Optimizing Out-of-Core Nearest Neighbor Problems on Multi-GPU Systems Using NVLink

Rajesh Bordawekar
IBM T. J. Watson Research Center
bordaw@us.ibm.com

Pidad D'Souza
IBM Systems
pidsouza@in.ibm.com

Outline

- Out-of-core GPU Computations
- Problem Statement
- Out-of-core Nearest Neighbor Computations
  - Motivating Workload
  - Choosing the out-of-core algorithms
  - Multi-GPU Out-of-core Implementation
- Experimental Evaluation
  - Platforms: IBM Minsky and Comparable PCI-e based x86 systems
  - Methodology and Datasets
- Summary

Out-of-core GPU Computations

- **Definition:** ***Out-of-core*** computations refer to a class of problems in which either input data is too large to fit in the GPU's device memory or input data is partially available (e.g., *streaming* data)

- Out-of-core problems *are* **observed across a wide variety of application domains**:
  - Deep Learning: Training/Inference over large models
  - Big Data Analytics (e.g., Spark): Regression, Recommendation Systems, Clustering, Graph Analytics, ...
  - HPC: Seismic, Computational Chemistry, CFD, ...
  - Databases: Relational or NoSQL Databases, Spark
  - Streaming Data Analytics: Time-series processing for IoT workloads

Problem Under Consideration

- ***Problem Statement*:** How do we avoid performance degradation when end-to-end GPU performance and scalability is bound by memory allocation and data transfer costs?
- ***Solution:*** Only IBM's POWER Systems with the POWER8 processor and CPU-GPU NVIDIA NVLink improves performance and scalability of an out-of-core GPU-accelerated workload

What is NVLink 1.0?

- **CPU-GPU high-bandwidth Interface** with 19.2 GB/s peak uni-directional bandwidth per link
  - **IBM S822LC for HPC** (aka *Minsky*) has 2 NVLink links per CPU-GPU connection with 38.4 GB/s peak uni-directional bandwidth

Results with IBM POWER Systems 822LC (Minsky)

- **For a GPU-accelerated workload, without any code change, we observe on average 2.5X performance improvement when using CPU-GPU NVLINK over PCI-e based systems**
  - Results observed across various implementations

- ***At least 3X in allocation time is saved*** due to the POWER8 CPU and its memory subsystem

- With code optimizations for NUMA CPU data affinity, **we achieve scalability up to 4 GPUs for out-of-core workloads** when scaling on IBM's Power System

Coupling the POWER8 CPU, its memory subsystem and industry-unique CPU-GPU NVLink, IBM Power Systems drastically improves performance and reduces the impact of I/O costs on scalability as the number of GPUs is increased.

Motivating Workload: Cognitive Database Queries

- Cognitive relational database uses an ***unsupervised*** neural network model (Word Embedding) to capture relationships in the database tokens and uses the model to answer novel SQL semantic queries. Currently implemented in Spark.
- The word embedding model represents each token in the database by a vector
- At runtime, the SQL engine executes each semantic query via evaluating similarities between vectors corresponding to the relational tokens
  - Vector similarity is computed using nearest neighbor algorithms
- Database tables can have millions of tokens
  - Models can be very large (multi-GB)
  - Efficient out-of-core nearest neighbor implementations critical for fast query execution

Bordawekar and Shmueli: arXiv:1603.07185, March 2016 and DEEM 2017

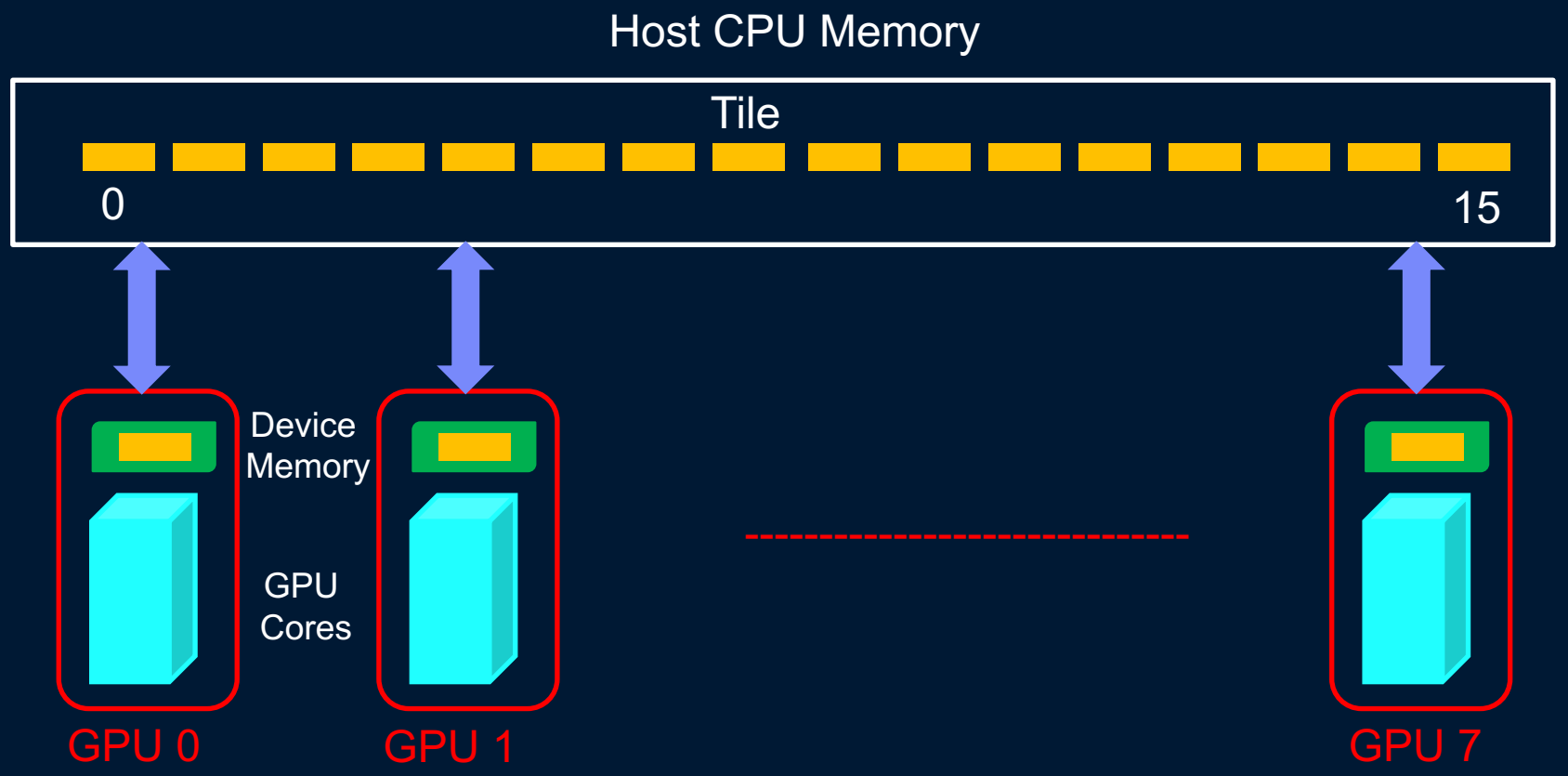
Nearest Neighbor Computations over Vectors

