A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing

Roknoddin Azizibarzoki

Seminar on Computer Architecture

Junwhan Ahn, Sungpack Hong*
Sungjoo Yoo, Onur Mutlu+
Kiyoung Choi

Seoul National University *Oracle Labs +Carnegie Mellon University

International **S**ymposium on **C**omputer **A**rchitecture **2015**

ETH zürich

Executive Summary

Problem: Performance of graph processing on conventional systems does not scale in proportion to graph size

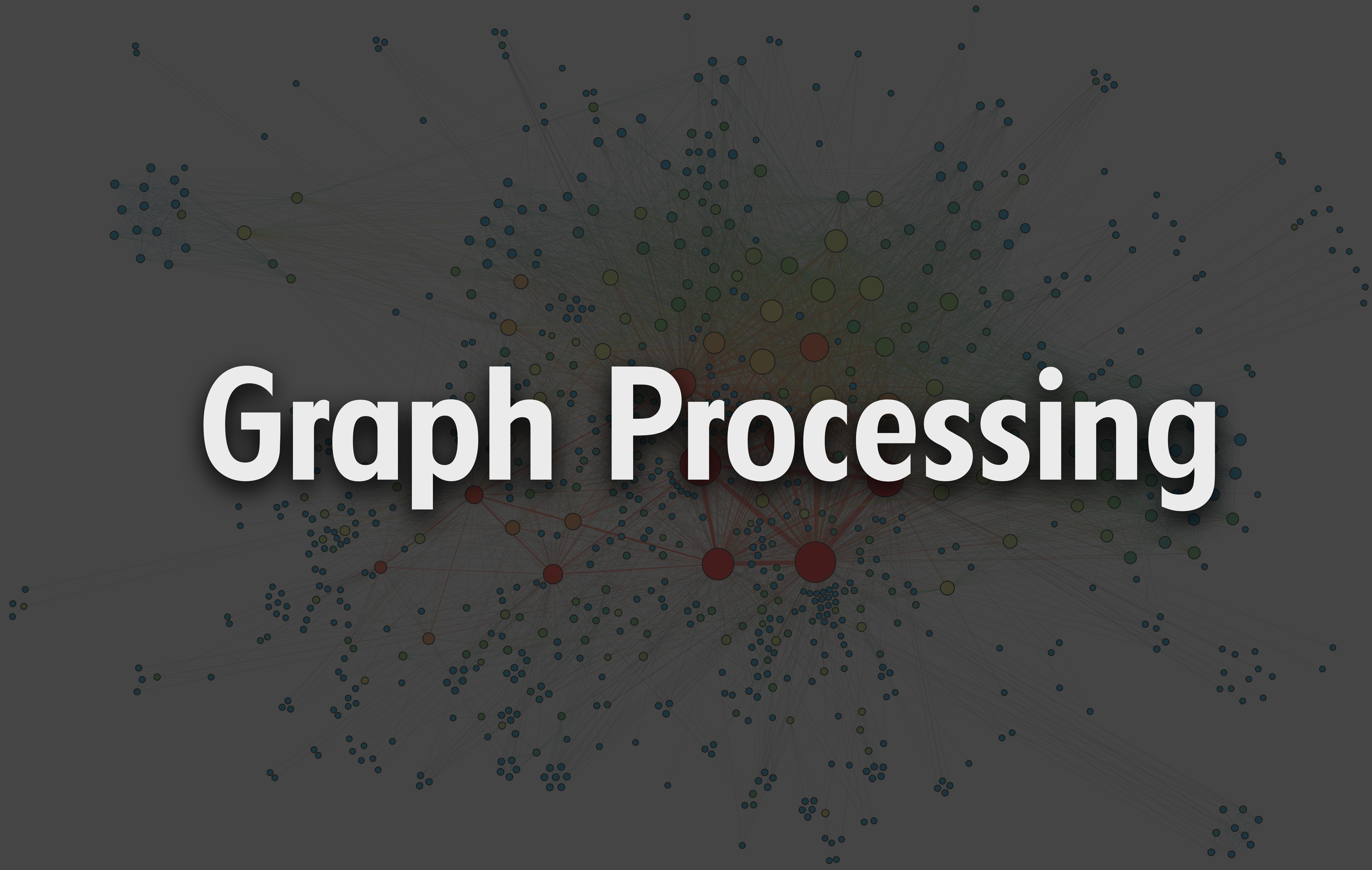
Goal: Design an infrastructure with scalable performance for graph processing

Observation: High memory bandwidth can sustain scalability in graph processing

Key Idea: Make use of Processing-In-Memory to provide high bandwidth, and design specially architected cores to utilize that bandwidth

Results: up to 13.8x performance improvement and 87% energy reduction

Graph Processing



Graphs

Abstractions used to represent objects and their relations

These representations can sometimes become very huge in real world applications

Graphs used in this paper can reach up to 200 million edges, 7 million vertices, and 3-5 GB of memory footprint

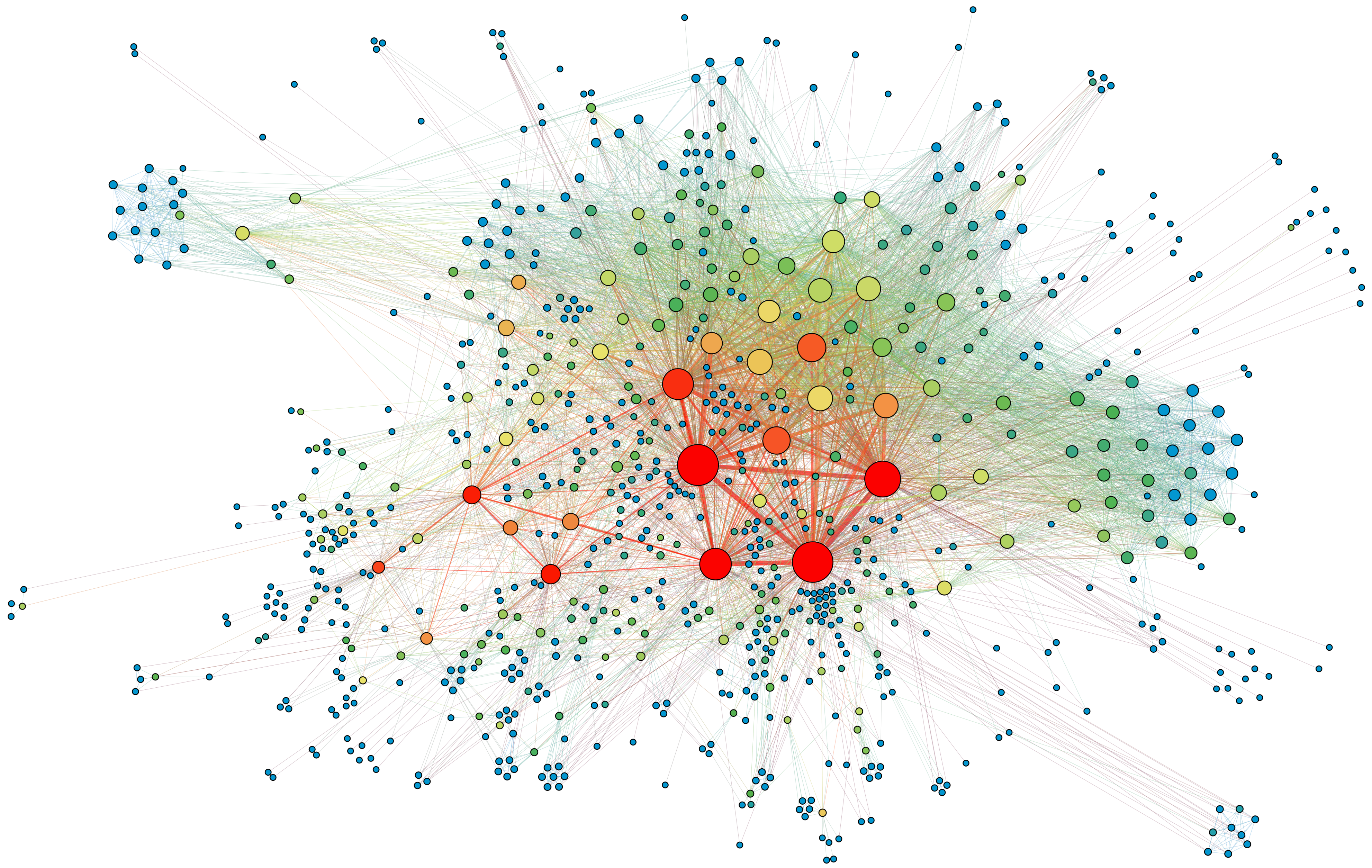


Image obtained from: Grandjean, Martin (2015), "Introduction à la visualisation de données, l'analyse de réseau en histoire", Geschichte und Informatik 18/19 (PDF), pp. 109–128

Graph Processing Workloads

Large … n other

Parallel computation almost independent for each vertex

Example: Page Rank
Originally designed to sort webpages based on number of views for Google, so as to do better webpage suggestions

```
for (v: graph.vertices):
   for (u: v.successors):
      u.new_rank = v.rank * weight
for (v: graph.vertices):
   v.rank = v.new_rank
   v.new_rank = alpha
```

