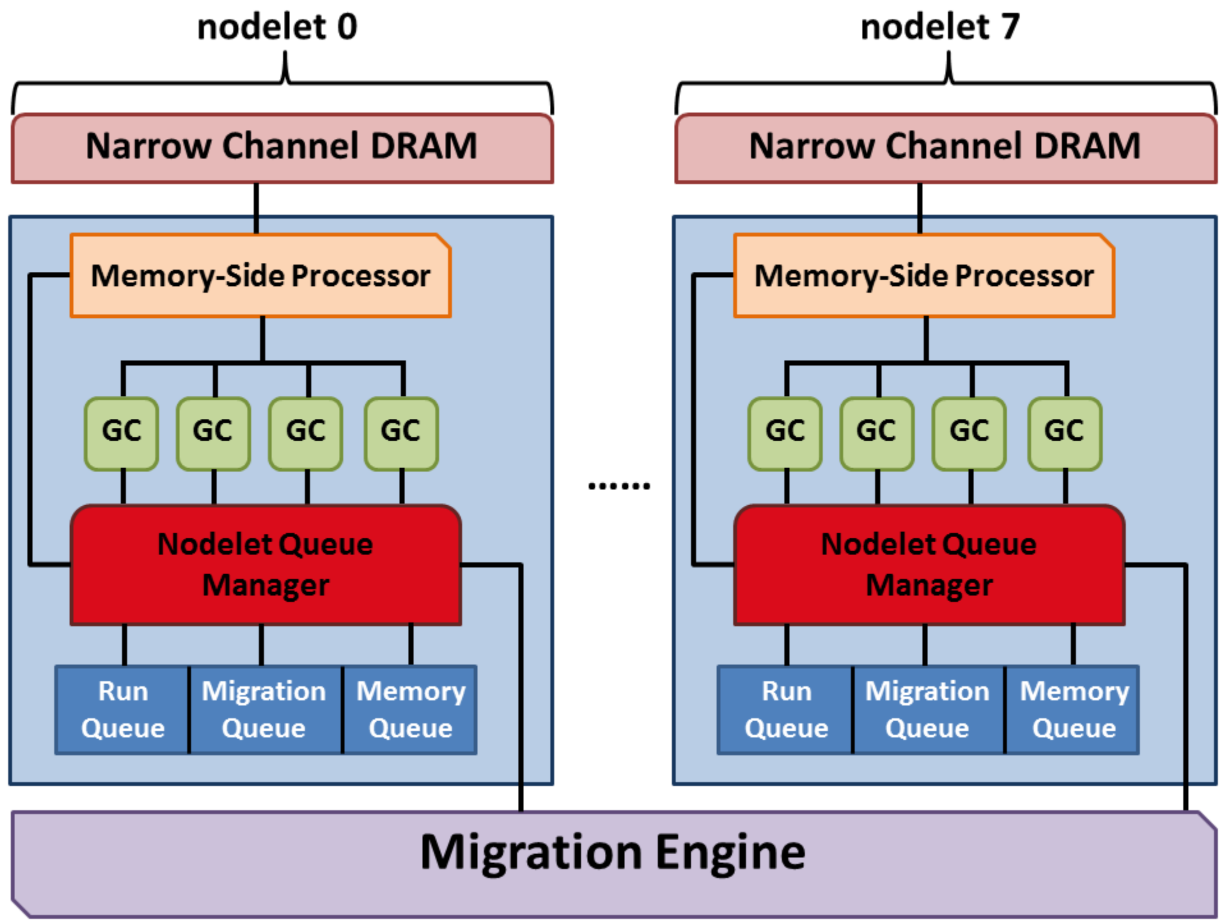
2.) Emu Architecture

- Gossamer Core (GC)
  - general purpose, cache-less
  - supports up to 64 concurrent light-weight threads
- Narrow Memory
  - eight 8-bit channels rather than a single, wider 64-bit interface
- Memory-side Processor
  - executes atomic and remote operations
  - remote ops do not generate migrations



2.) Emu Architecture

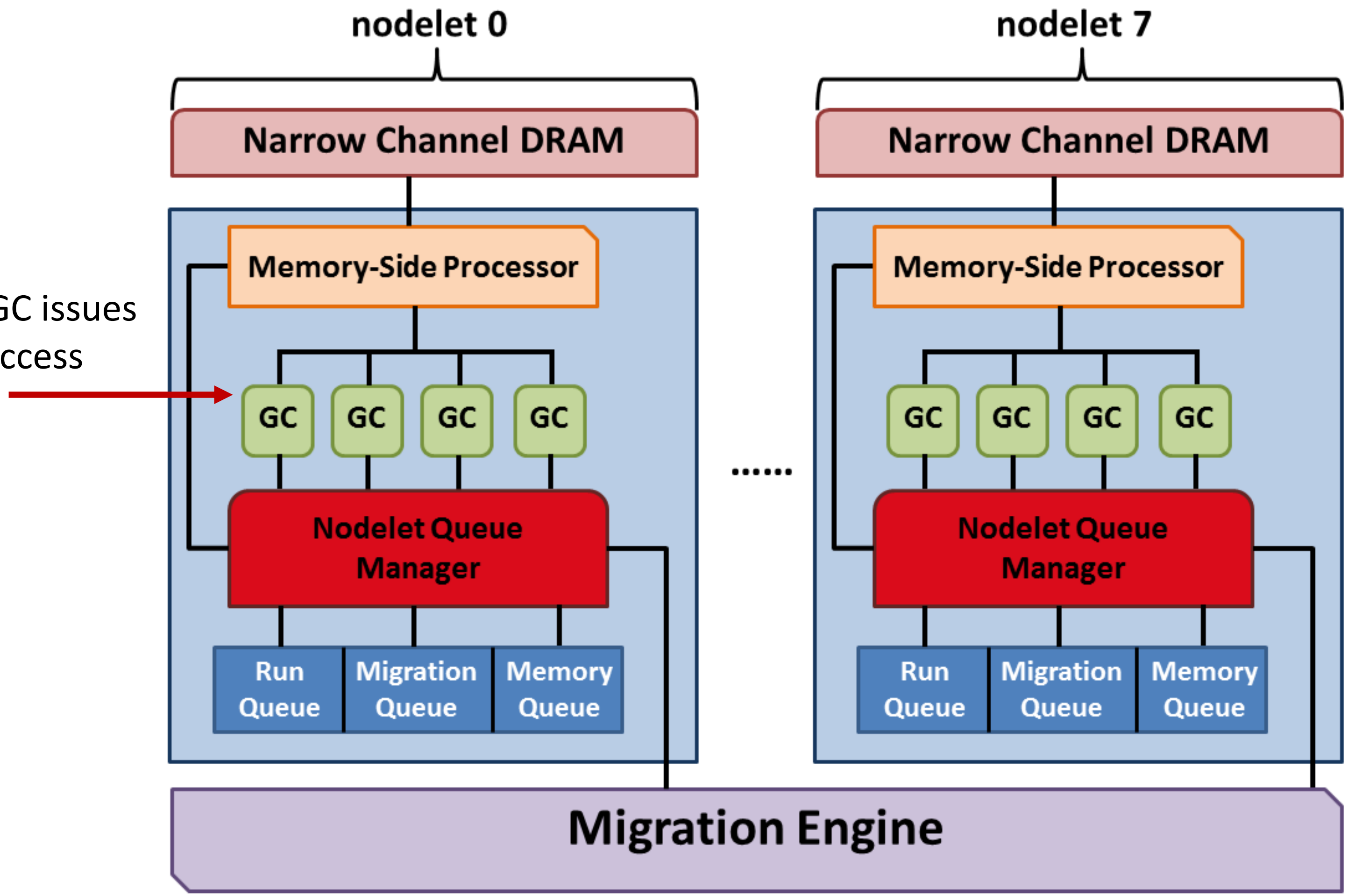
- Gossamer Core (GC)
  - general purpose, cache-less
  - supports up to 64 concurrent light-weight threads
- Narrow Memory
  - eight 8-bit channels rather than a single, wider 64-bit interface
- Memory-side Processor
  - executes atomic and remote operations
  - remote ops do not generate migrations



System used in our work:
1 node: 8 nodelets with 1 GC per nodelet (150MHz)
8GB DDR4 1600MHz per nodelet
64 threads per nodelet (512 total)

2.) Emu Architecture: Migrations

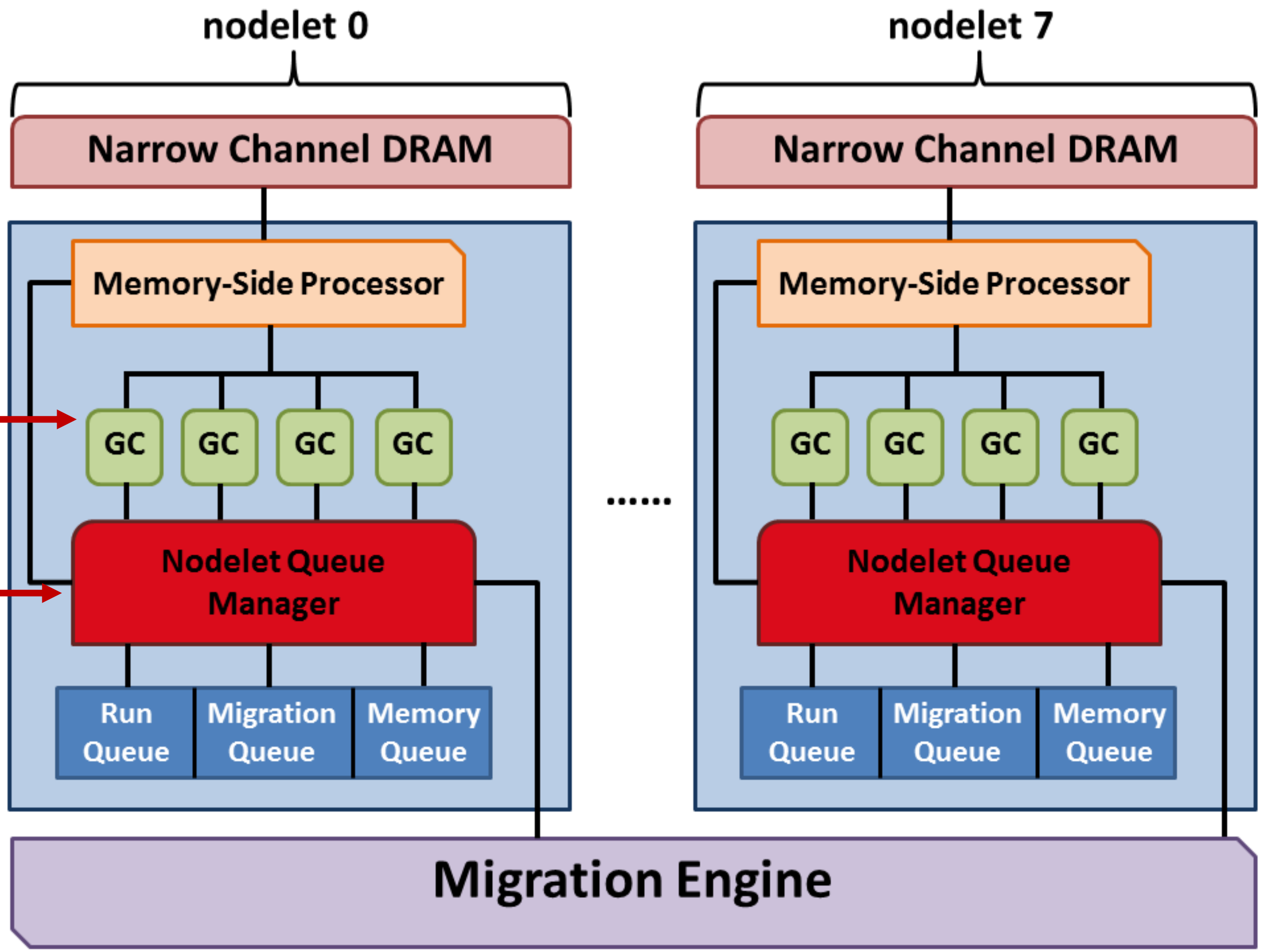
1.) Thread on GC issues remote mem access