- Nearest Neighbor (NN) calculations involve two stages: distance-based similarity calculations and finding a pair with minimum distances
- Vector similarity computed using the cosine distance between the vectors as a metric
  - Similarity varies from 1 (closest) to -1 (farthest)
- Can be implemented as a dot-product between two normalized vectors
  - Generalized as matrix multiplication for computing distances between two sets of vectors
- For very large datasets, e.g., in cognitive databases, the NN calculations will require multiplying the entire out-of-core matrix
- To reduce the costs, approximate NN algorithms are needed

Approximate Nearest Neighbor (ANN) Algorithms

- Use approximate NN algorithm to identify candidates and then use expensive matrix multiplication algorithm on the smaller dataset to compute exact results
- Requirements for the approximate Nearest Neighbor algorithms
  - Good accuracy, smaller memory footprint, easy to partition and parallelize on GPUs
- Two alternatives:
  - **Locality Sensitive Hashing** (LSH) using signed random projections (SRP-LSH)
    - Project each input vector onto k random planes. Use the resulting k element signature to approximate cosine distance between vectors [Charikar, 2002]
    - Map the signature to a hash bucket to cluster similar vectors
  - **Spherical K-Means** [Dhillon and Modha, 2001 and Hornik, 2012]
    - Cluster the vectors into a pre-determined number of classes using cosine distance as the clustering metric
    - Vectors in a cluster are very close (higher cosine distance) to the centroid vector

Multi-GPU Out-of-core Implementation



Computational *Strip-mining:* Each GPU fetches input data in smaller chunks (*tiles*) based on available device memory

Given available memory of 1 tile per GPU:
16 accesses for 1 GPU, 8 each for 2 GPUs,…, 2 each for 8 GPUs

Multi-GPU Out-of-core Implementations: SRP-LSH

- Input: An out-of-core feature matrix with *n* vectors of dimension *d*
- Output: *n*-element array representing hash positions for each vector
- Three stages:
  1. Project each vector of dimension d onto k random planes of dimension d, to compute a k element projection
  2. Convert the k-element projection to a k-element binary signature
  3. Hash the k-element signature
- Out-of-core Algorithm
  - Partition the input feature matrix into *k* tiles of *s* samples (last tile can have different size)
  - For each tile:
    1. Project all vectors in the tile by multiplying it by the in-core random planes matrix [k,d]. Use in-core CUBLAS Sgemm function
    2. Convert each of the *s* results to *s* k-element signatures
    3. Map each signature to a hash location value (steps 2 and 3 implemented by a single kernel)
- Overall algorithm embarrassingly-parallel, each tile can be executed independently
- Similar approach used for the naïve matrix multiplication scenario
  - Normalize vectors in each feature tile and do the multiplication with input vectors

Multi-GPU Out-of-core Implementations: Spherical K-Means

- Input: An out-of-core feature matrix with *n* vectors of dimension *d*, number of clusters C, iteration count "I"
- Output: Map of size *n* storing target cluster index
- Three stages: Initial clustering, finding new clusters, update total weights
- **Focus on the most expensive component: finding new clusters**
- Out-of-core Algorithm
  - Partition the input feature matrix into *k* tiles of *s* samples (last tile can have different size)
  - For each tile:
    1. Normalize each vector in the tile
    2. Compute cosine distances of each vector with *C* cluster centroid vectors using CUBLAS Sgemm
    3. For each vector, compute the largest distance among the *C* computed distance. Return the offset as the index of the next cluster
- Overall algorithm embarrassingly-parallel and each tile can be executed independently
  - Stage 2 implemented using in-core CUBLAS Sgemm
  - Stages 1 and 3 implemented as separate kernels