Graph Processing Workloads Characteristics

Characteristics of this parallel, vertex independent computation:

1. Frequent random memory accesses

2. Small amount of computation per vertex

```
for (u: v.successors):
    u.new_rank = v.rank * weight
```

1

Each successor might lead you to a whole new subgraph

2

Simple multiplication computation

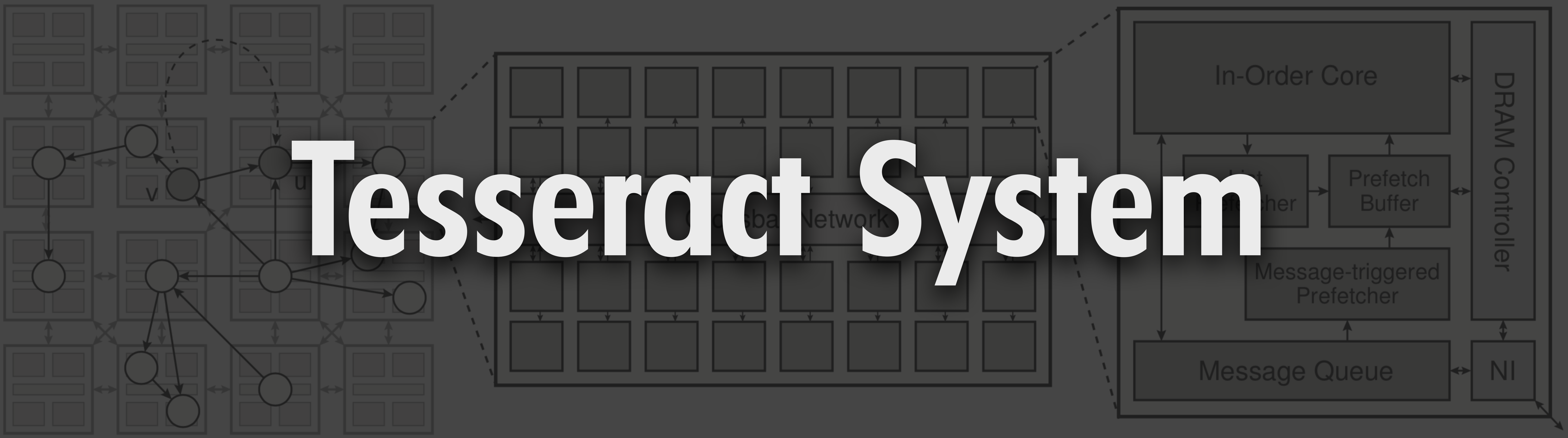
Graph Processing on Conventional Systems

INSIGHT:
High bandwidth can mitigate the performance bottleneck!

IDEA:
1. Let's use HMC based Processing-In-Memory to provide high bandwidth
2. And design specially architected cores to exploit this bandwidth
(Tesseract Cores)

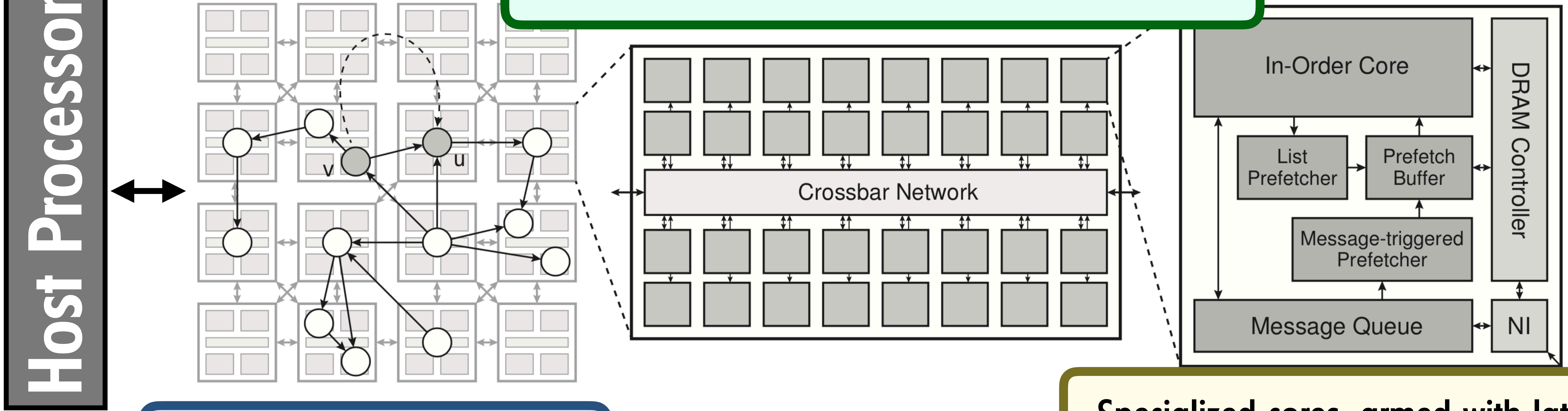
32 Cores DDR3 (102.4GB/s)	128 Cores DDR3 (102.4GB/s)	128 Cores HMC (640GB/s)	128 Cores HMC Internal BW (8TB/s)

Tesseract System



Tesseract System

- Each HMC cube contains 32 vaults, each armed with a simple in-order core in their logic layer, so that the cores can use HMC's internal BW
- Vaults communicating over a crossbar network for remote function calls



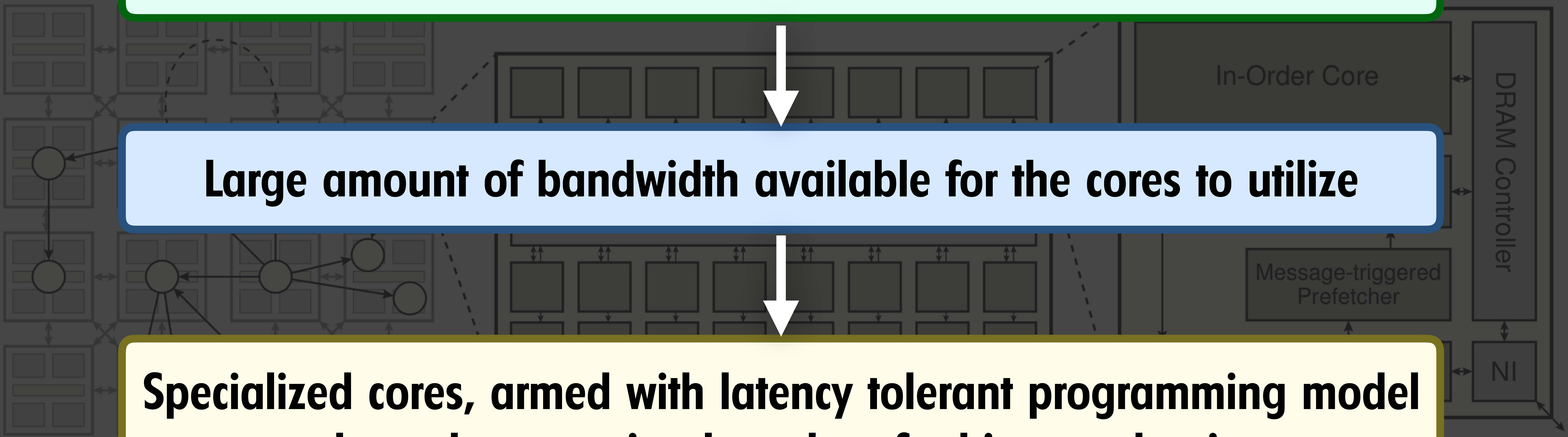
-A network of HMC cubes
-Memory mapped accelerator interface, non-cacheable, and no support for virtualization

- Specialized cores, armed with latency tolerant programming model and graph processing based prefetching mechanisms
- Message passing interface, prefetching mechanisms