2.) Emu Architecture: Migrations

1.) Thread on GC issues remote mem access

2.) GC makes request to NQM to migrate thread

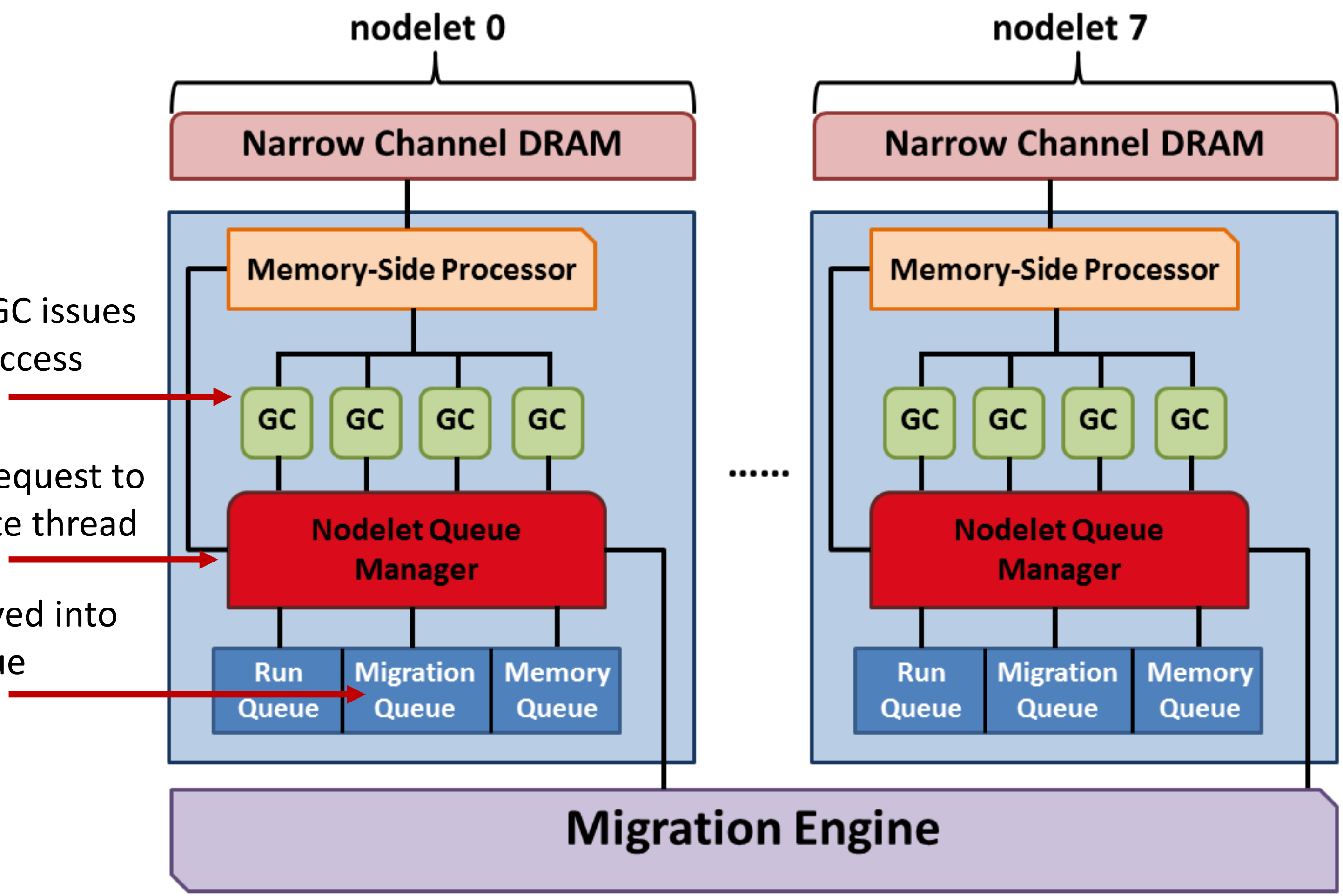


2.) Emu Architecture: **Migrations**

1.) Thread on GC issues remote mem access

2.) GC makes request to NQM to migrate thread

3.) Thread moved into migration queue



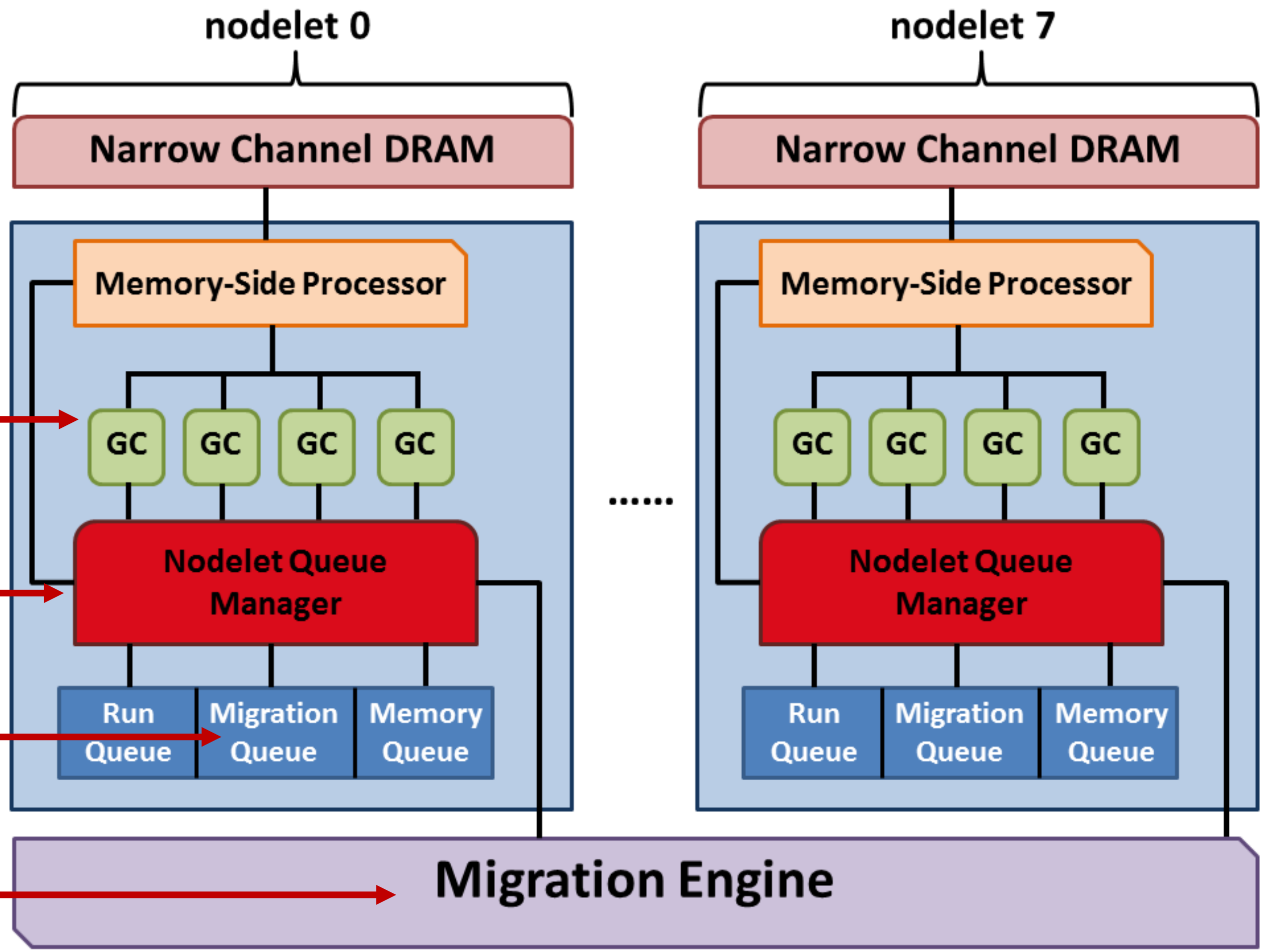
2.) Emu Architecture: Migrations

1.) Thread on GC issues remote mem access

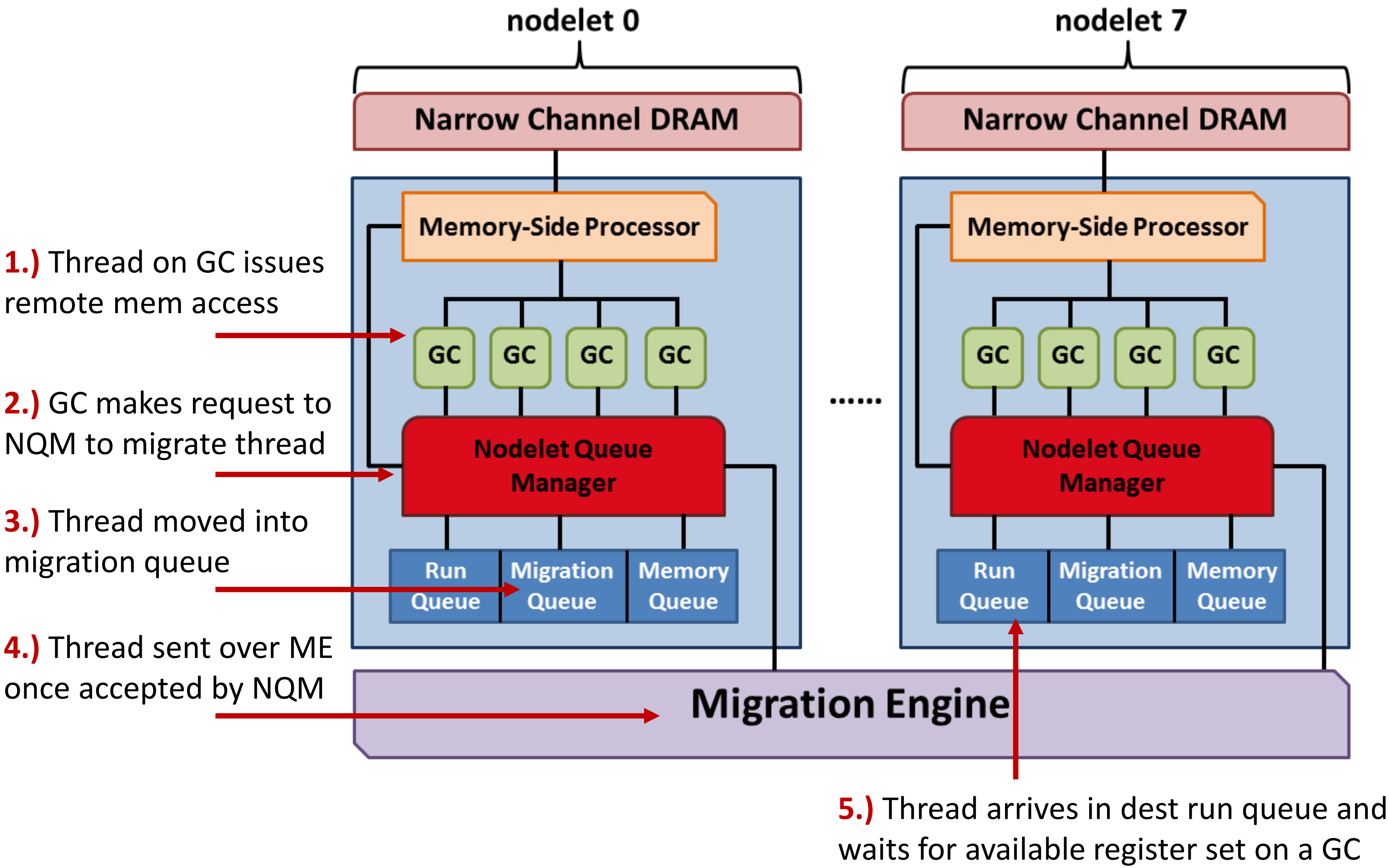
2.) GC makes request to NQM to migrate thread