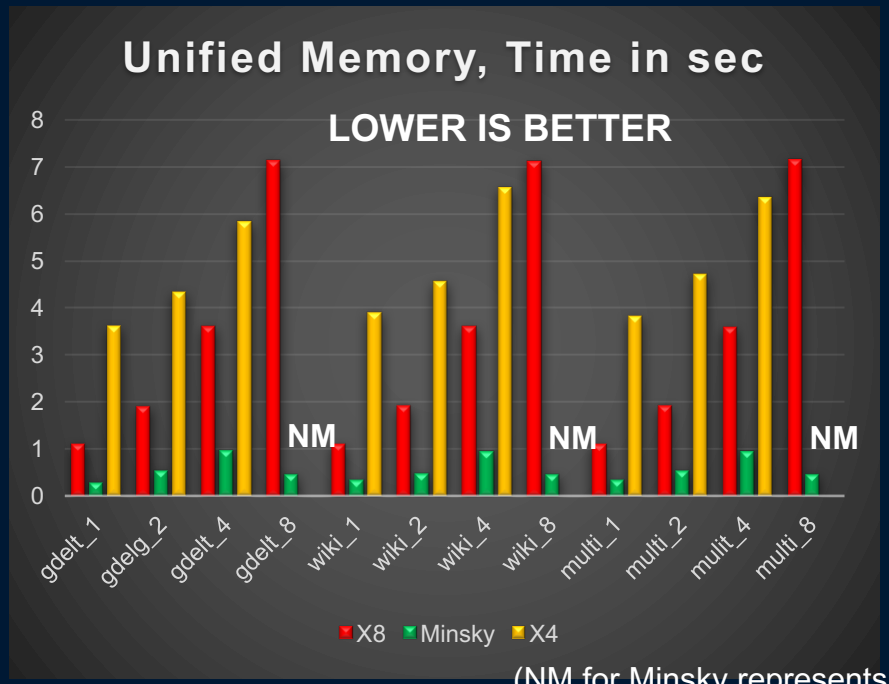
Experimental Evaluation: System Infrastructure

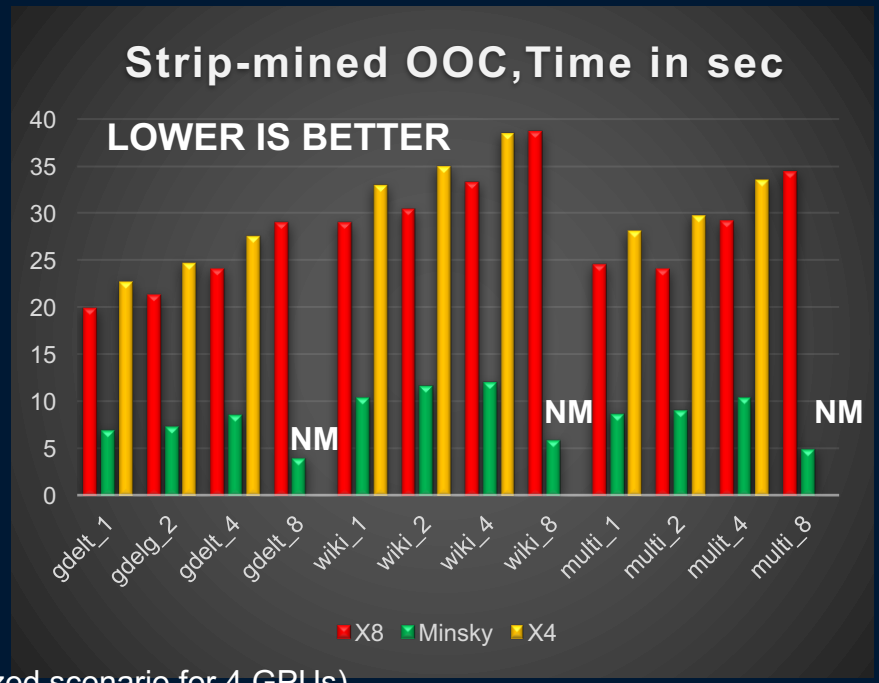
- 3 different multi-GPU Linux systems:
  - All using Pascal GPUs, each with 16 GB of device memory
  - Two x86 based: Both Dual Socket Xeons, X8 with 8 GPUs, X4 with 4 GPUs
  - Power8 based: IBM Minsky S822LC with Dual Socket Power 8s and 4 GPUs
- CPU-GPU Connectivity
  - Both x86 systems used PCIe Gen 3 x16 interconnect
  - IBM Minsky used NVLink 1.0 (2 links per CPU-GPU connection)
- All systems supporting CUDA 8.0
- Test data:
  - Three vector models built from datasets used in cognitive workloads
    - **GDELT:** vectors from an global events database with 10520822 vectors of dimension 300 (12 GB)
    - **WIKI**: vectors from Wikipedia with 5137305 vectors of dimension 1000 (20 GB)
    - **MULTI**: vectors from a multi-lingual dataset with 13120143 vectors of dimension 300 (15 GB)

Experimental Evaluation Methodology

- Each algorithm implemented using 2 approaches:
  - Unified Memory using CUDA 6+ Unified Memory primitives
    - ***cudaMallocManaged()*** used to allocate data structures shared across CPU and GPUs
  - Strip-mined Out-of-core implementation using explicit memory allocation & copy routines
    - ***cudaMallocHost()*** used to allocate host memory
- Each algorithm tested using three datasets, on the three machines: X8, X4, and Minsky
  - On each machine, number of GPUs varied from 1 to 4 or 8
  - On Minsky, we also evaluated NUMA-optimized code that ran 2 concurrent tasks assigned to two different sockets
- For each run, we measured
  - Allocation time: time for allocating memory regions
  - Execution time: compute time and memory allocation and copy time (for strip-mined case)
  - Total time = allocation time + execution time