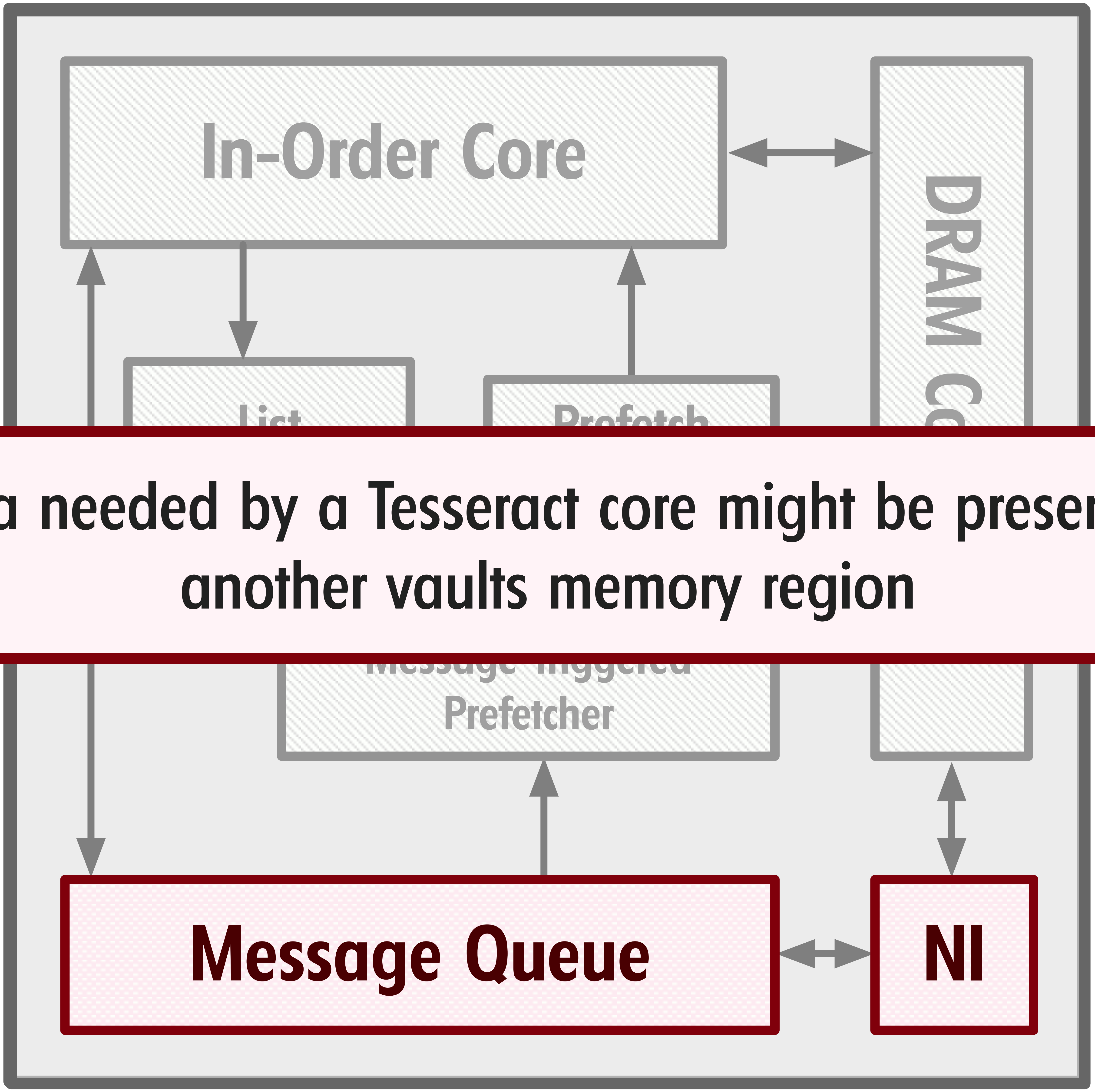
Processing-In-Memory with 3D stacked DRAM

Large amount of bandwidth available for the cores to utilize

Specialized cores, armed with latency tolerant programming model and graph processing based prefetching mechanisms



Communications in Tesseract



Data needed by a Tesseract core might be present in another vaults memory region

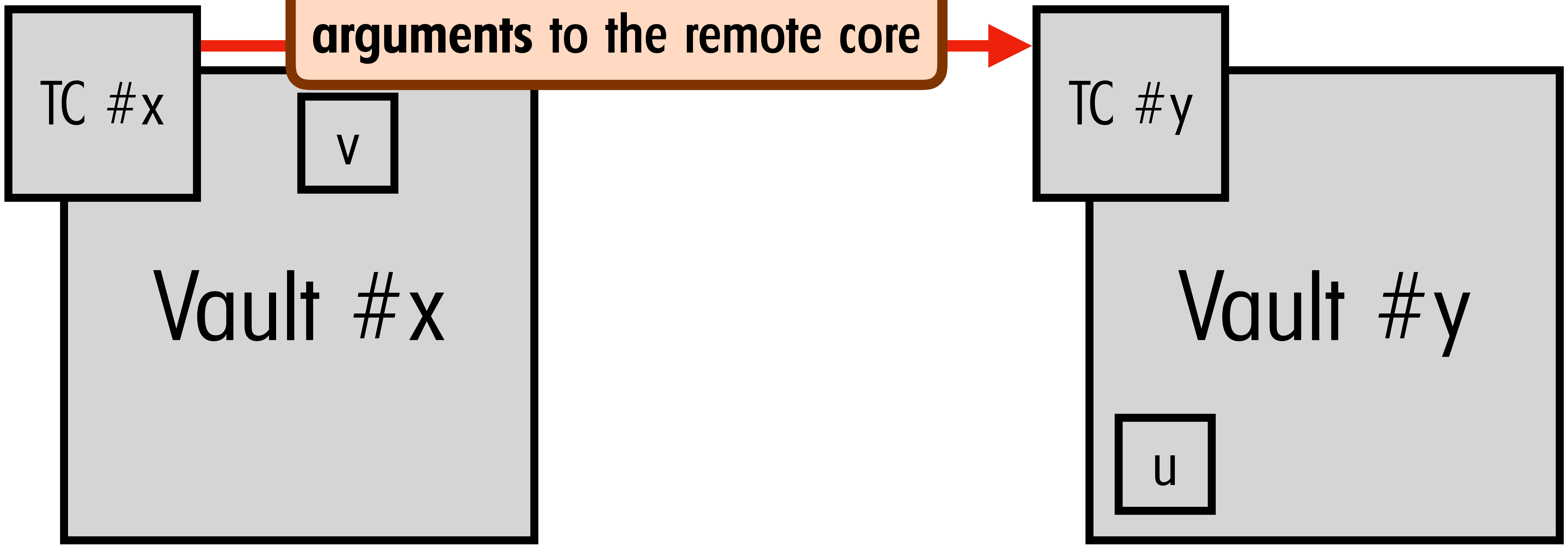
Communications in Tesseract

Data needed by a tesseract core might be present in another vaults memory region

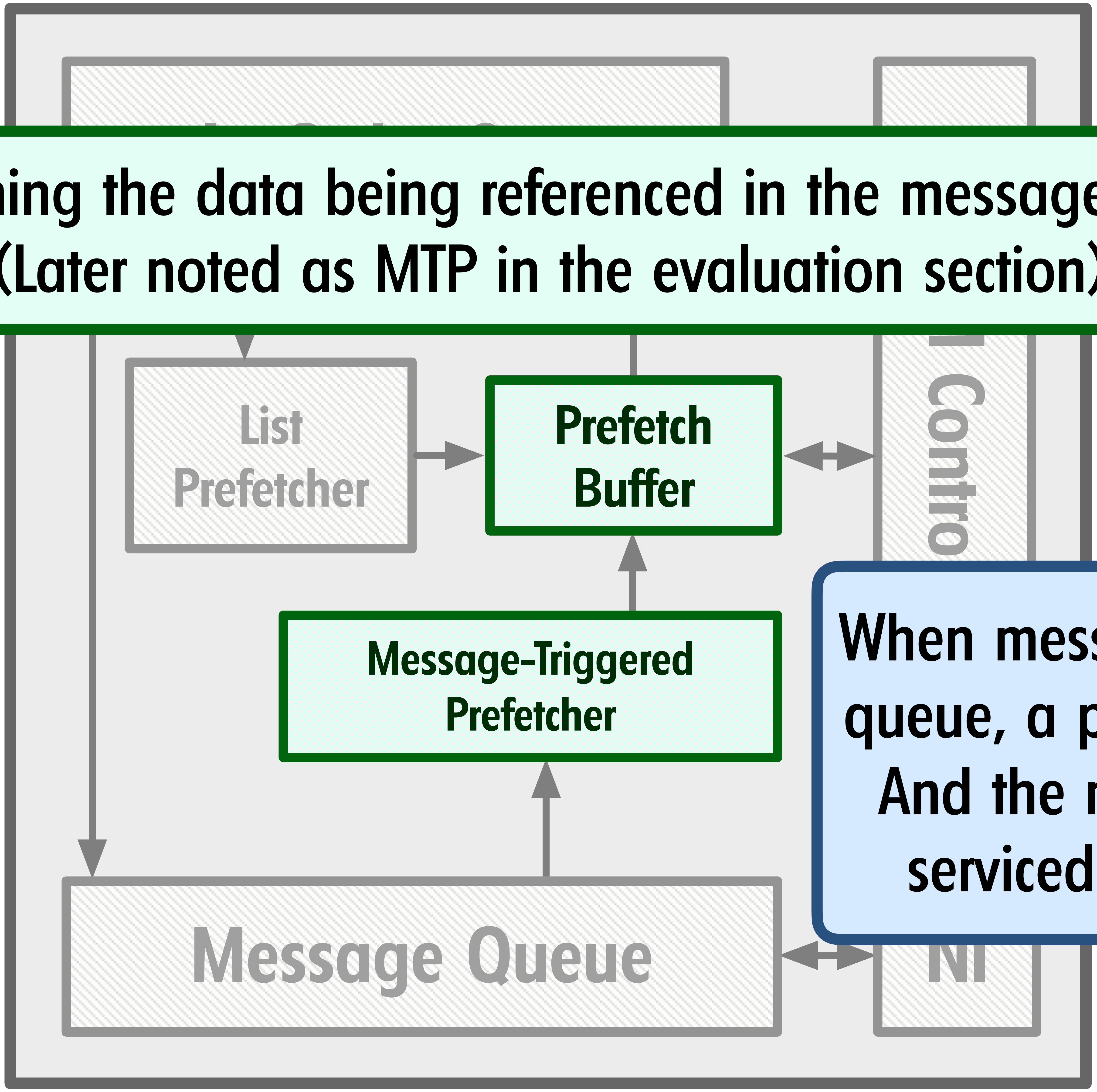
```
for (u: v.successors):
    put(w.id, function() { w.next_rank += weight *
barrier()
```