3.) Thread moved into migration queue

4.) Thread sent over ME once accepted by NQM



2.) Emu Architecture: Migrations



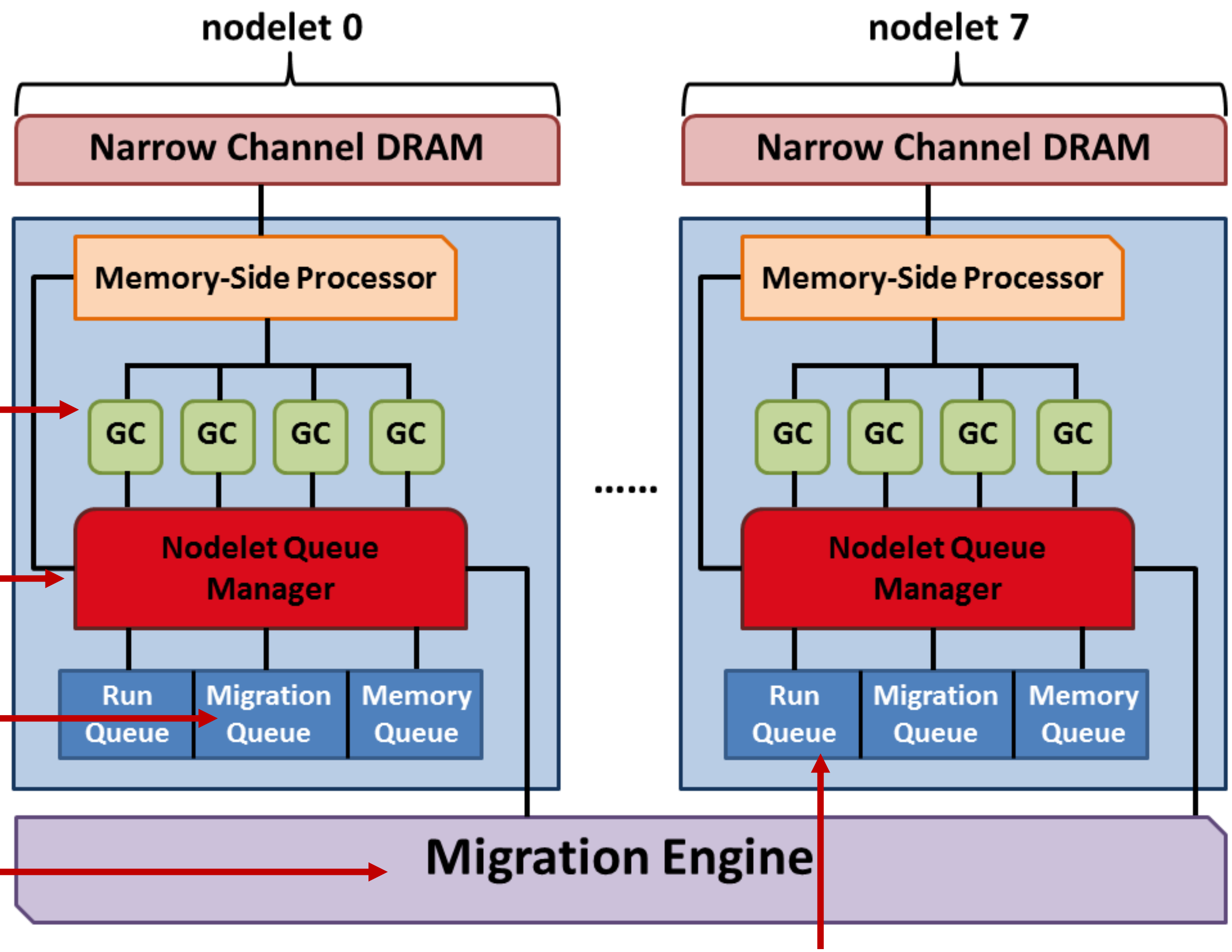
2.) Emu Architecture: Migrations

1.) Thread on GC issues remote mem access

2.) GC makes request to NQM to migrate thread

3.) Thread moved into migration queue

4.) Thread sent over ME once accepted by NQM



5.) Thread arrives in dest run queue and waits for available register set on a GC

Thread Context: Roughly 200 bytes (PC, registers, stack counter, etc.)

Migration Cost: ~2x more than a local access

3.) SpMV Optimizations

3.) SpMV Optimizations: **Vector Data Layout**

- Updating **b** may require remote writes
  - non-zeros on row *i* are all assigned to a single thread → **b**[*i*] accumulated in register and then updated via single remote write (or local write)

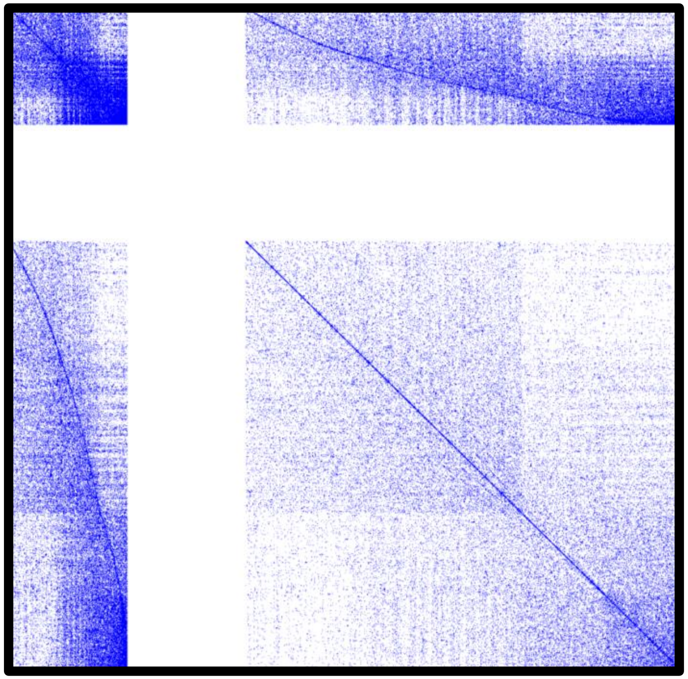
3.) SpMV Optimizations: Vector Data Layout

- Updating **b** may require remote writes
  - non-zeros on row *i* are all assigned to a single thread → **b**[*i*] accumulated in register and then updated via single remote write (or local write)
- SpMV requires one load from **x** per non-zero
  - each access may generate migration → layout of **x** is crucial to performance

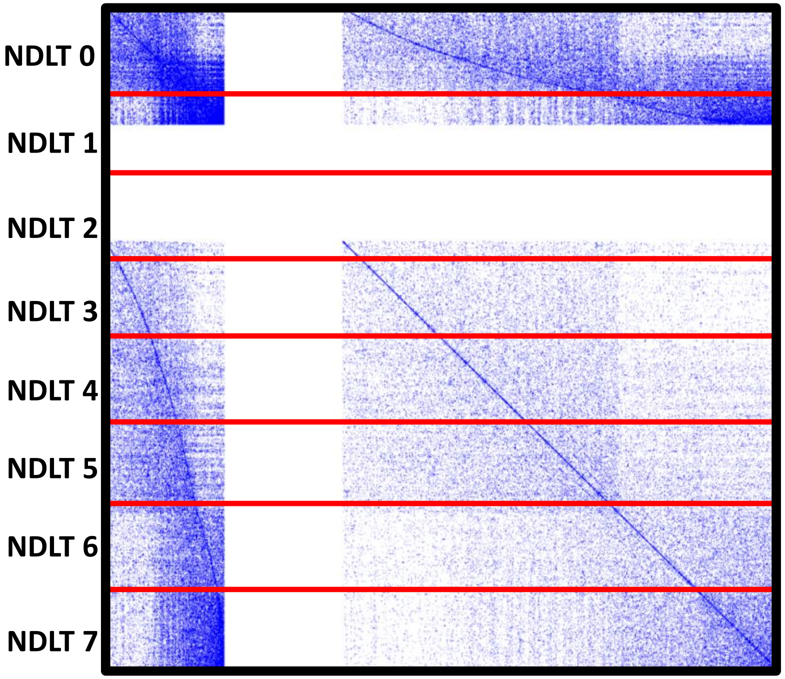
3.) SpMV Optimizations: **Vector Data Layout**

- Updating **b** may require remote writes
  - non-zeros on row *i* are all assigned to a single thread → **b**[*i*] accumulated in register and then updated via single remote write (or local write)
- SpMV requires one load from **x** per non-zero
  - each access may generate migration → layout of **x** is crucial to performance
- Cyclic and Block layouts
  - **Cyclic**: adjacent elements of vector are on different nodelets (round-robin) → consecutive accesses require migrations
  - **Block**: equally divide the vectors into fixed-size blocks and place 1 block on each nodelet

3.) SpMV Optimizations: **Work Distribution**

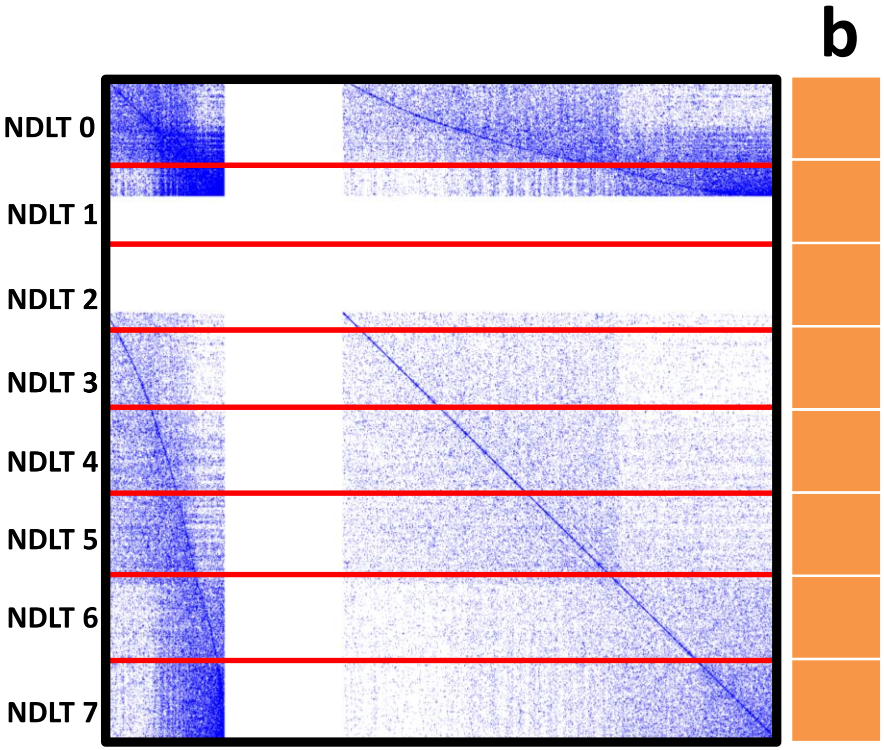


3.) SpMV Optimizations: **Work Distribution**



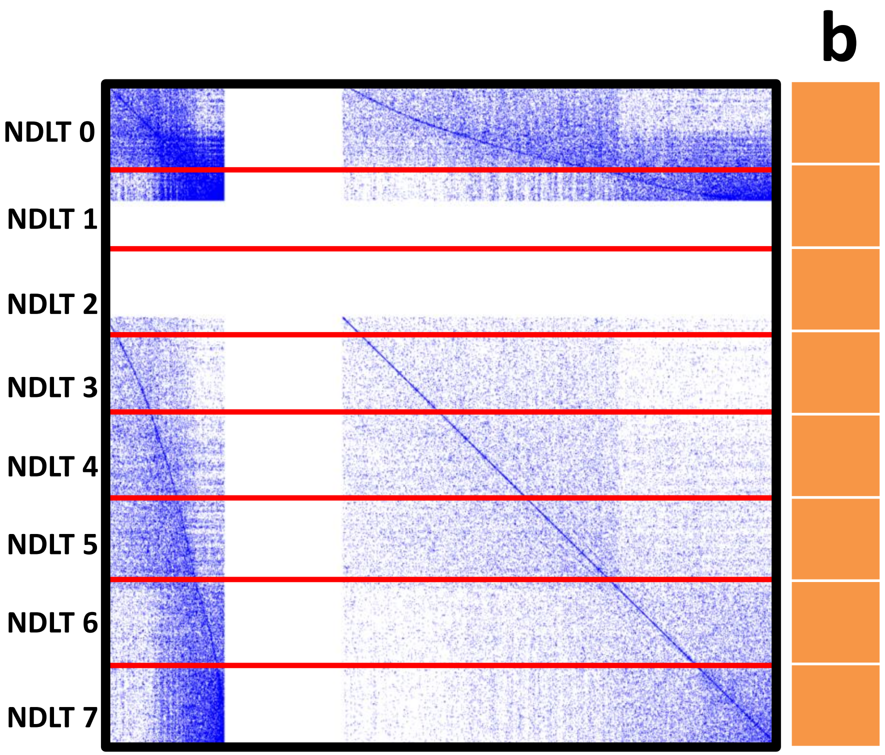
- **Row** based
  - evenly distribute rows