Experimental Results: Scalability of MM Nearest Neighbor

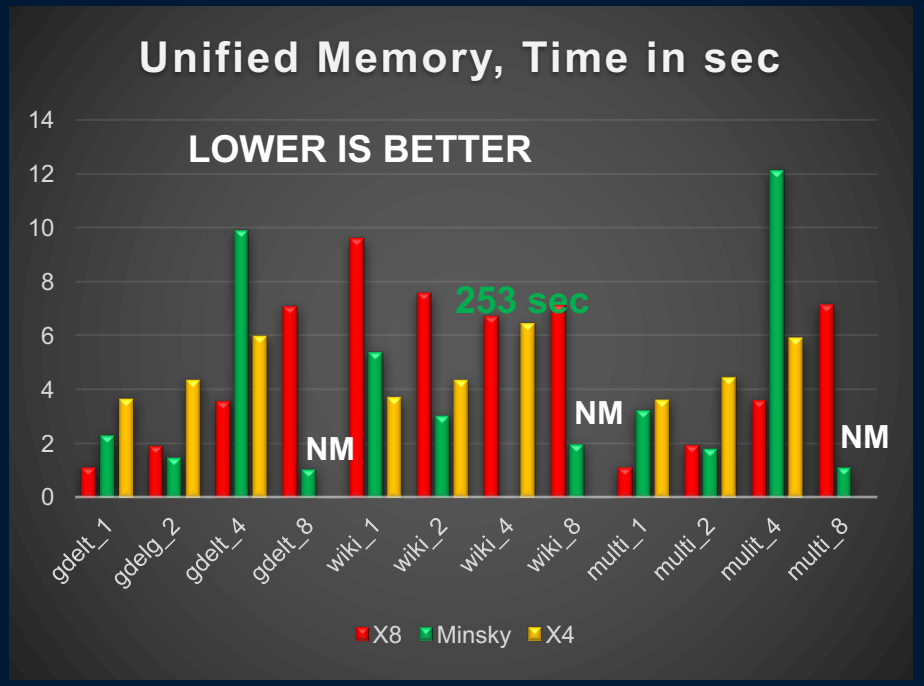


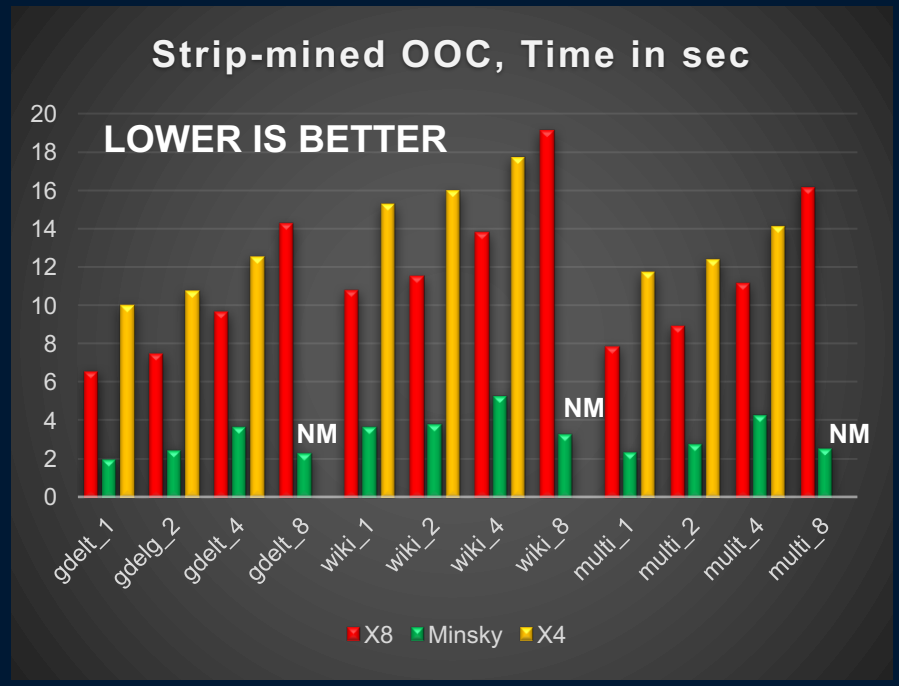


(NM for Minsky represents NUMA optimized scenario for 4 GPUs)

- GPU computation kernels executed concurrently by multiple GPUs
- End-to-end performance affected by I/O costs
- Performance does not scale as the number of GPUs is increased
- Unified Memory performs significantly better than the strip-mined approach
- In all cases, NVLink improves performance by at least 3X
- **With NUMA optimizations, 4 GPU performance scales on IBM Minsky**

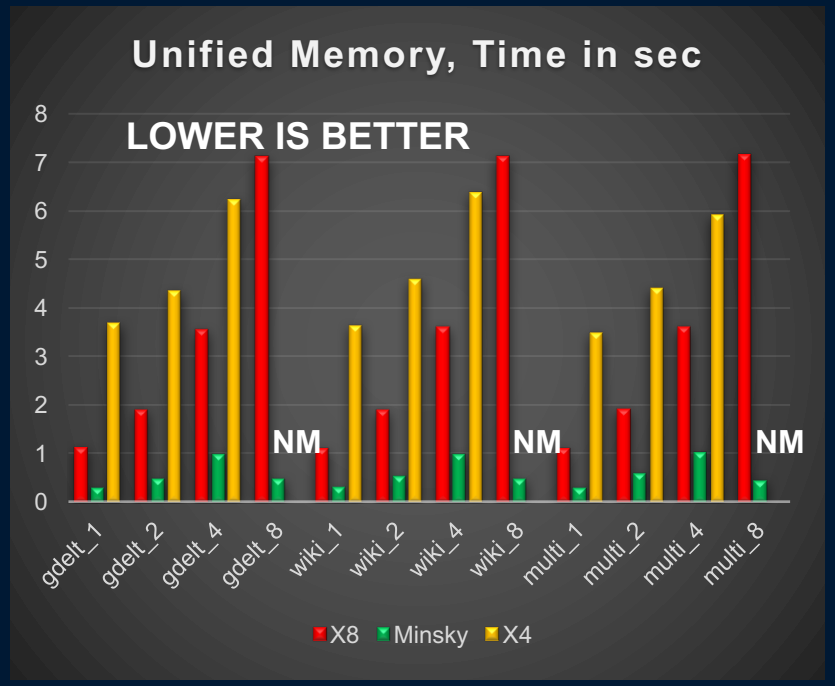
Experimental Results: Scalability of SRP-LSH Nearest Neighbor