Non-blocking remote function call, increases latency toleration in the source core and guarantees atomicity

Send function address and **arguments** to the remote core



Prefetching in Tesseract



Prefetching the data being referenced in the message queue
(Later noted as MTP in the evaluation section)

When message enters the message queue, a prefetch request is issued
And the message is ready to be serviced when data is present

Tesseract Core

Novelties of Tesseract

- Usage of PIM (logic layer integration) to **increase the BW** available to the cores
- Message passing employed, to **increase latency tolerance** and guarantee atomicity
- Specially crafted prefetching mechanisms are used to **utilize the abundant BW** available for graph processing

Other Constructs of Tesseract:

1. List Prefetching: Prefetching based on the next elements in the list of traversal, with a constant stride (later noted as LP in the evaluation section)

2. Programming API
3. Blocking remote function calls

Evaluation

DDR3-OoO
HMC-OoO
HMC-MC
Tesseract (No Prefetching)
Tesseract + LP
Tesseract + LP + MTP
Speedup
0
5
10
15
20
25
37.1
43.7
33.9
40.1
AT.WK
AT.IC
AT.LJ
CT.WK
CT.IC
CT.LJ
PR.WK
PR.IC
PR.LJ
SP.WK
SP.IC
SP.LJ
VC.WK
VC.IC
VC.LJ
GM

- DDR3 + OoO cores
- HMC + OoO cores, higher bandwidth
- HMC + more number of simpler, less powerful cores
- Tesseract, logic layer integration of the HMC with Tesseract cores

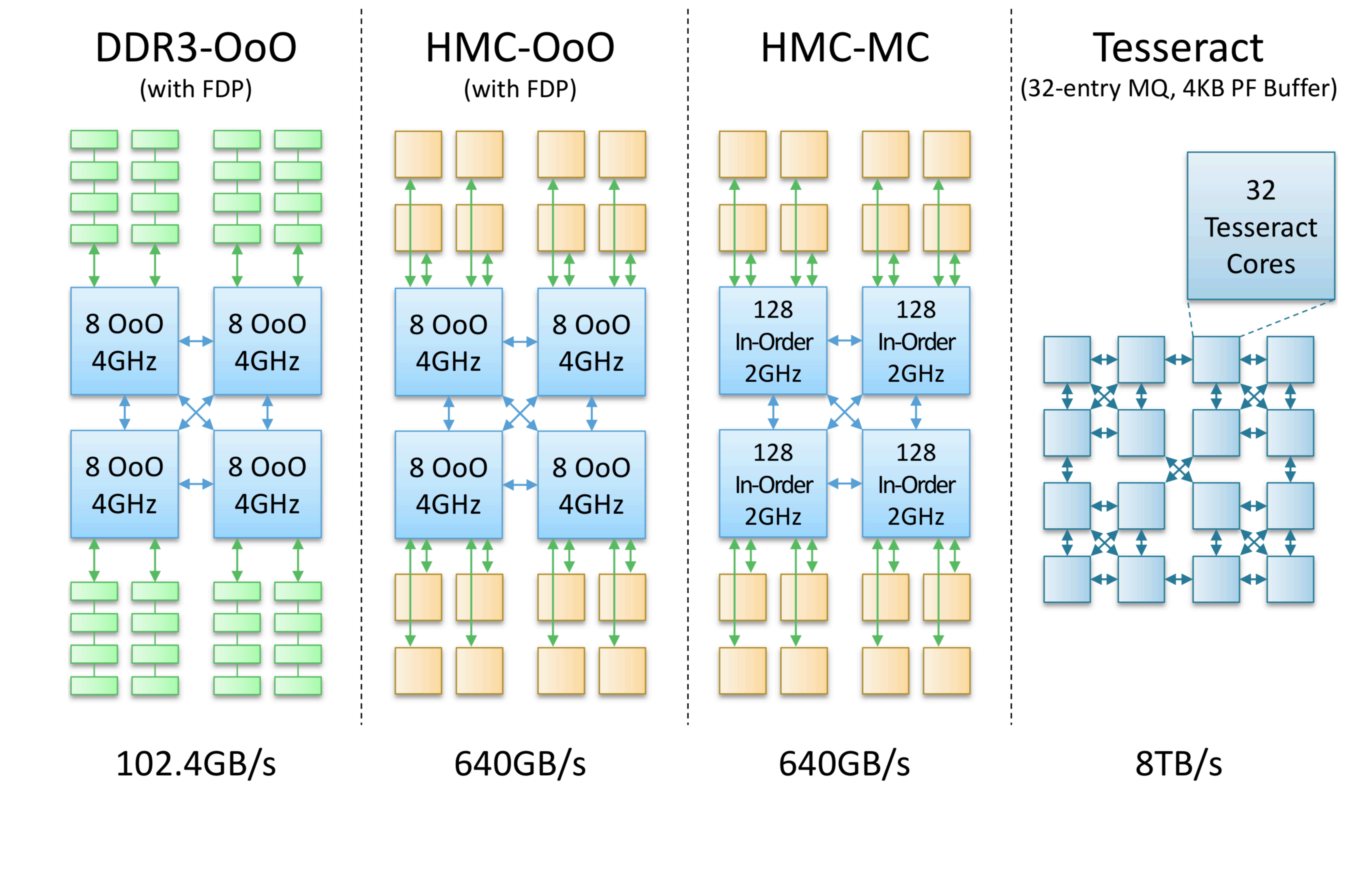
Wor … tems

3 real world graphs:

- ljournal-2008 (social network)
- enwiki-2003 (Wikipedia)
- indochina-0024 (web graph)

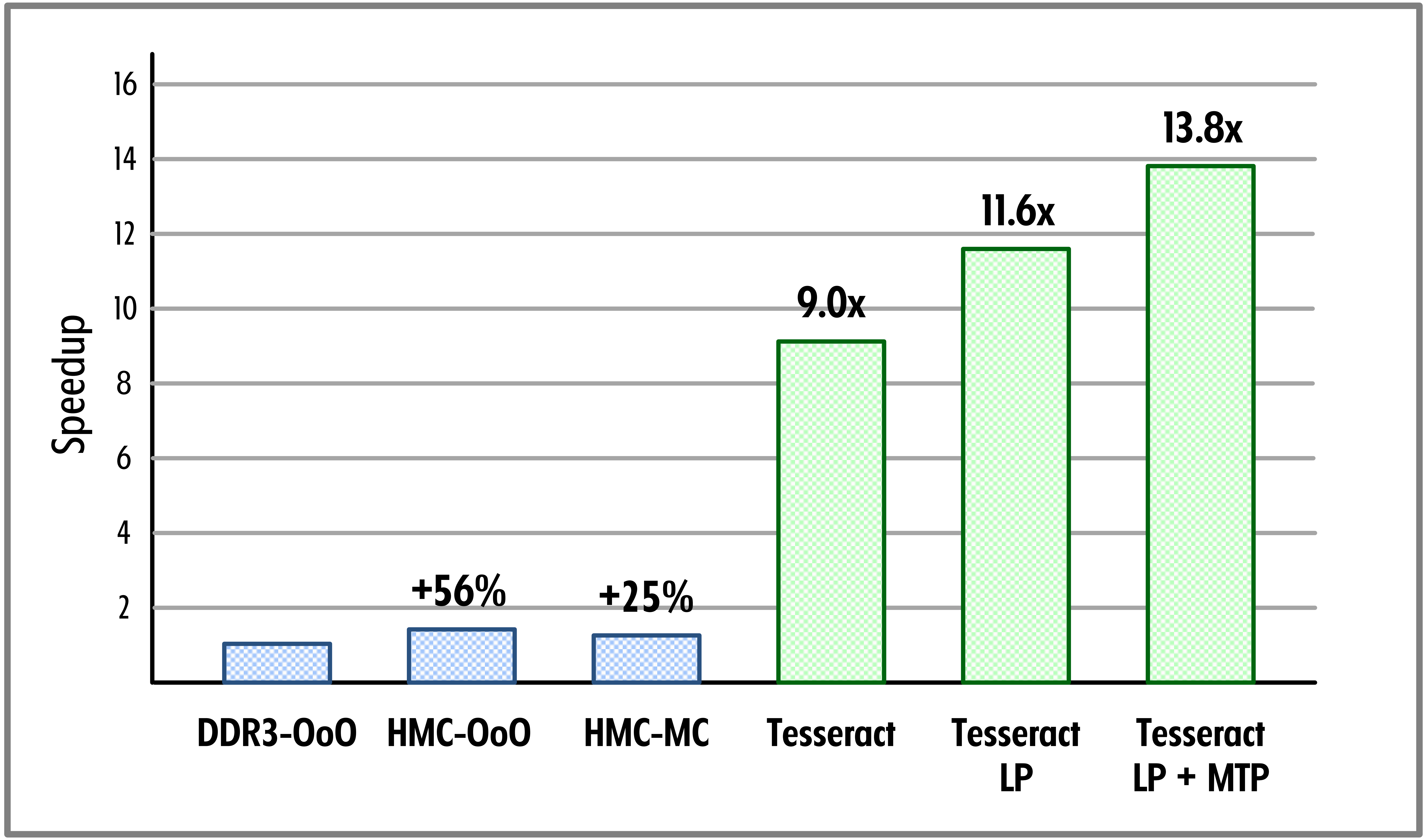
5 graph processing algorithms:

- Average teenage follower
- Conductance
- PageRank
- Single-source shortest path
- Vertex cover



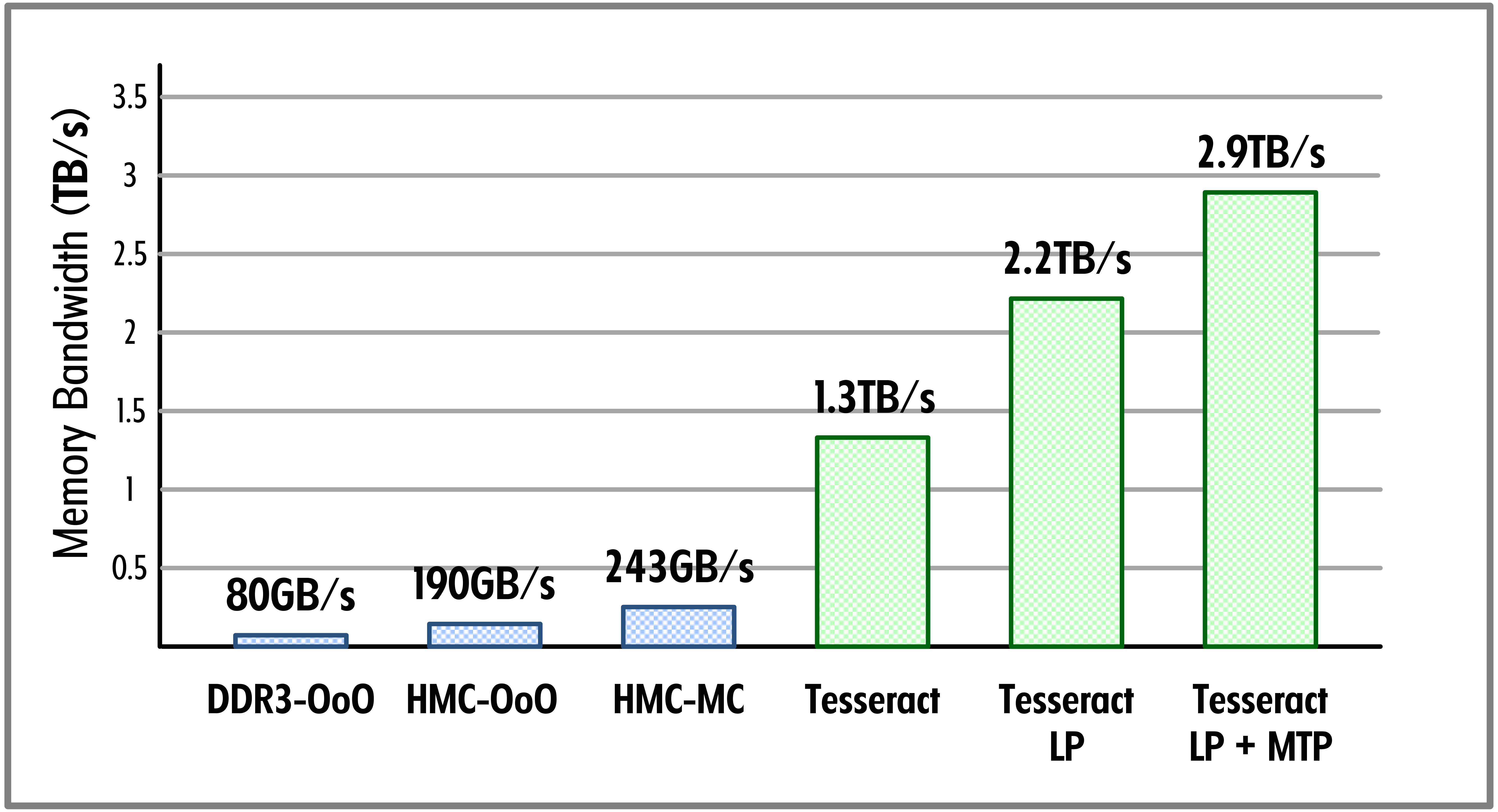
Evaluation Results

Average Performance



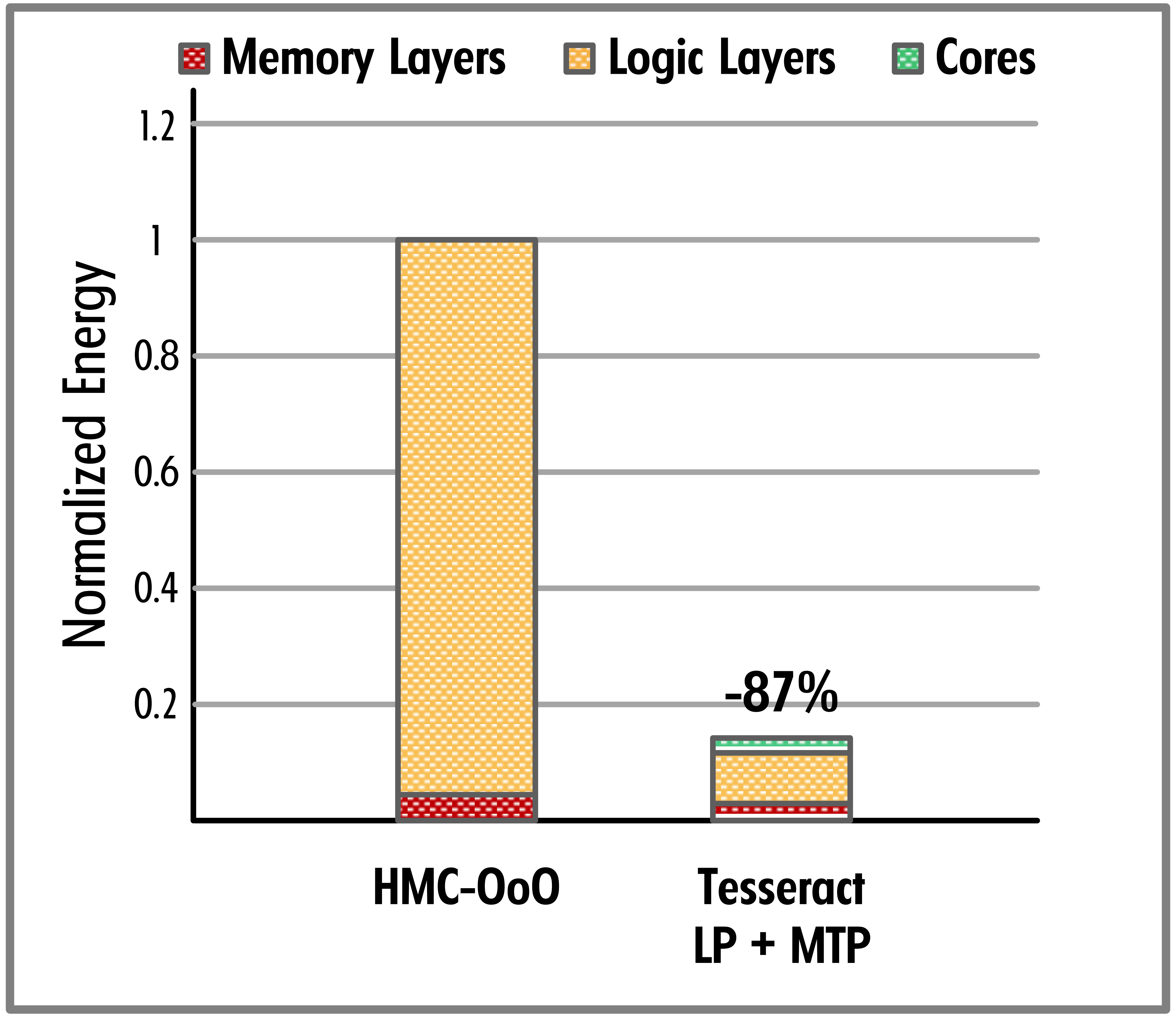
Evaluation Results

Average Bandwidth Utilization



Evaluation Results

Average Memory Energy Consumption



Executive Summary

Problem: Performance of graph processing on conventional systems does not scale in proportion to graph size

Goal: Design an infrastructure with scalable performance for graph processing

Observation: High memory bandwidth can sustain scalability in graph processing

Key Idea: Make use of Processing-In-Memory to provide high bandwidth, and design specially architected cores to utilize that bandwidth

Results: 10x performance improvement and 87% energy reduction


Analysis

Strengths

1. First work to introduce Processing-In-Memory to graph computations

2. Employing specially designed prefetching mechanisms to better utilize BW

3. Non-blocking remote function call is an effective way to increase latency tolerance

4. The paper is written in a way that is easy to follow

Weaknesses

1. Data placement is not taken as a serious concern in this work (GraphP [1], Reduce communication in Tesseract with efficient data placement)
2. The paper has not discussed why it is limited to graph applications
3. Introducing barriers raises the concern of load balancing
4. No comparison against prevalent graph processing platforms like GPUs is included in the paper
5. Adapting common applications to the programming model is not easy

Takeaways

1. Optimizing a narrow set of factors might lead to underutilization of resources

2. If designed effectively, PIM might be a promising approach to provide high bandwidth for large scale data processing

Discussions

1. There is the other construct called Blocking Remote Function Calls

The difference is that in that one you have return values that you want to wait for them to come back to the source core

Can you think of ways to optimize remote blocking function calls?