3.) SpMV Optimizations: **Work Distribution**



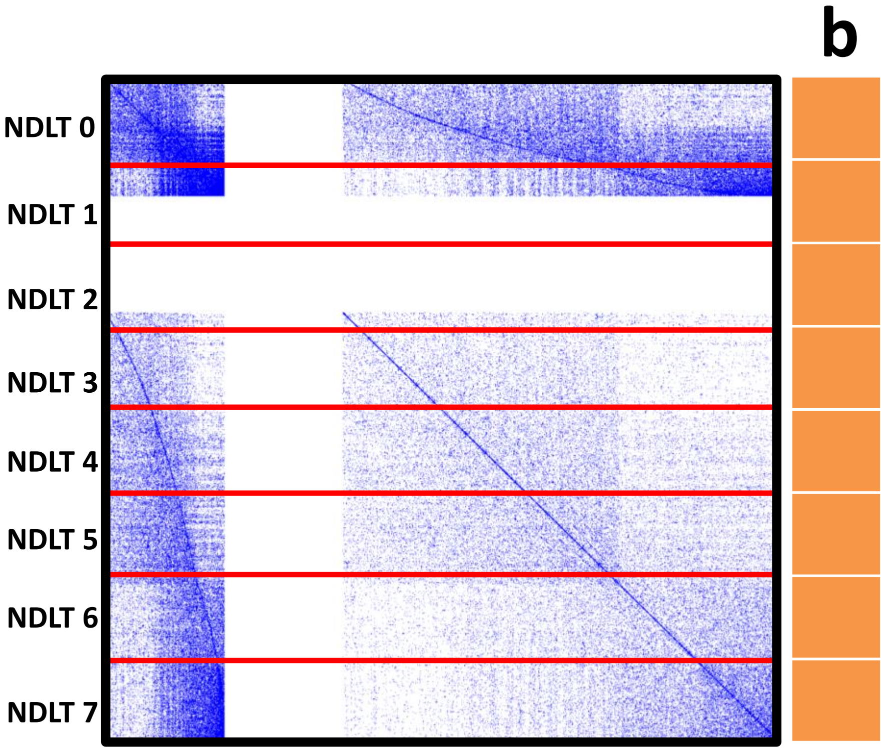
- **Row** based
  - evenly distribute rows
  - block size of **b** == # rows per nodelet

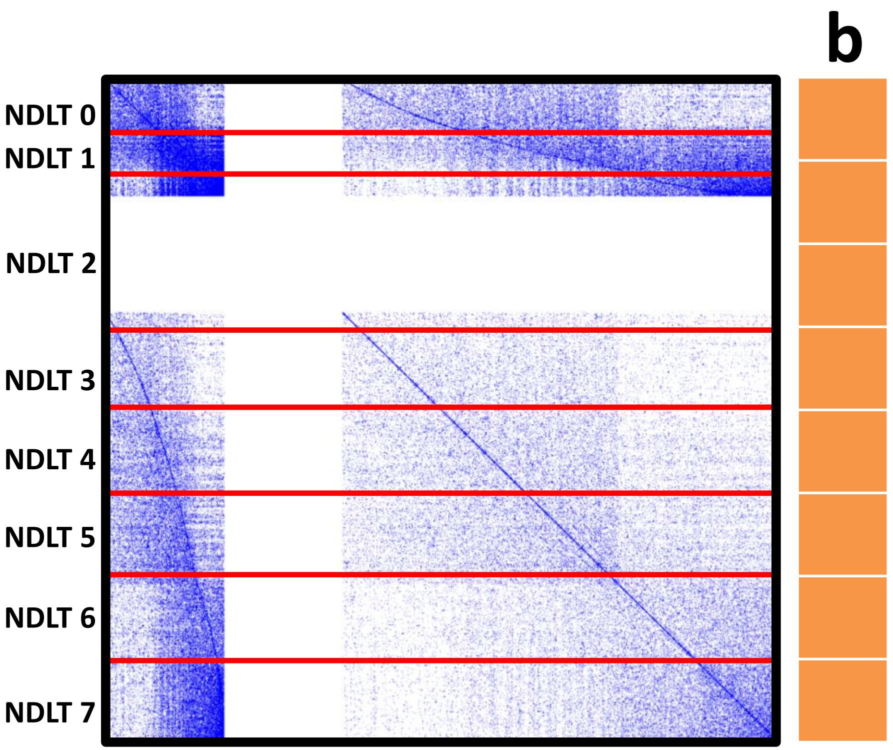
3.) SpMV Optimizations: **Work Distribution**



- **Row** based
  - evenly distribute rows
  - block size of **b** == # rows per nodelet
  - may assign unequal # of non-zeros to each nodelet

3.) SpMV Optimizations: **Work Distribution**

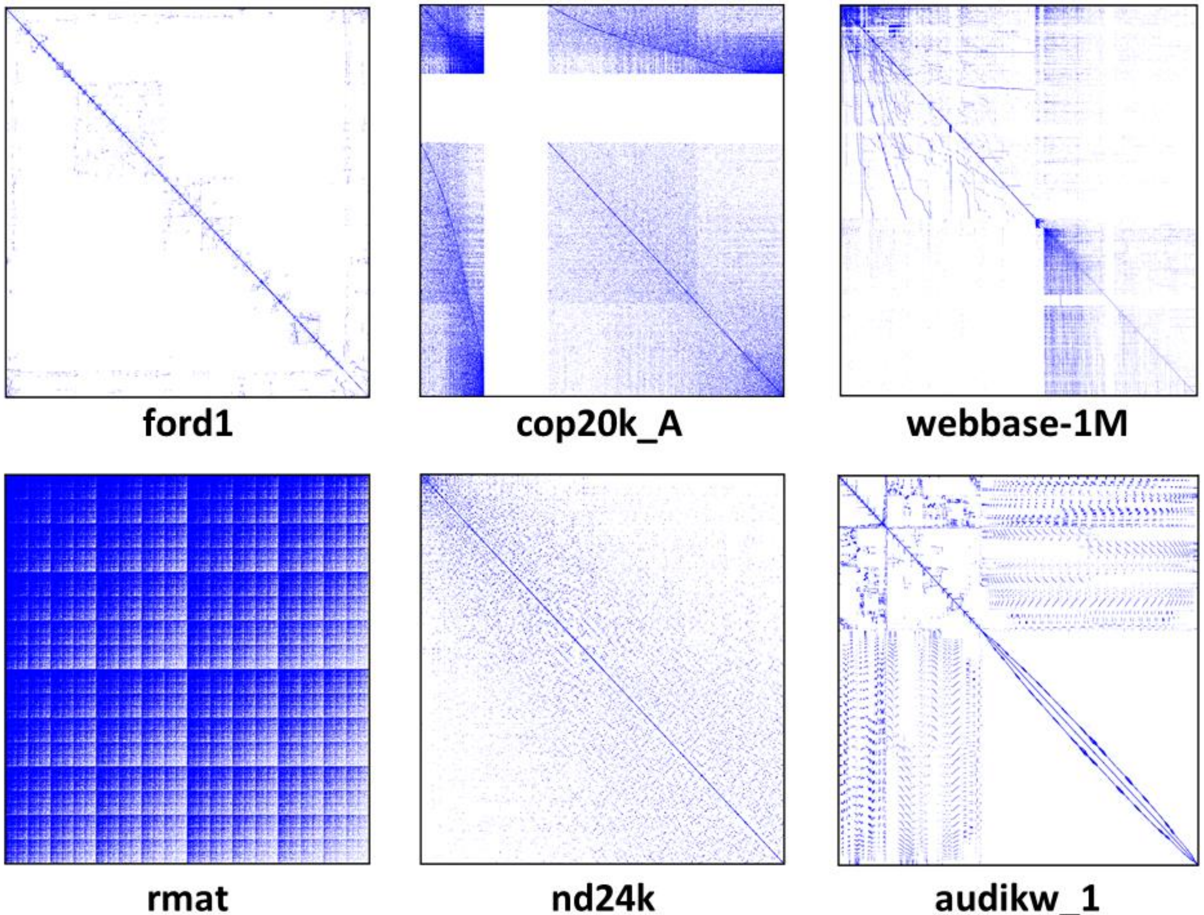




- **Row** based
  - evenly distribute rows
  - block size of **b** == # rows per nodelet
  - may assign unequal # of non-zeros to each nodelet
- **Non-zero** based
  - "evenly" distribute non-zeros
  - may assign unequal # of rows to each nodelet
    - remote writes may be required for **b**