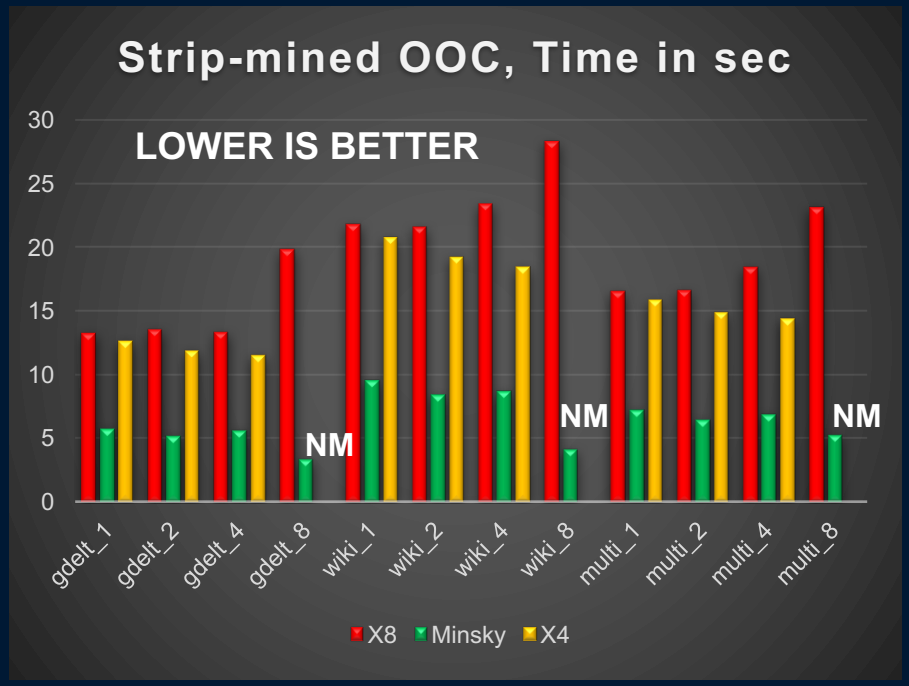




(NM for Minsky represents NUMA optimized scenario for 4 GPUs)

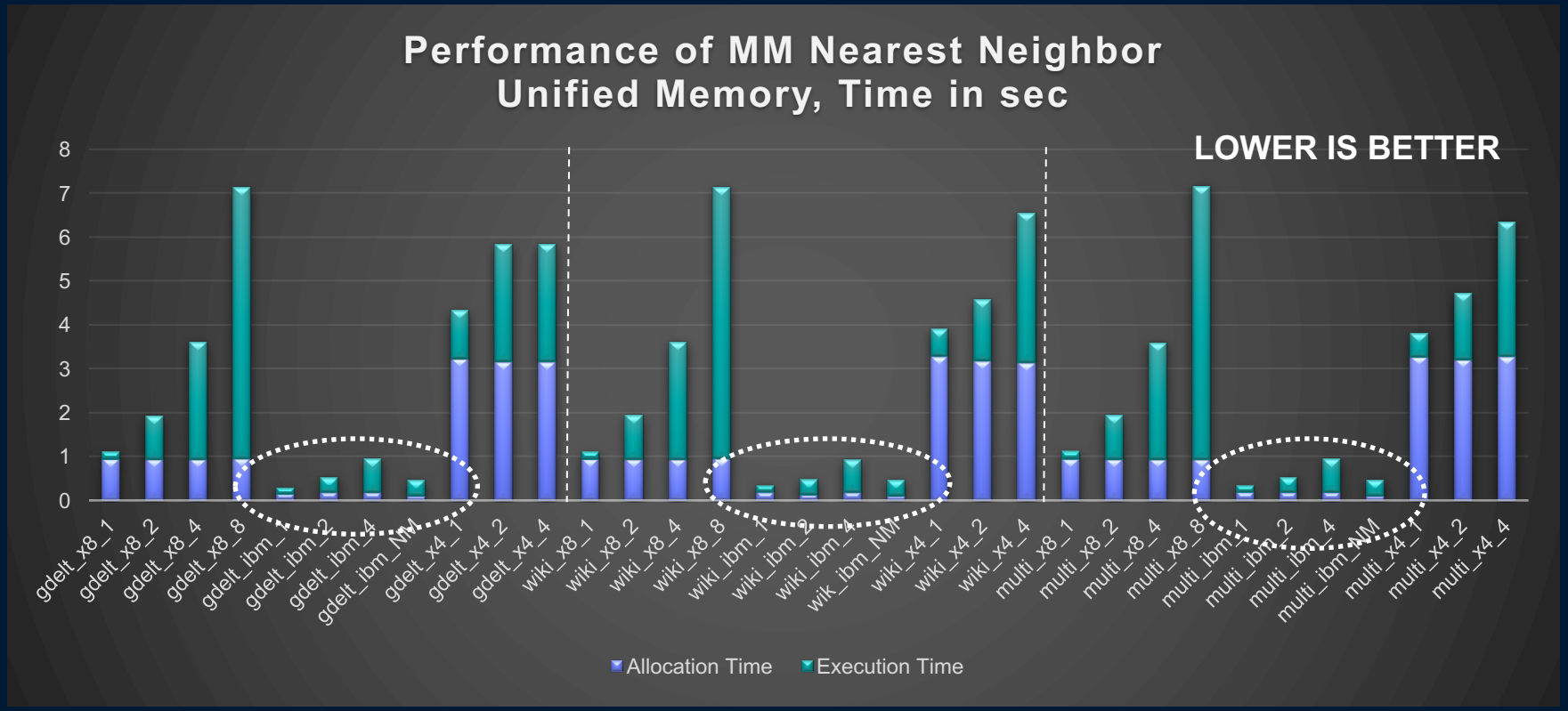
Experimental Results: Spherical K-Means Clustering





