Seminar on Computer Architecture (Fall 2018)

Tesseract

A Scalable Processing-in-Memory Accelerator
for Parallel Graph Processing (ISCA'15)

presented by Mauro Bringolf

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

Seoul National University, § Oracle Labs, *Carnegie Mellon University

Background and Problem

- Big-data analytics requires processing of ever-growing, large graphs
- Conventional architectures not well suited for graph processing

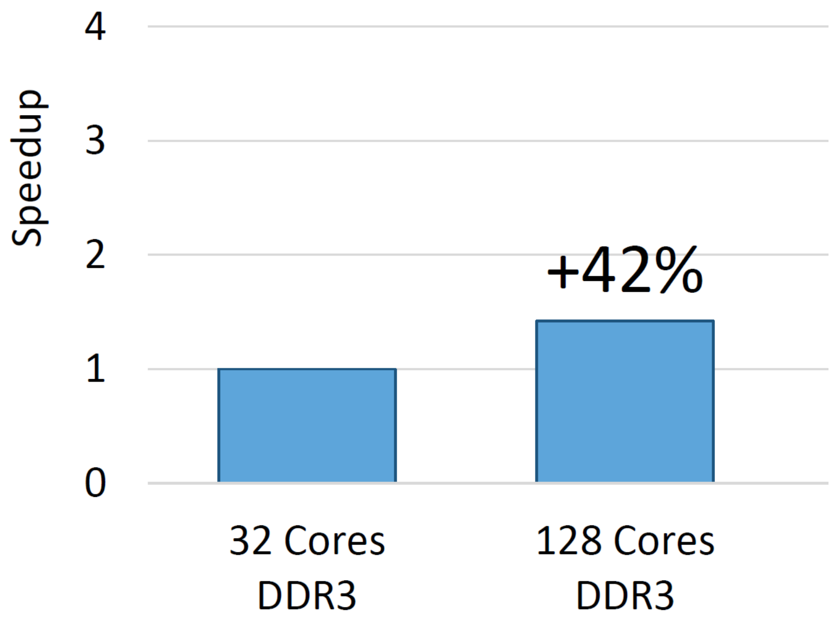
2+ billion users

300+ million users

45+ million pages

Graph Processing Characteristics

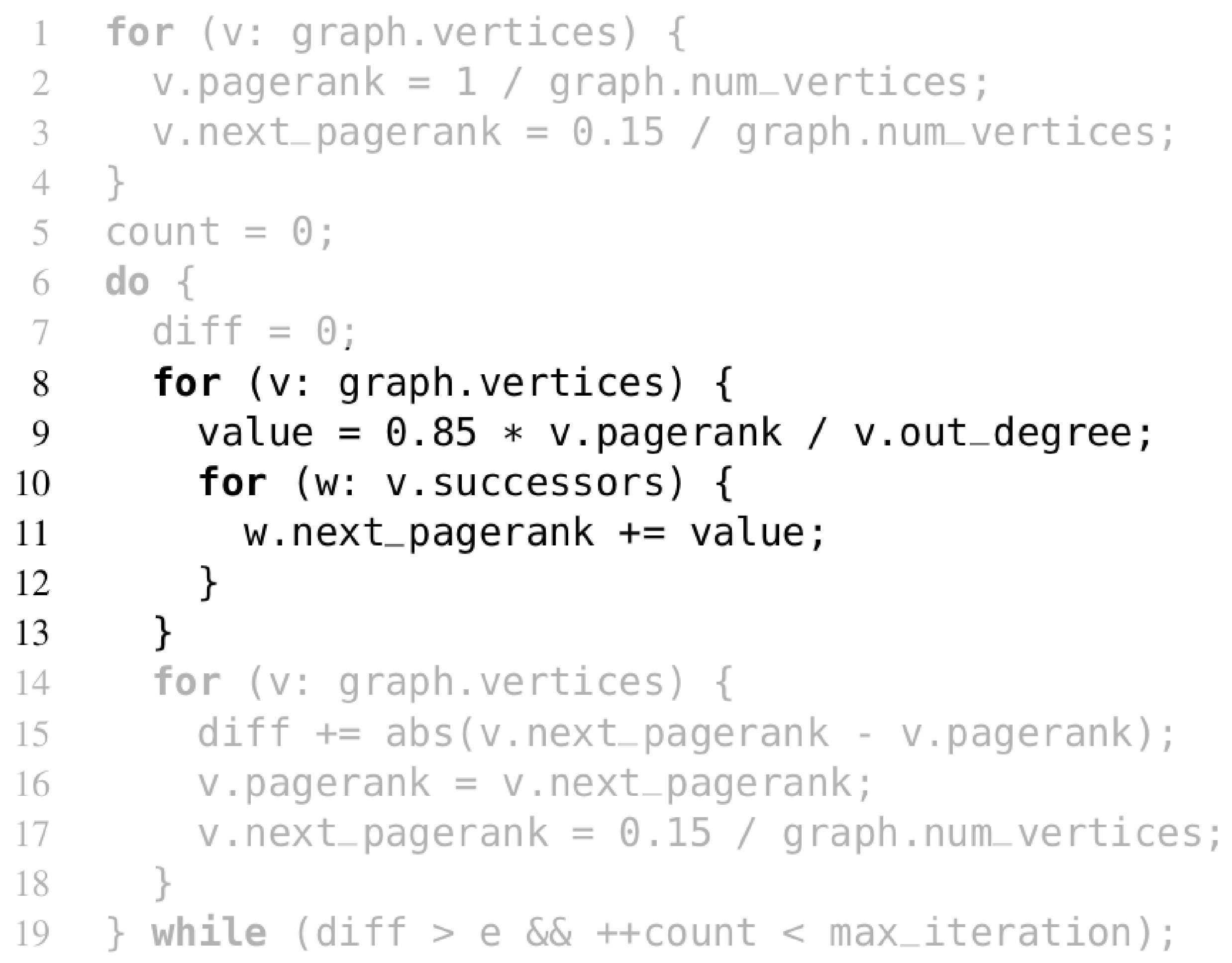
- Frequent random memory accesses during neighbor traversals
- Typically small amount of computation per vertex