Discussions

2. How hard will it be to expand Tesseract to other applications?

Discussions

3. How bad will Tesseract suffer from unbalanced workloads?

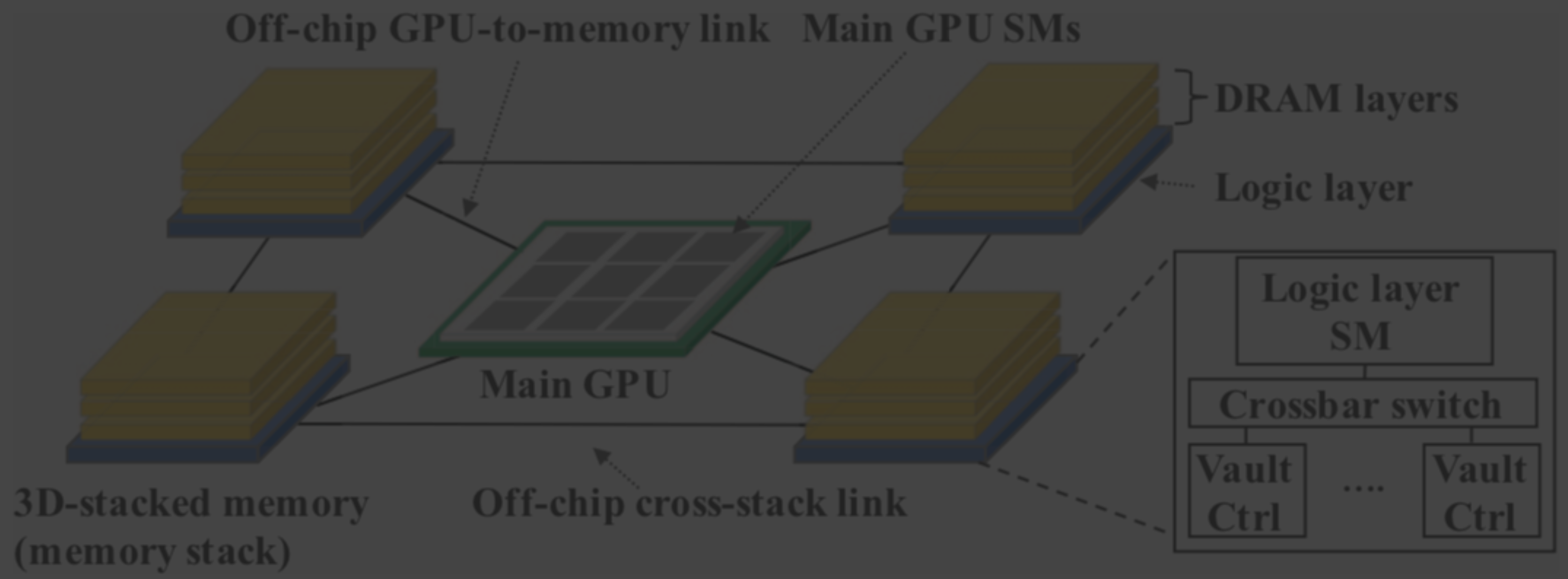
Discussions

4. What if we switch Tesseract cores with GPU Streaming Multiprocessors?

TOM[2]: Transparent Offloading and Mapping

1. What to
2. How to m … he most:
Subsequent … together

30% average performance gain over a baseline with a GPU without offloading



Discussions

4. What if we switch Tesseract cores with GPU Streaming Multiprocessors?

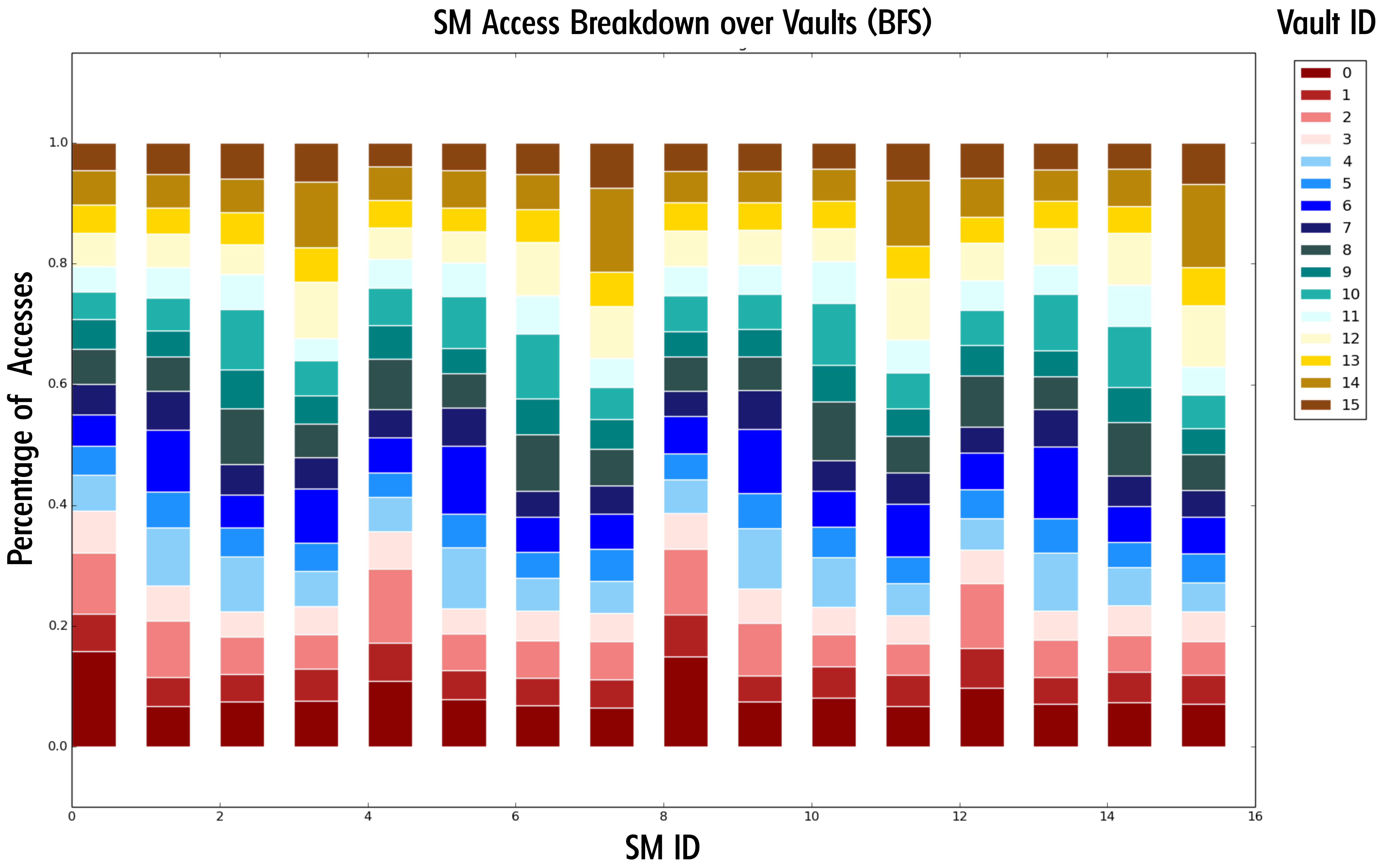
But still, TOM does not employ specially designed mechanisms to mitigate communication between vaults and we will have this problem.

New question: if we have a PIM cube which has GPU cores in its logic layer, how can we reduce the data movement?