4.) Experiments and Results

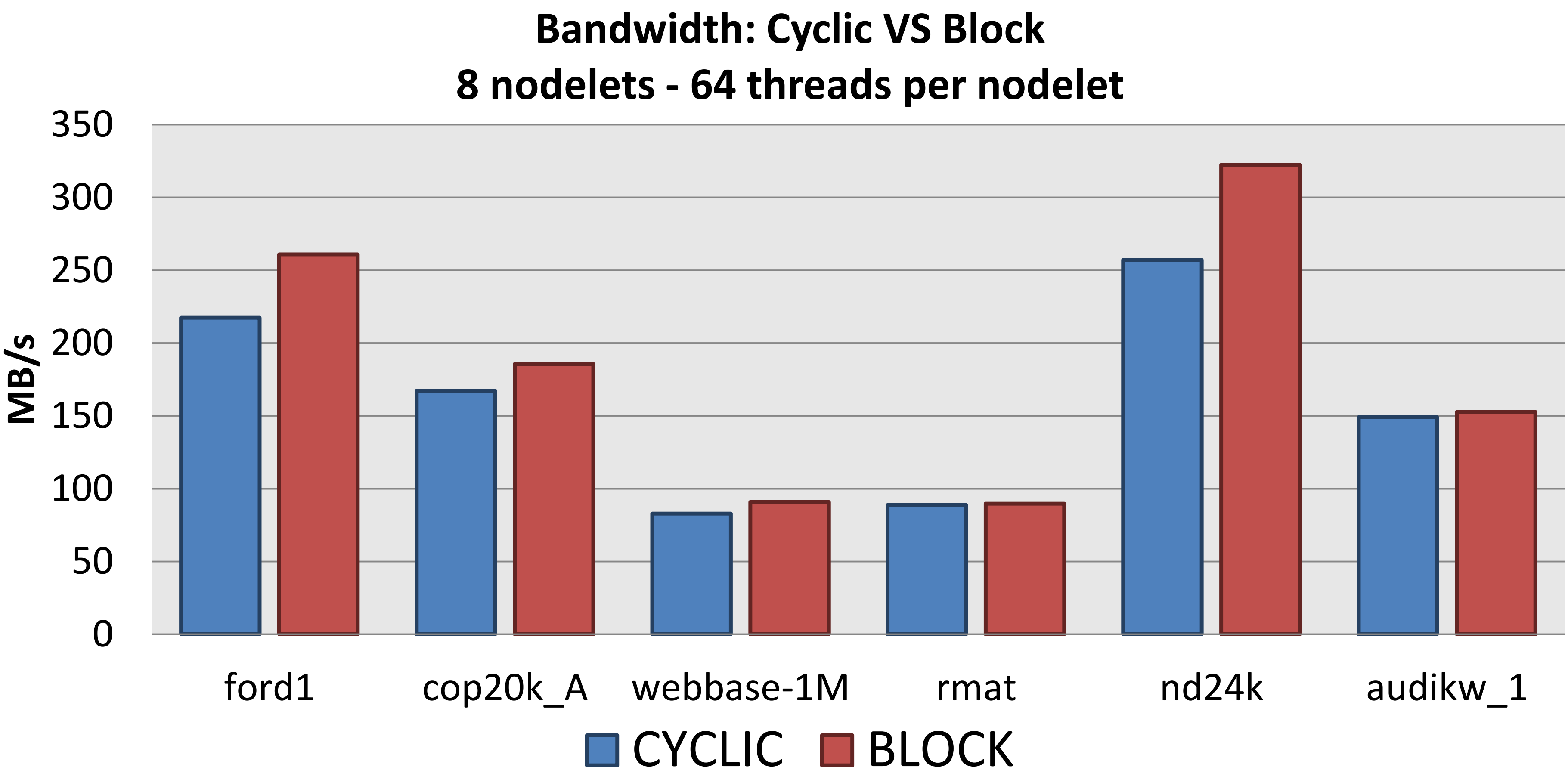
4.) Experiments: Matrices



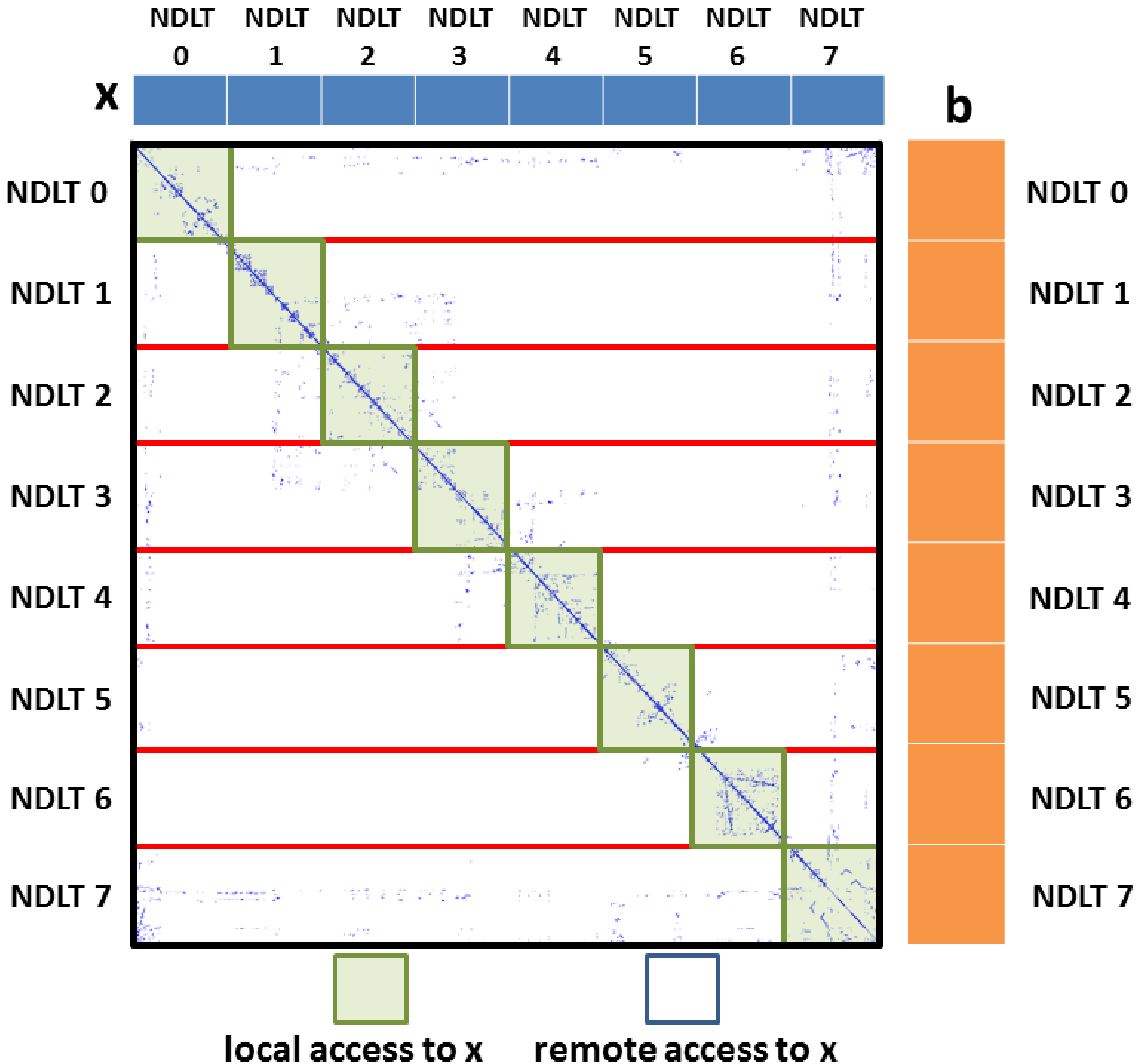
Name	Rows	Non-Zeros	Density
ford1	18K	100K	2.9×10^{-4}
cop20k_A	120K	2.6M	1.79×10^{-4}
webbase-1M*	1M	3.1M	3.11×10^{-6}
rmat*	445K	7.4M	3.74×10^{-5}
nd24k	72K	28.7M	5.54×10^{-3}
audikw_1	943K	77.6M	8.72×10^{-5}

- Evaluated SpMV across 40 matrices
  - Following results focus on a representative subset
  - RMAT graph produced with a=0.45, b=0.22, c=0.22
  - All matrices are square
  - Non-symmetric denoted with "*", symmetric matrices stored in their entirety

4.) Results: Vector Data Layouts

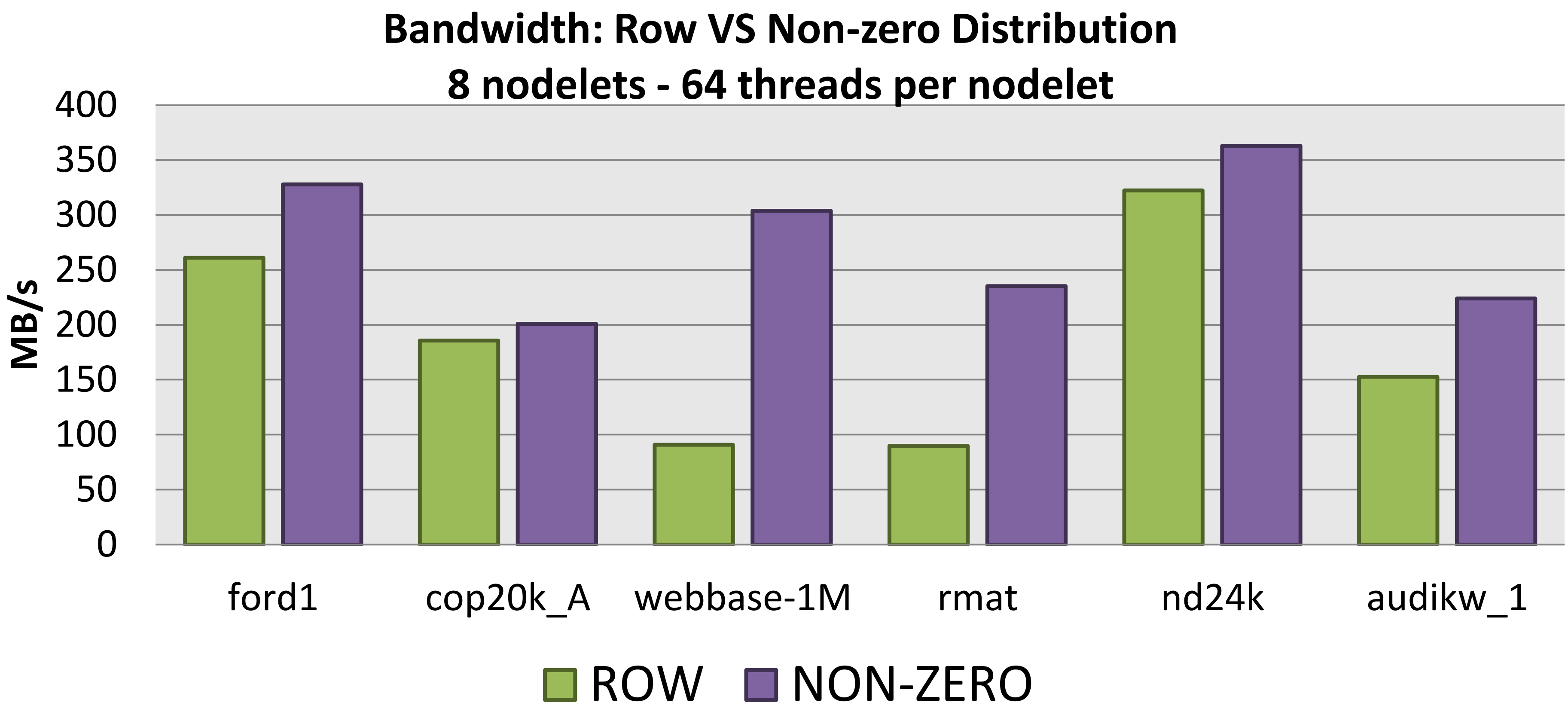


- Row-based work distribution used
- Block layout achieves up to **25% more BW**
  - better at reducing migrations on matrices with “tight” main diagonal (next slide) → **1.4x – 6.3x fewer** migrations than cyclic



NDLT 0
NDLT 1
NDLT 2
NDLT 3
NDLT 4
NDLT 5
NDLT 6
NDLT 7
x
b
NDLT 0
NDLT 1
NDLT 2
NDLT 3
NDLT 4
NDLT 5
NDLT 6
NDLT 7
NDLT 0
NDLT 1
NDLT 2
NDLT 3
NDLT 4
NDLT 5
NDLT 6
NDLT 7
local access to x
remote access to x

4.) Results: **Work Distribution**



- Block vector data layout used
- Non-zero distribution achieves up to **3.34x more BW**
  - provides significantly better load balancing
  - but incurs more migrations, on average → suggests that load balancing can be equally important to performance as reducing migrations

4.) Results: **Hardware Load Balancing**

- Cannot isolate threads to hardware resources

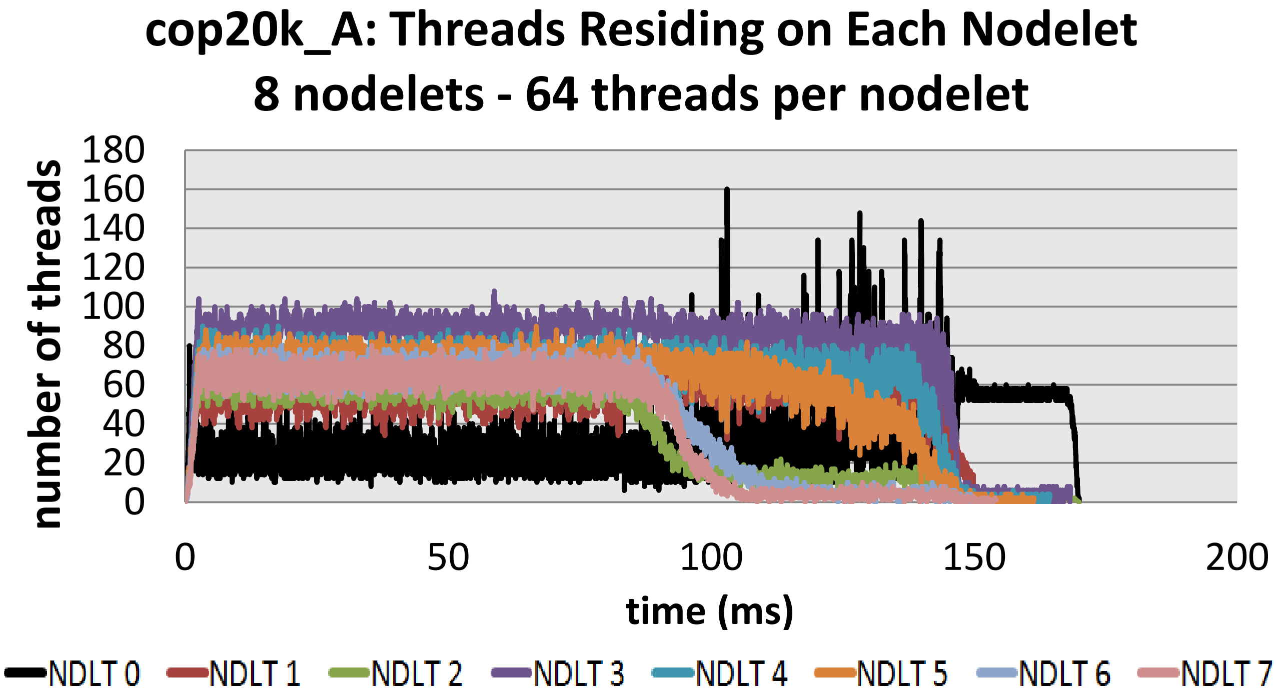
4.) Results: **Hardware Load Balancing**

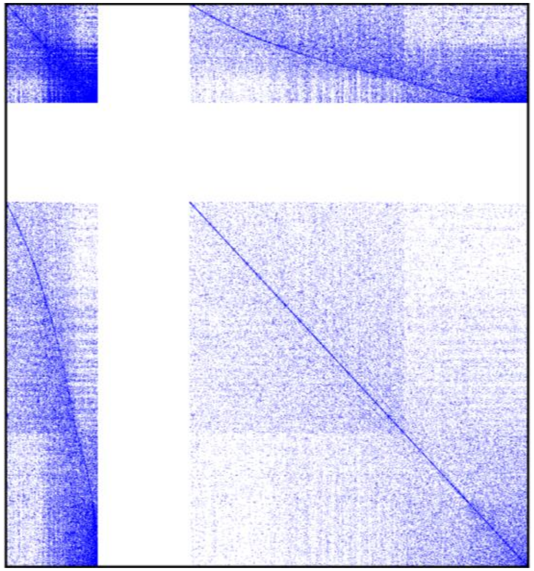
- Cannot isolate threads to hardware resources
  - Due to migratory nature of Emu threads
  - Data layout and memory access pattern dictate the load balancing of hardware
    - Very difficult to control for irregular algorithms