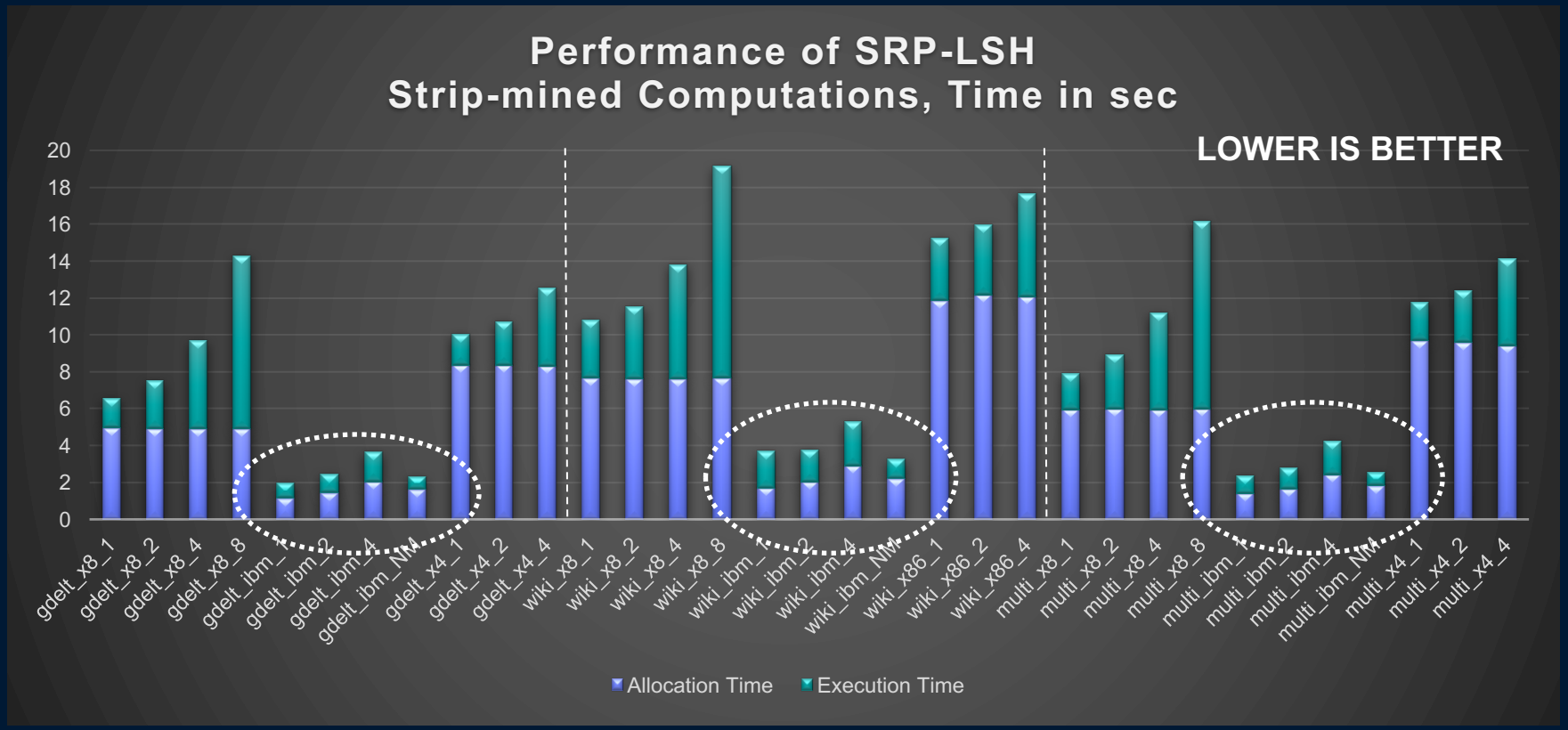
(NM for Minsky represents NUMA optimized scenario for 4 GPUs)

IBM Minsky Impact on Allocation and Execution Times



- In unified memory, overall performance is affected by allocation costs
- IBM Minsky's virtual memory manager (VMM) is able to improve allocations costs by up to 3X

IBM Minsky Impact on Allocation and Execution Times



For the strip-mined OOC scenario, both allocation and execution phase contribute to the overall performance. IBM Minsky is able improve both costs.

Experimental Results: Key Observations

- CUDA 6+ Unified Memory experiences
  - Very easy to use, simplifies programming tremendously
  - Does not scale as the number of GPUs is increased
  - Dependent on the Host OS memory management capabilities
- CUDA 6+ Unified Memory *always* performs better than the strip-mined OOC approach (using explicit I/O)
- In spite of GPU kernels executing concurrently, out-of-core applications do not scale across multiple GPUs
  - Performance affected by both memory allocation and copying costs
- Improving out-of-core performance requires support from both host CPU infrastructure and CPU-GPU networking fabric