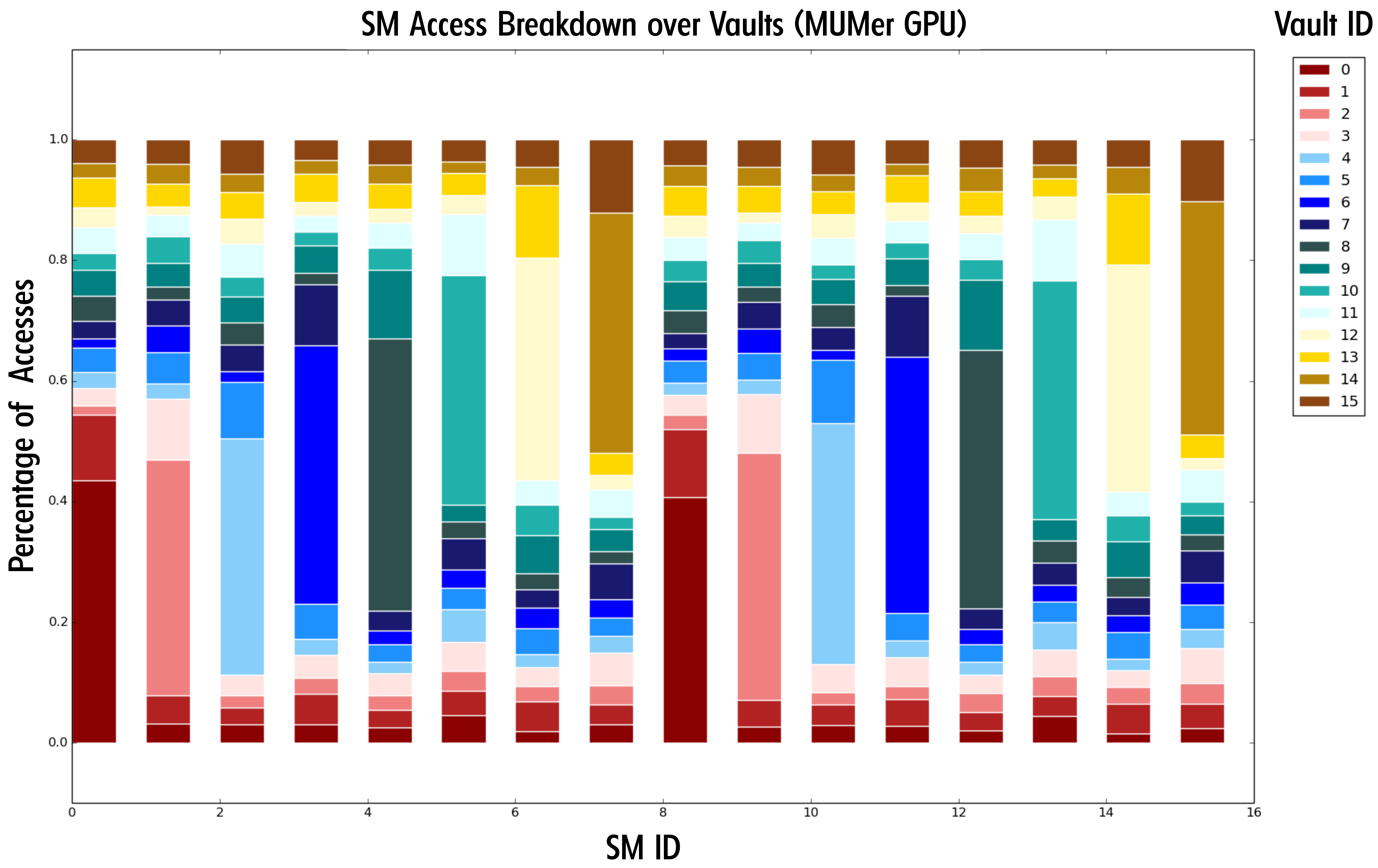
Discussions

1. Remapping?
2. CTA Migration?

CTA is the set of threads running on a GPU SM at a given time



Discussions



Discussions

5. What about data movement between cubes?

GraphP[1]: Reduce communication between the cubes in Tesseract with efficient data placement

3 key techniques:

1. "Source-cut" Partitioning: an algorithm to ensure a vertex and all its incoming edges are in the same cube
2. "Two-phase Vertex Program": a programming model designed for the "source-cut" partitioning
3. "Hierarchical Communication and Overlapping"

Discussions

6. Other mechanisms for the same problem:

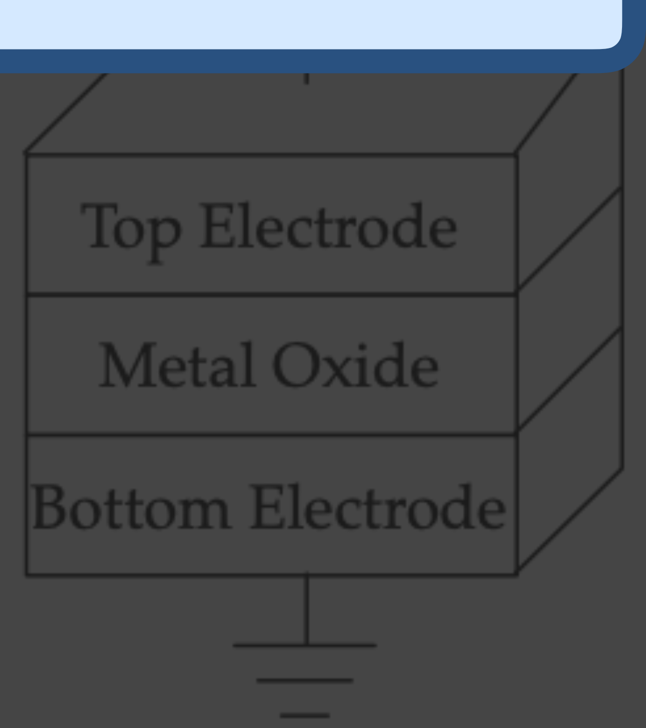
GraphR[3]: Accelerating Graph Processing Using ReRAM

Using c

comput

Results: Up to 4.12x speedup and 10.96% energy saving over Tesseract

With ReRAMs you can do analog computation



References

References

[1] M. Zhang et al., "GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition," 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, 2018, pp. 544-557.

[2] K. Hsieh et al., "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016, pp. 204-216.

[3] Song, Linghao & Zhuo, Youwei & Qian, Xuehai & Li, Hai & Chen, Yiran. (2017). GraphR: Accelerating Graph Processing Using ReRAM. arXiv'17

A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing

Roknoddin Azizibarzoki **Seminar on Computer Architecture**

Junwhan Ahn, Sungpack Hong*
Sungjoo Yoo, Onur Mutlu+
Kiyoung Choi

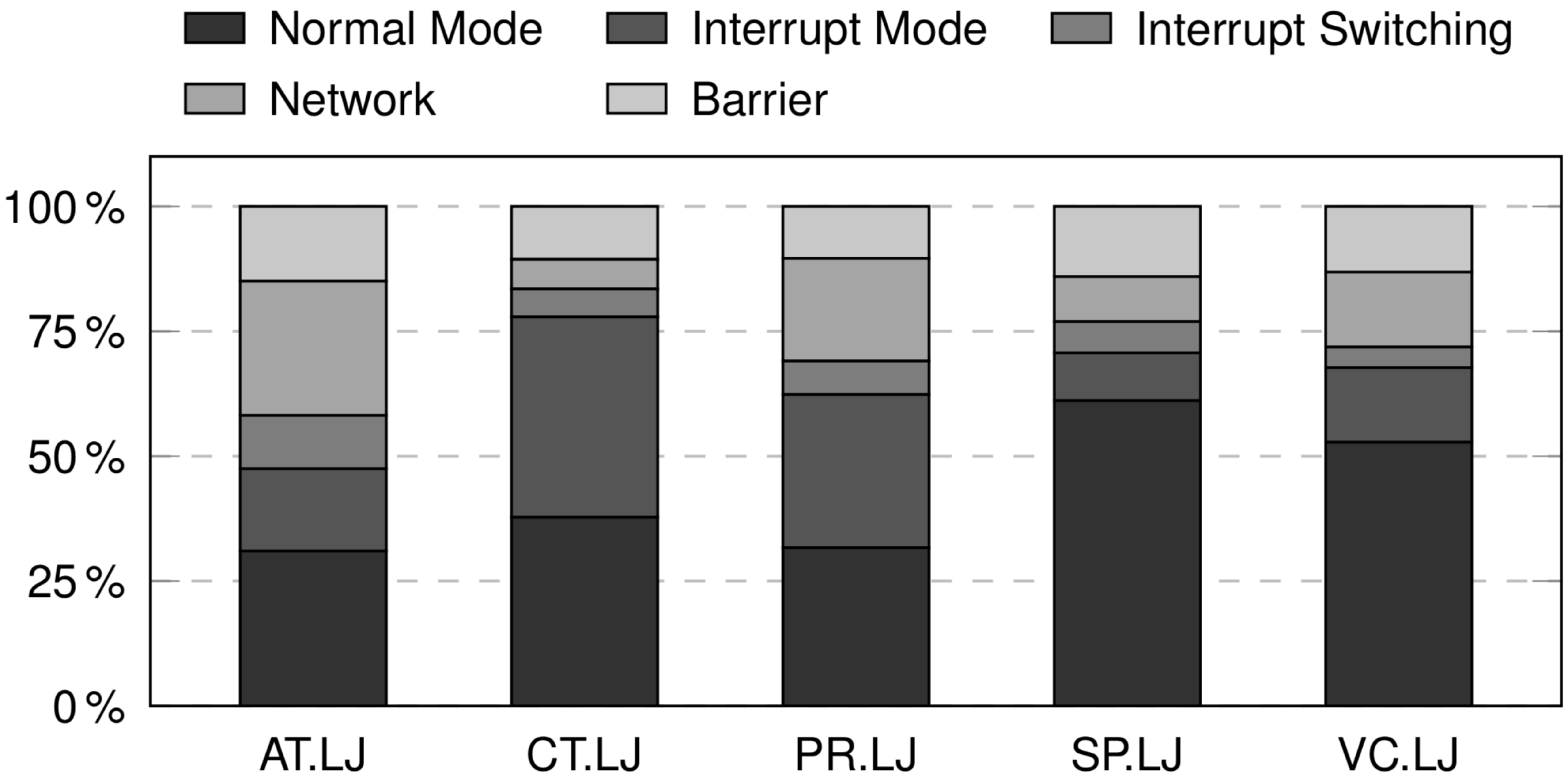
Seoul National University *Oracle Labs +Carnegie Mellon University