4.) Results: **Hardware Load Balancing**

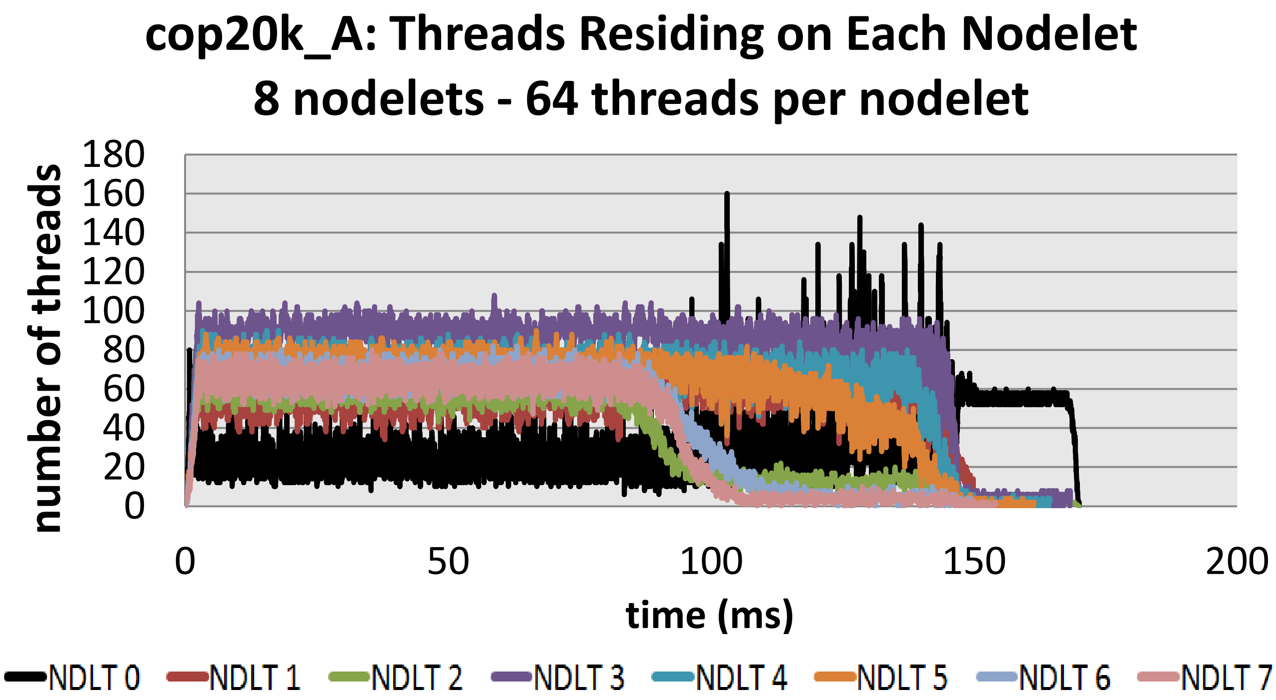
- Cannot isolate threads to hardware resources
  - Due to migratory nature of Emu threads
  - Data layout and memory access pattern dictate the load balancing of hardware
    - Very difficult to control for irregular algorithms
  - Hot-spots can form despite best efforts to evenly distribute work
    - Example: cop20k_A

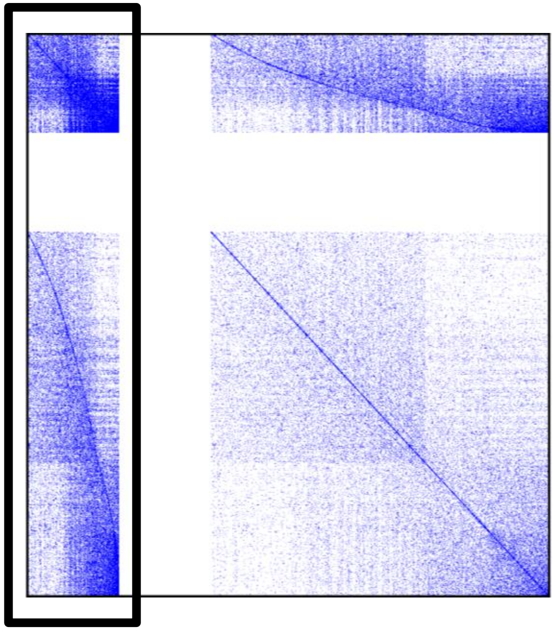
4.) Results: Hardware Load Balancing (cont.)





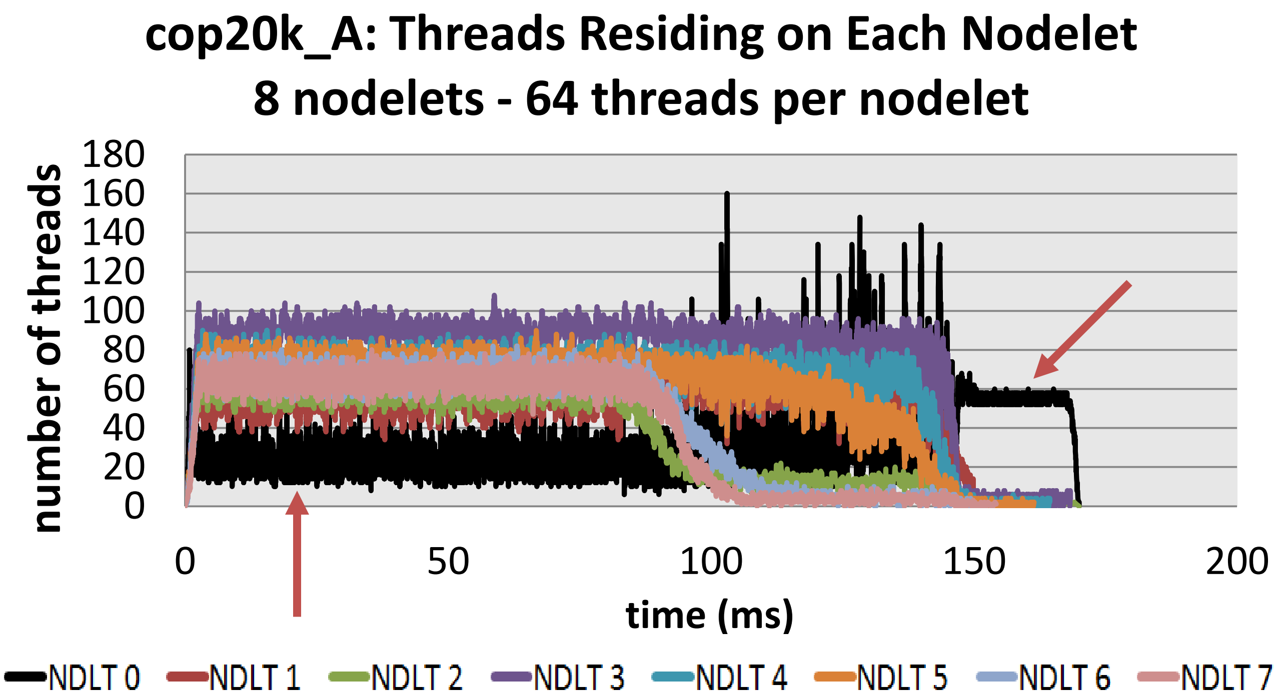
4.) Results: Hardware Load Balancing (cont.)

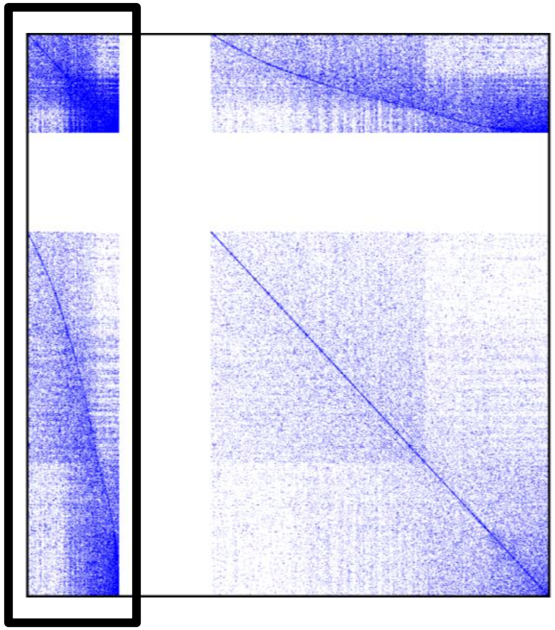




- 25% of the non-zeros require access to elements of **x** that are on nodelet 0 → majority of threads converge on nodelet 0 at roughly same time

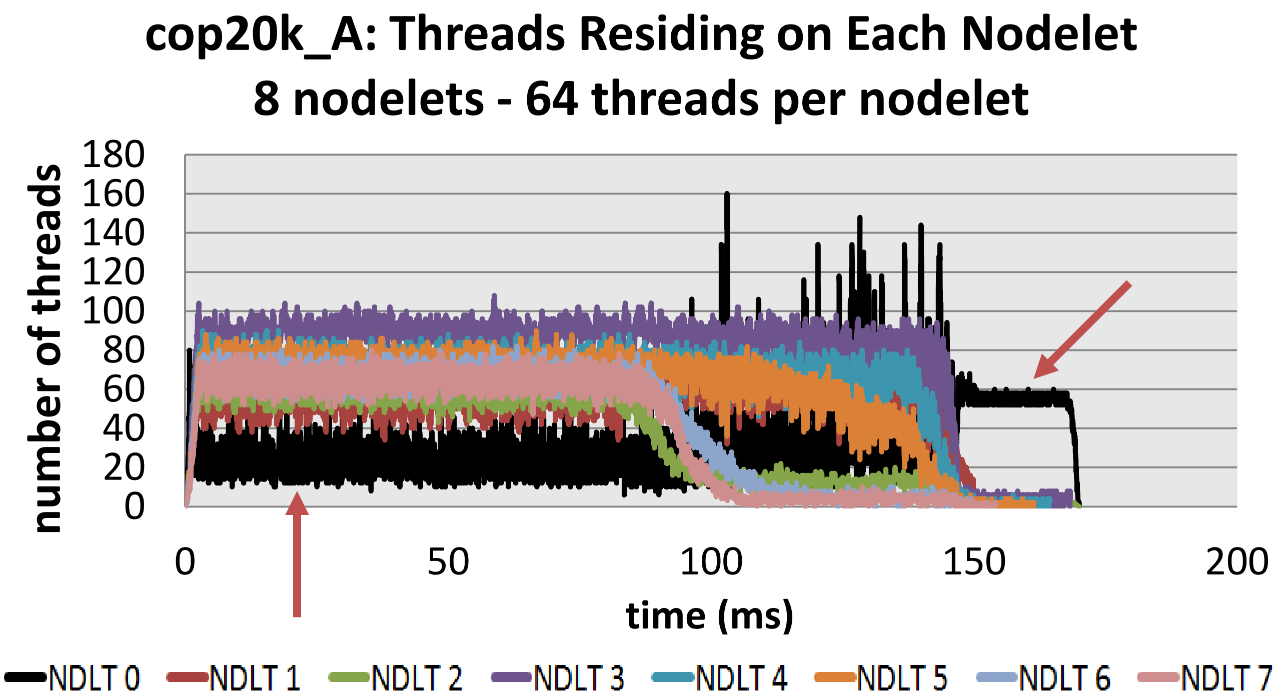
4.) Results: Hardware Load Balancing (cont.)

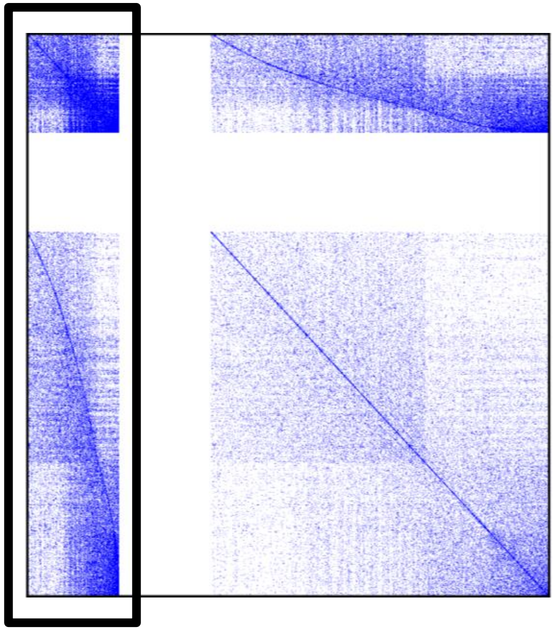




- 25% of the non-zeros require access to elements of **x** that are on nodelet 0 → majority of threads converge on nodelet 0 at roughly same time
- Nodelet 0 cannot main high thread activity
  - migration queue becomes swamped immediately
  - Emu currently throttles # of active threads based on resource availability on nodelet (i.e., queue sizes)

4.) Results: Hardware Load Balancing (cont.)





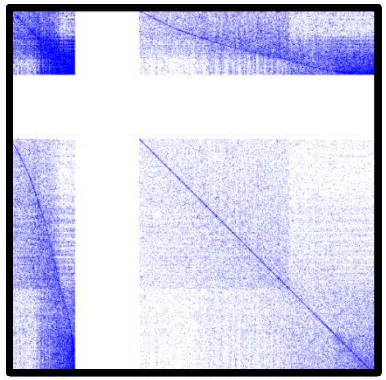
- 25% of the non-zeros require access to elements of **x** that are on nodelet 0 → majority of threads converge on nodelet 0 at roughly same time
- Nodelet 0 cannot main high thread activity
  - migration queue becomes swamped immediately
  - Emu currently throttles # of active threads based on resource availability on nodelet (i.e., queue sizes)
- Load balancing drastically improved by running with fewer nodelets/threads
  - suggests that the load imbalance issue will persist/be worse in multi-node execution

4.) Results: Matrix Reordering

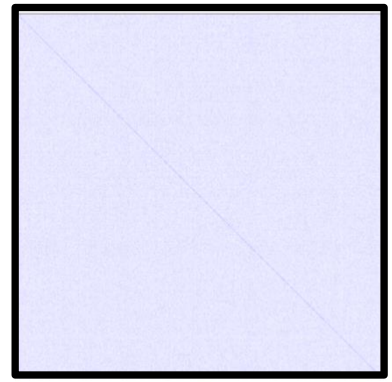
- Question: can known matrix reordering techniques offer performance gains, and mitigate hardware load balancing issues?
- We looked at
  - Breadth First Search (BFS)
  - METIS
  - Randomly permute rows/columns

4.) Results: Matrix Reordering (cont.)