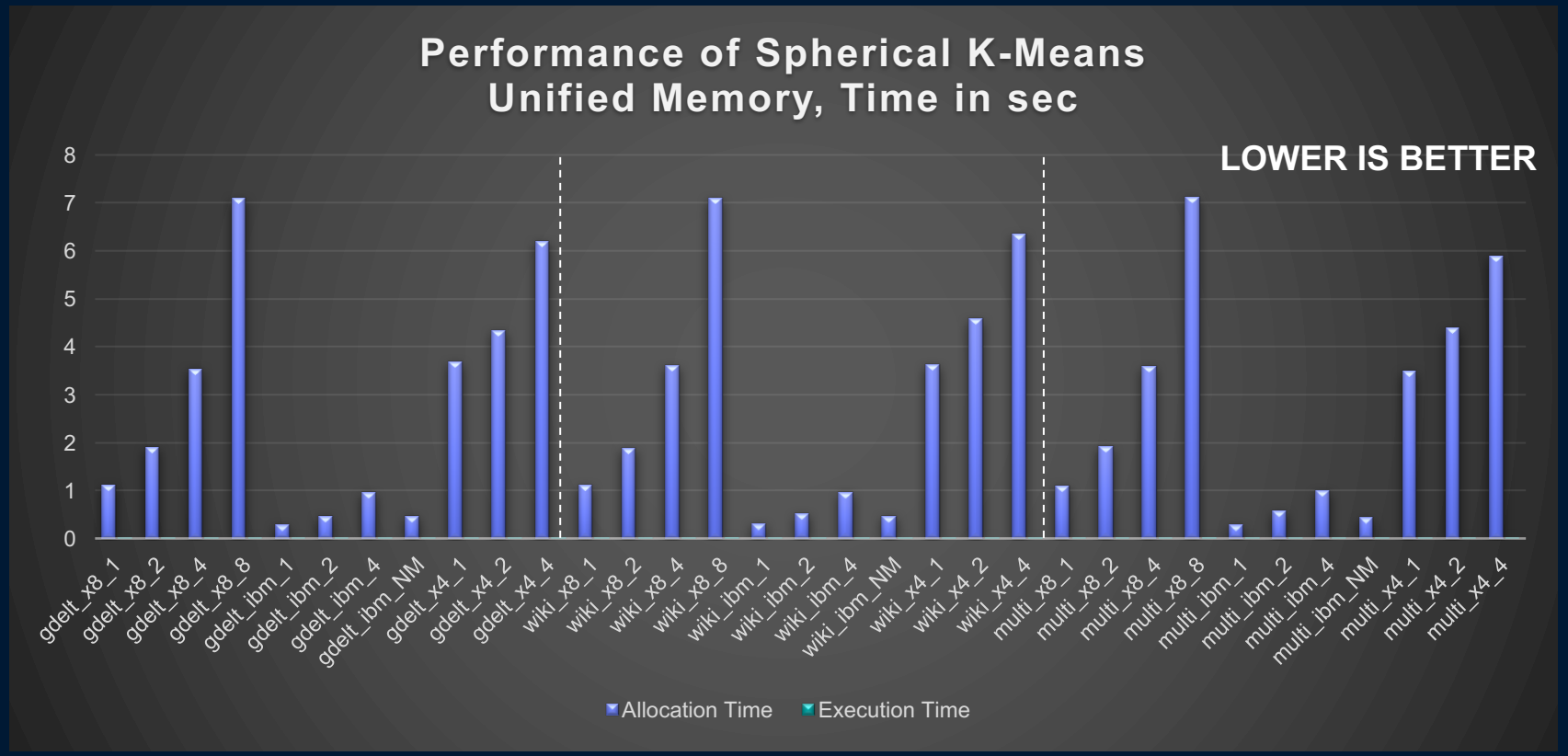
Summary

- Investigated performance of a key cognitive computation, Nearest Neighbor calculations, for out-of-core multi-GPU scenarios
- Evaluated three different implementations of Nearest Neighbor algorithms on both POWER and x86-based multi-GPU Pascal systems
- Analyzed performance of the algorithms using real datasets and identified key performance issues
- Demonstrated that IBM Minsky not only improves performance significantly (at least 2.5X) but achieves scalability for out-of-core multi-GPU scenarios

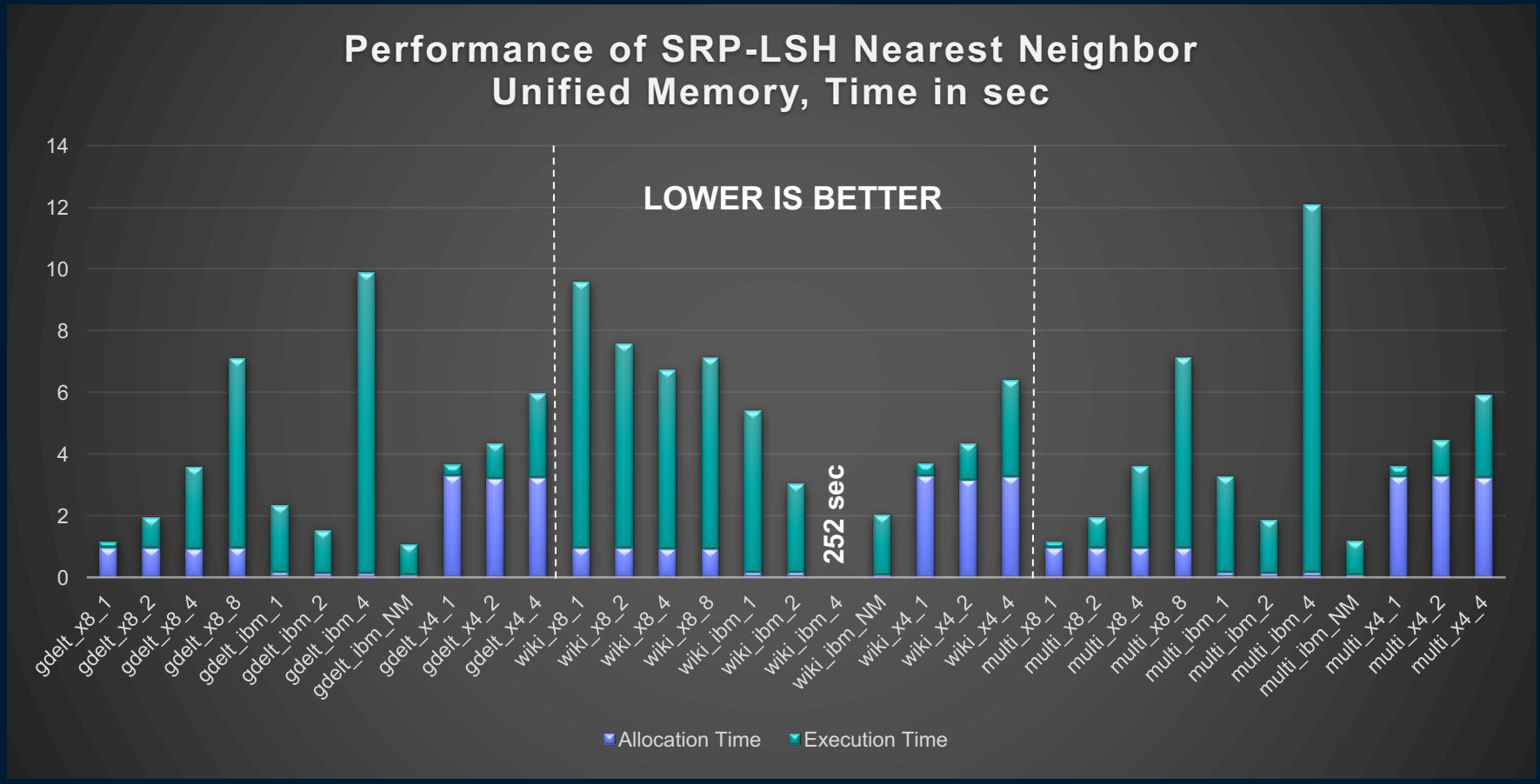
IBM Minsky is able to effectively use all three system components – CPU system stack, NVLink interconnect, and Pascal GPUs to deliver scalability and performance for out-of-core multi-GPU applications