International **S**ymposium on **C**omputer **A**rchitecture **2015**

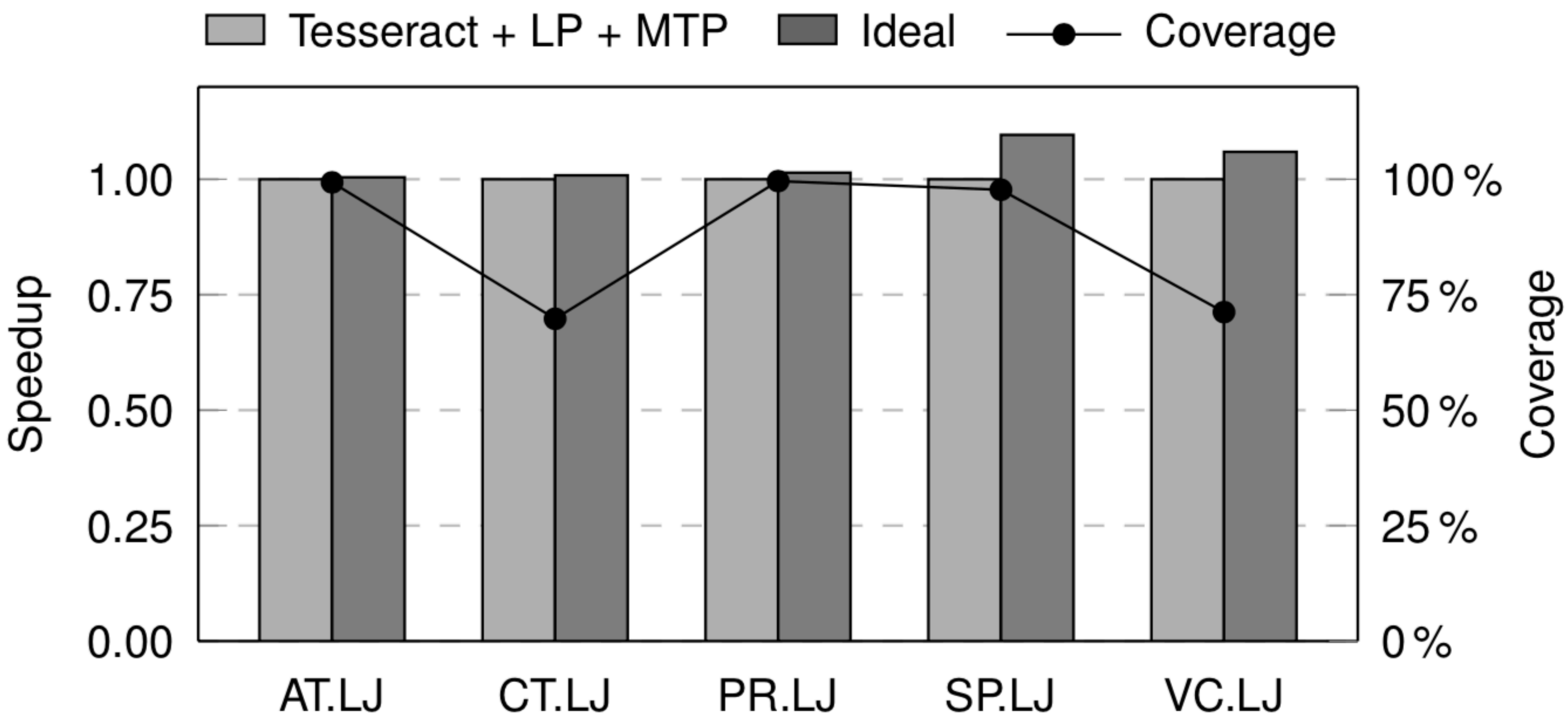
ETH zürich

Backup Slides

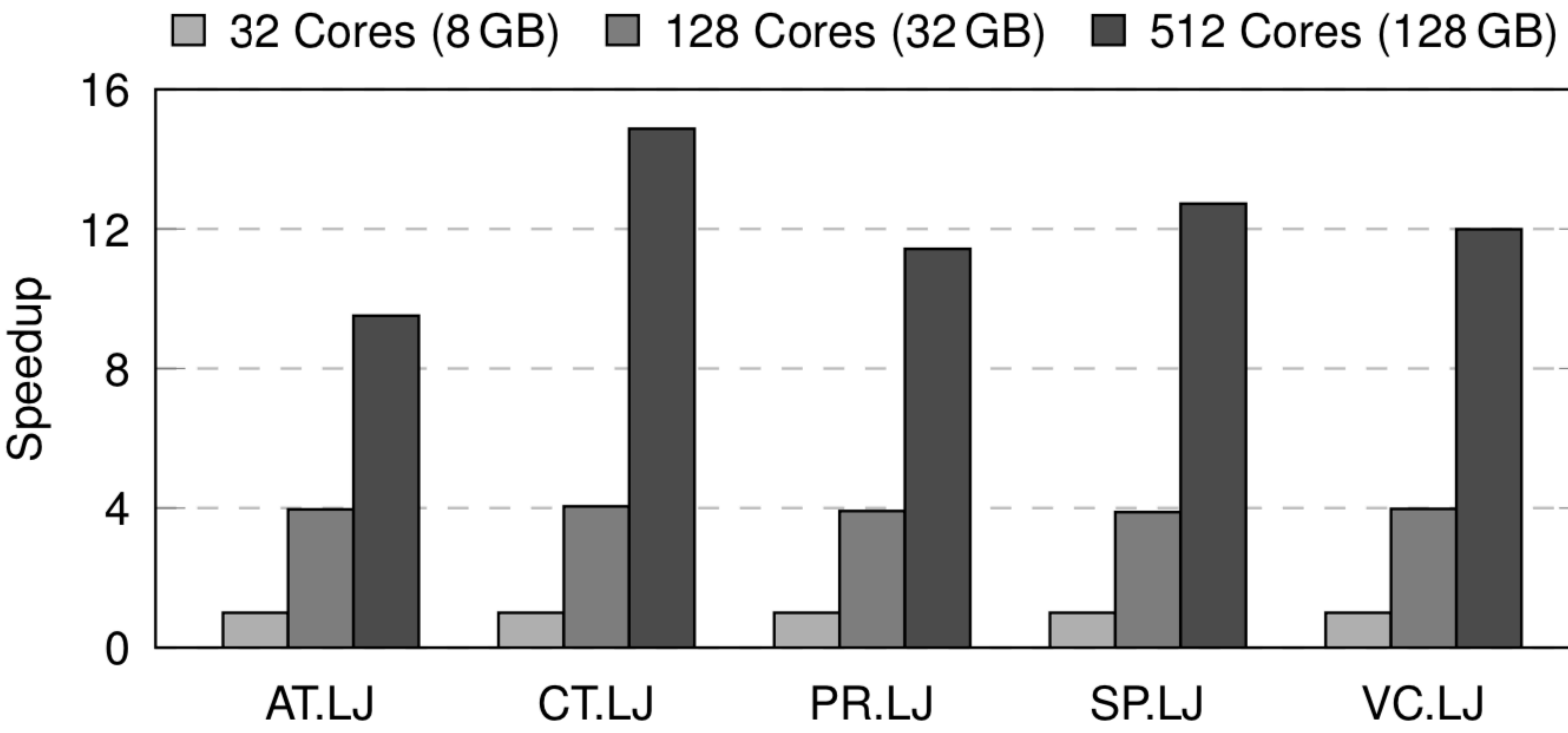
Backup Slides



Backup Slides



Backup Slides



Backup Slides

```
get(id, A func, A arg, S arg_size, A ret, S ret_size)
put(id, A func, A arg, S arg_size, A prefetch_addr)
disable_interrupt(), enable_interrupt()
copy(id, A local, A remote, S size)
list_begin(A address, S size, S stride)
list_end(A address, S size, S stride)
barrier()
```