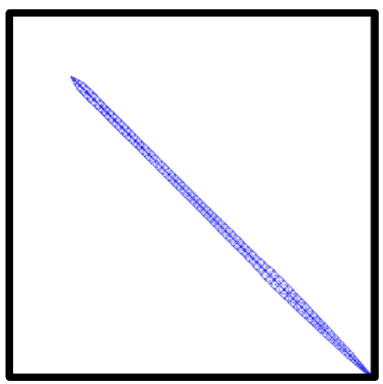
- cop20k_A matrix when reordered



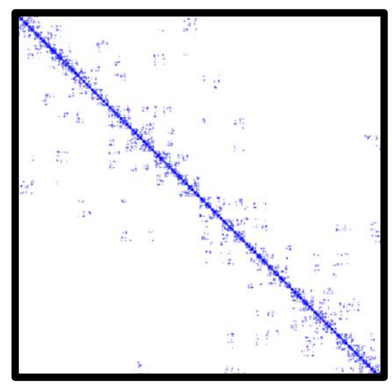
NONE



RANDOM

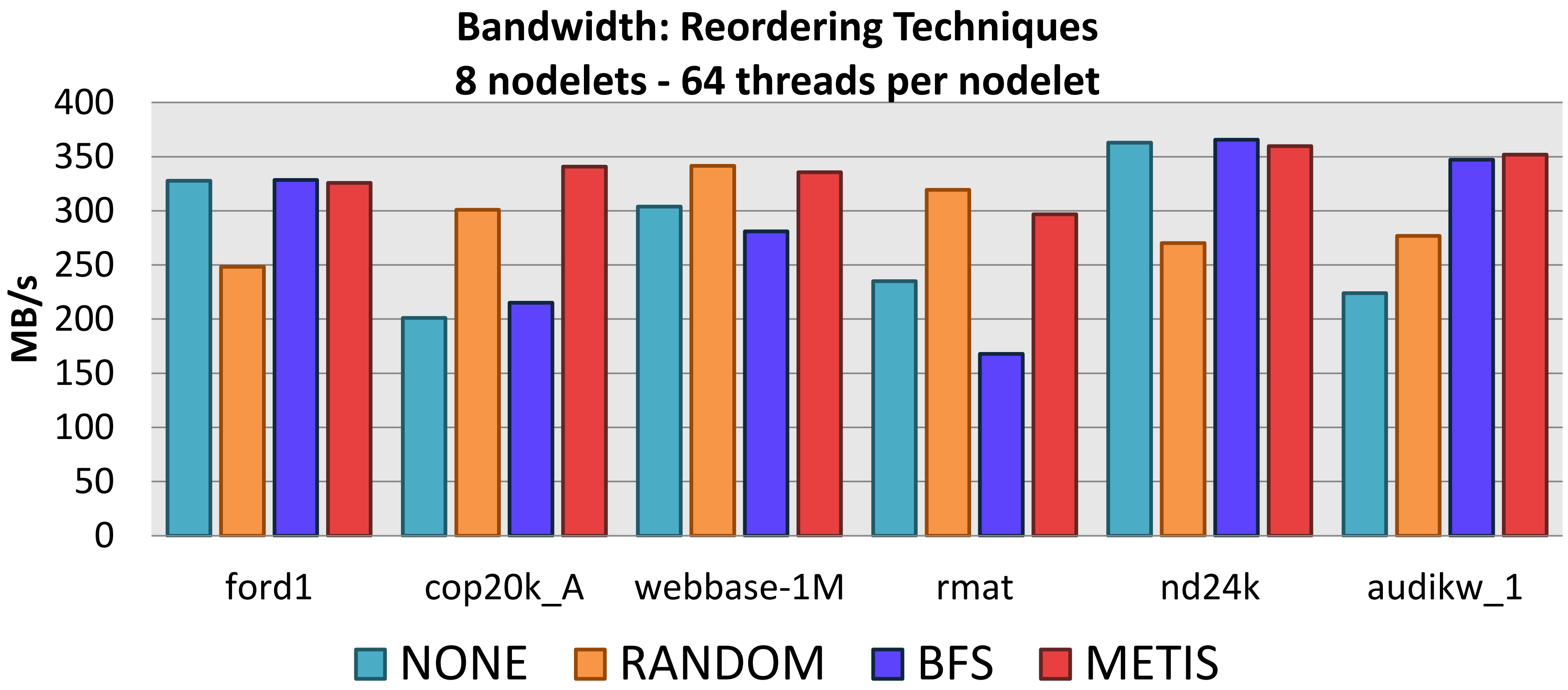


BFS

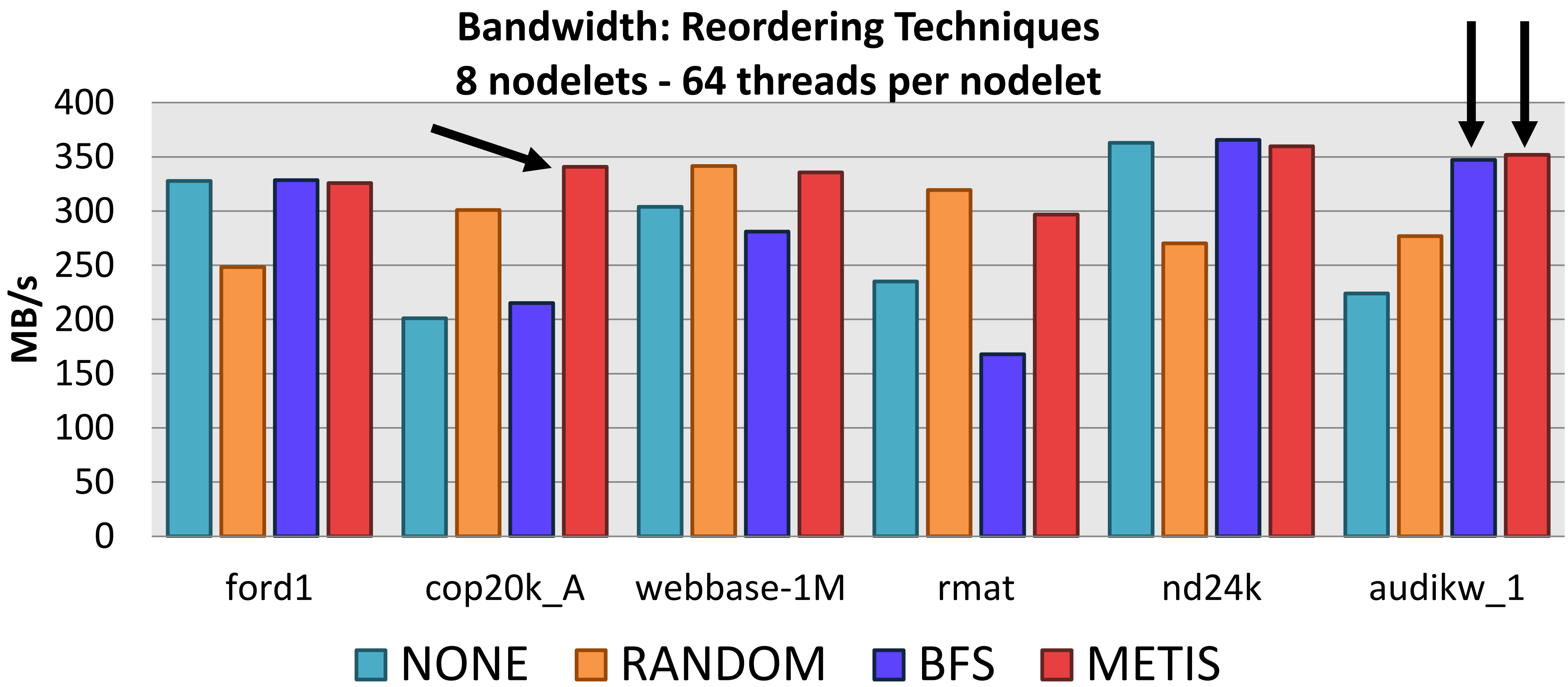


METIS

4.) Results: Matrix Reordering (cont.)

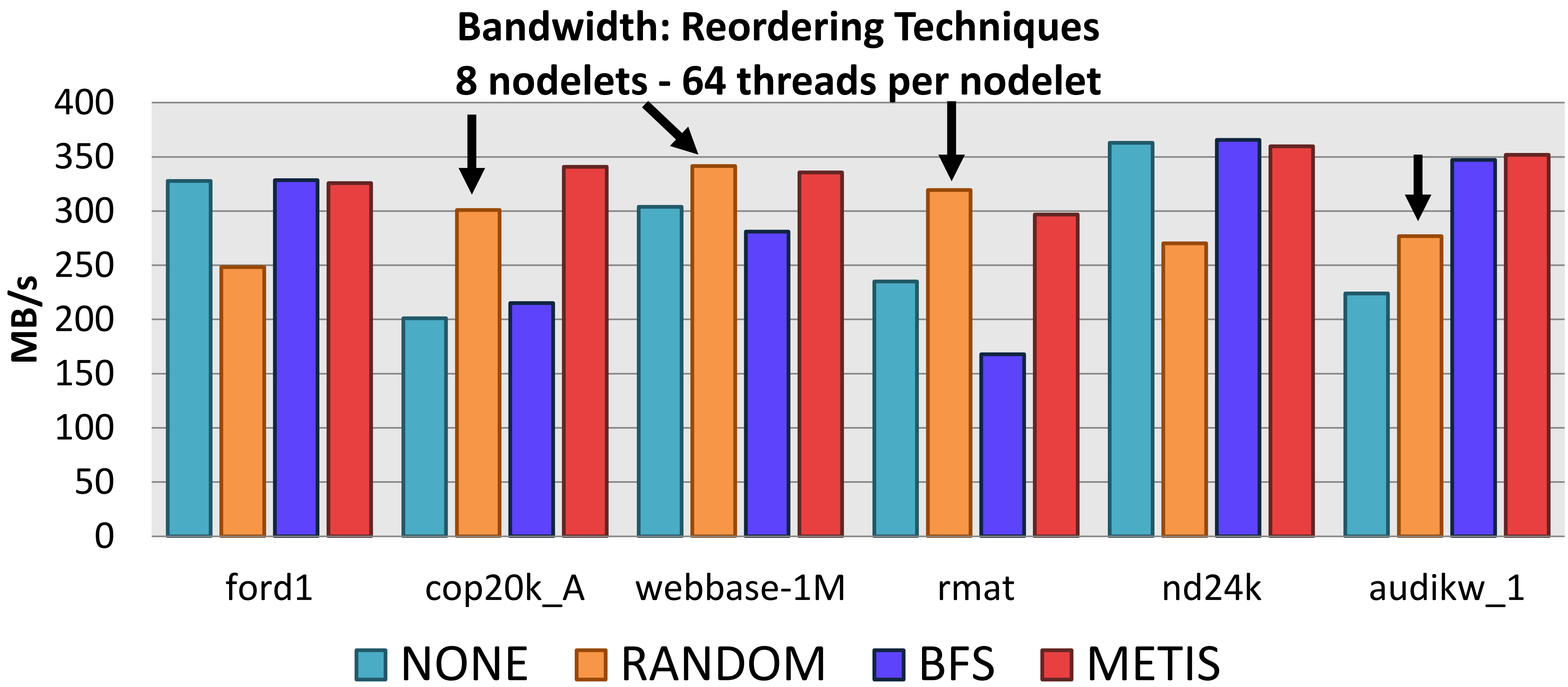


4.) Results: Matrix Reordering (cont.)