Questions?

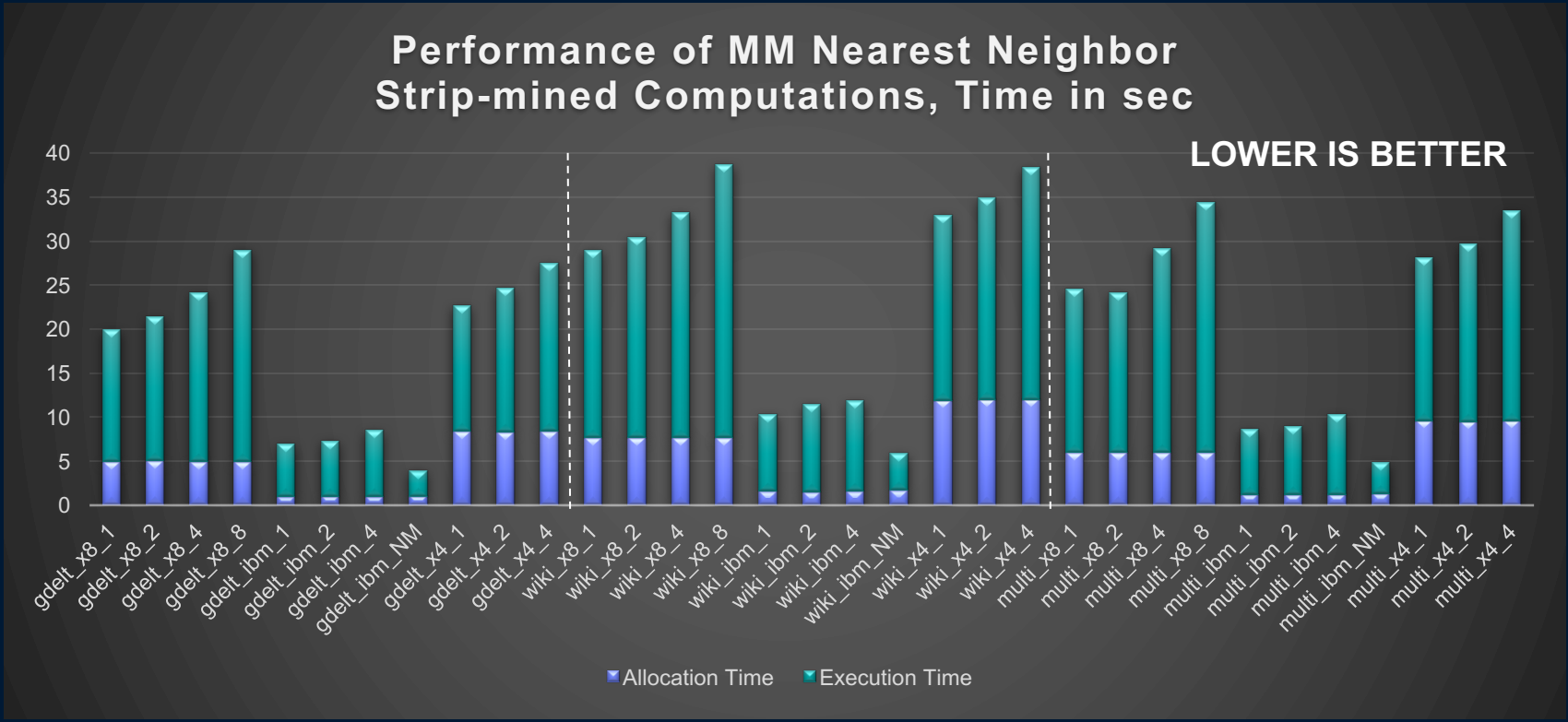
IBM Minsky Impact on Allocation and Execution Times



NVLink Impact on Allocation and Execution Times



NVLink Impact on Allocation and Execution Times



NVLink Impact on Allocation and Execution Times