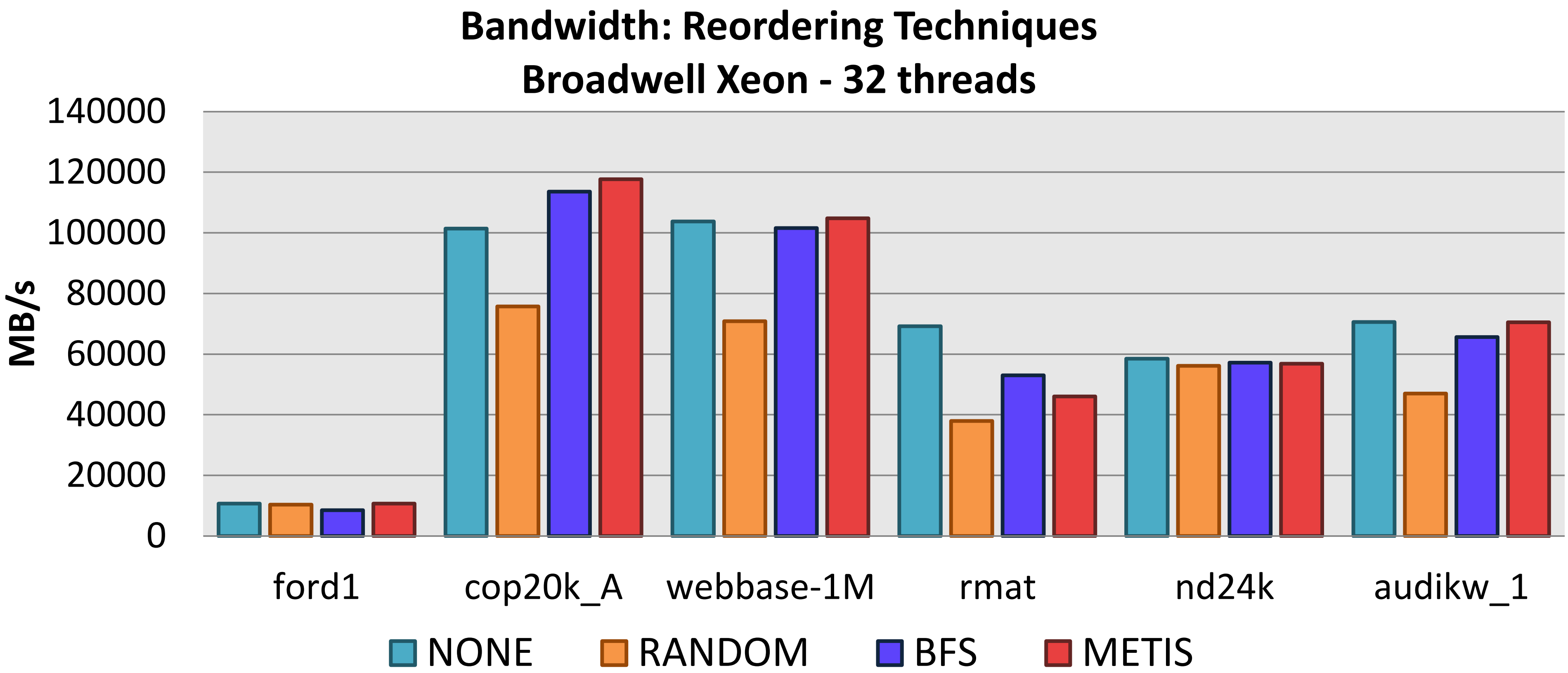
- BFS and METIS provide up to **70%** more BW over original
  - tend to cluster along main diagonal and produce balanced rows → reduces migrations and provides good load balancing

4.) Results: Matrix Reordering (cont.)



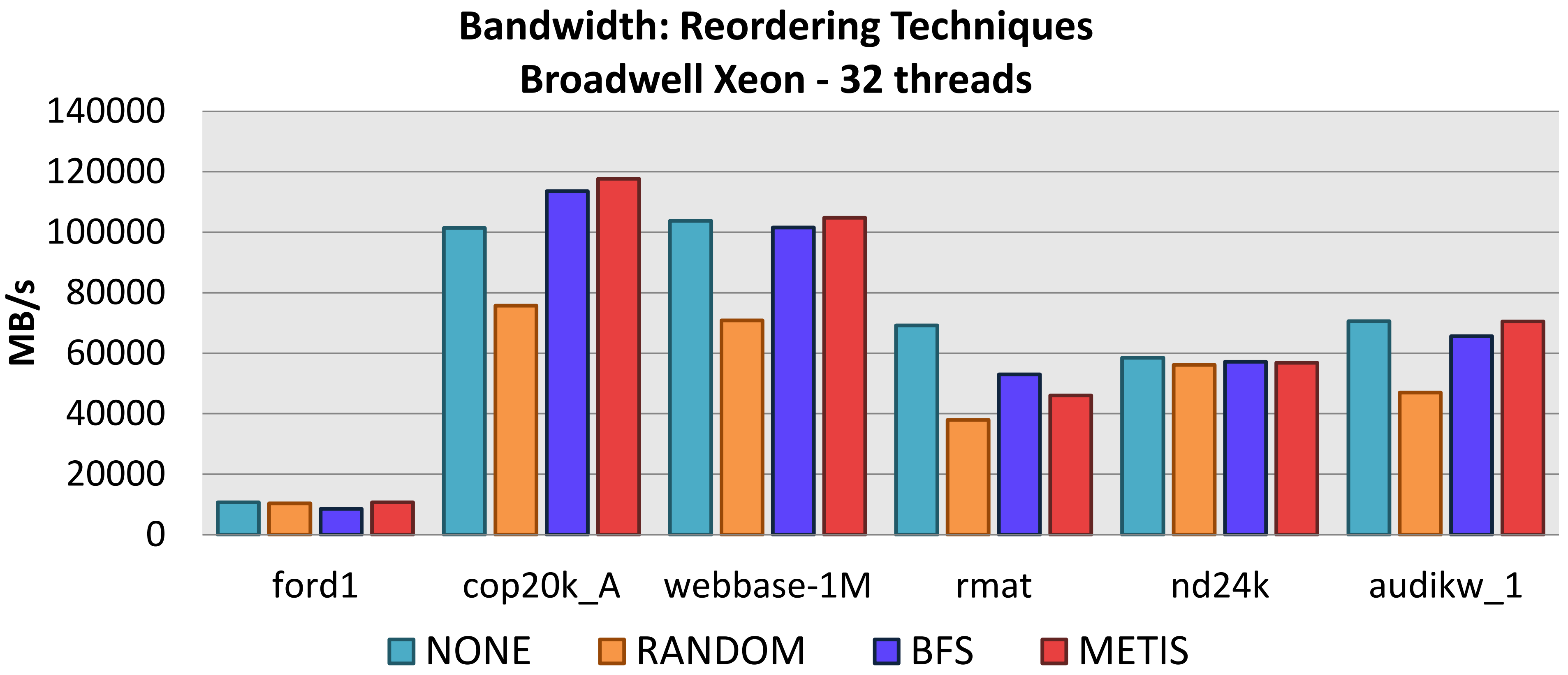
- BFS and METIS provide up to **70%** more BW over original
  - tend to cluster along main diagonal and produce balanced rows → reduces migrations and provides good load balancing
- Random offers up to **50%** more BW over original
  - produces balanced rows by uniformly spreading out non-zeros
  - incurs many more migrations but provides “natural” hot-spot mitigation

4.) Results: Matrix Reordering (cont.)



- BFS and METIS only provide up to **16%** more BW over original on cache-memory based system

4.) Results: Matrix Reordering (cont.)



- BFS and METIS only provide up to **16%** more BW over original on cache-memory based system
- Random is never better than original, and is usually much worse
  - penalty of a cache miss is much more severe when compared to a migration on Emu

5.) Conclusions and Future Work

5.) Conclusions

- Minimizing migrations is generally a good strategy on Emu, but work distribution and load balancing is of similar importance for high performance

5.) Conclusions

- Minimizing migrations is generally a good strategy on Emu, but work distribution and load balancing is of similar importance for high performance
- Very difficult to enforce explicit hardware load balancing on Emu due to migratory threads
  - data placement and memory access patterns entirely dictate the work performed by hardware resources

5.) Conclusions

- Minimizing migrations is generally a good strategy on Emu, but work distribution and load balancing is of similar importance for high performance
- Very difficult to enforce explicit hardware load balancing on Emu due to migratory threads
  - data placement and memory access patterns entirely dictate the work performed by hardware resources
- Matrix reordering on Emu has a larger impact on SpMV performance than traditional systems
  - **70%** improvement on Emu Vs **16%** on x86
  - Random reordering performs very well on Emu

5.) Future Work

- Evaluate new hardware/software upgrades for Emu
  - faster GC clock, hot-spot mitigation improvements
- Run across multiple nodes
- Investigate other sparse storage formats
- Look closer at randomized data distributions (work by Valiant) and how it could be applied on Emu

Questions?

Work published at the 8th Workshop on Irregular Applications: Architectures and Algorithms (**IA^3**) for SC18

Contact: tbrolin@cs.umd.edu

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

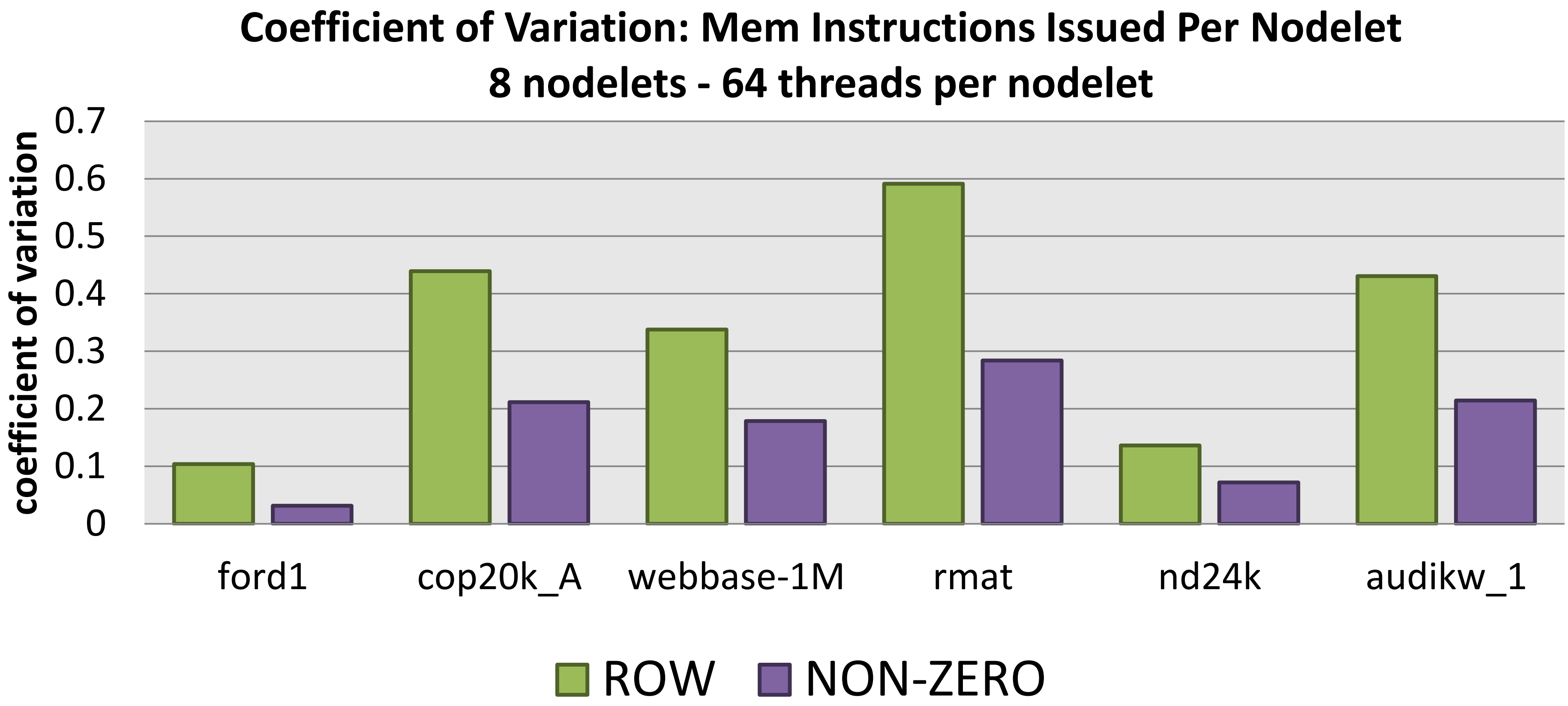
LPS
The Laboratory for Physical Sciences

Back up Slides

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

The Laboratory for Physical Sciences

4.) Results: Work Distribution (cont.)



- Coefficient of Variation (CV): stdev/mean
- Low CV for memory instructions issued per nodelet
  - indication of balanced work, as SpMV is memory bound
- Non-zero approach incurs an average of **1.69x** more migrations
  - suggests that proper load balancing can be more beneficial than reducing migrations

4.) Results: **Matrix Reordering (cont.)**