Summary

- Problem: Memory bandwidth is the bottleneck for graph processing on conventional architectures
- Goal: Ideally, performance should increase proportionally to size of stored graphs in a system
- Key Mechanism: A new Processing-in-Memory architecture which increases available memory bandwidth by 10x and a programming model to use it efficiently
- Results: In evaluation, Tesseract achieves 10x performance and 87% energy reduction over conventional architectures

Example - PageRank



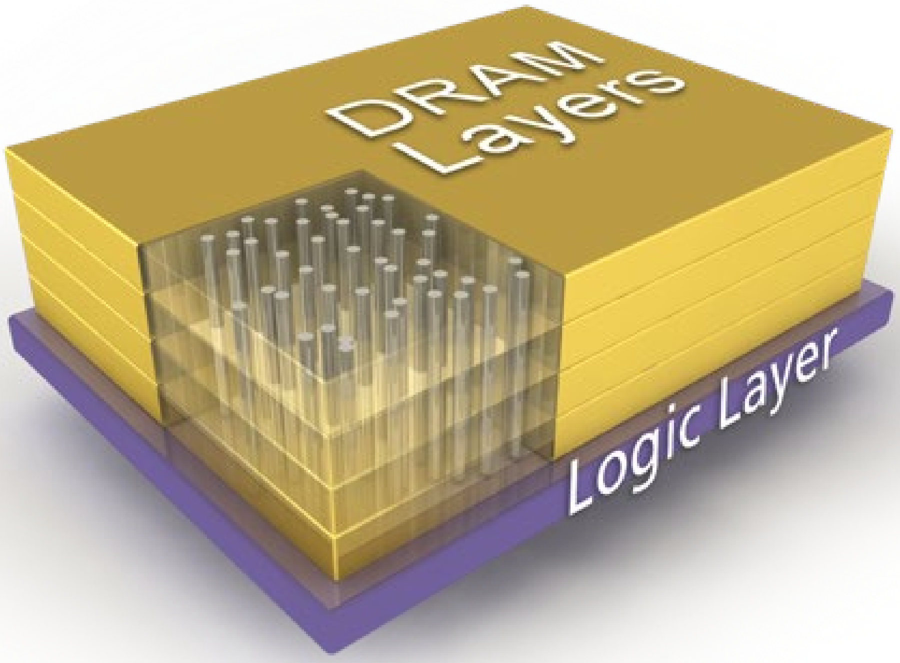
```
for (v: graph.vertices) {
  v.pagerank = 1 / graph.num_vertices;
  v.next_pagerank = 0.15 / graph.num_vertices;
}
count = 0;
do {
  diff = 0;
  for (v: graph.vertices) {
    value = 0.85 * v.pagerank / v.out_degree;
    for (w: v.successors) {
      w.next_pagerank += value;
    }
  }
  for (v: graph.vertices) {
    diff += abs(v.next_pagerank - v.pagerank);
    v.pagerank = v.next_pagerank;
    v.next_pagerank = 0.15 / graph.num_vertices;
  }
} while (diff > e && ++count < max_iteration);
```

Figure 1: Pseudocode of PageRank computation.

Source: [2] J. Ahn et al., "A Scalable Processing-in-Memory…"

Key Ideas – Processing-in-Memory

- Apply the idea of Processing-in-Memory (PIM) using a Hybrid Memory Cube (HMC) which allows efficient stacking of memory and logic layers
- Exploit internal memory bandwidth of HMC



Architecture

- A Tesseract core is an HMC enhanced with one processor per vault
- Tesseract is memory-mapped to non-cacheable memory of the host processor
- Each processor only has access to its local vault but can communicate with other vaults via messages

HMC Internal Memory Bandwidth

- Each vault is connected via a 64bit wide interface sending at 2GB/s to the crossbar network
- One HMC consists of 32 vaults which yields an internal **available** bandwidth of 512GB/s versus 320 GB/s external

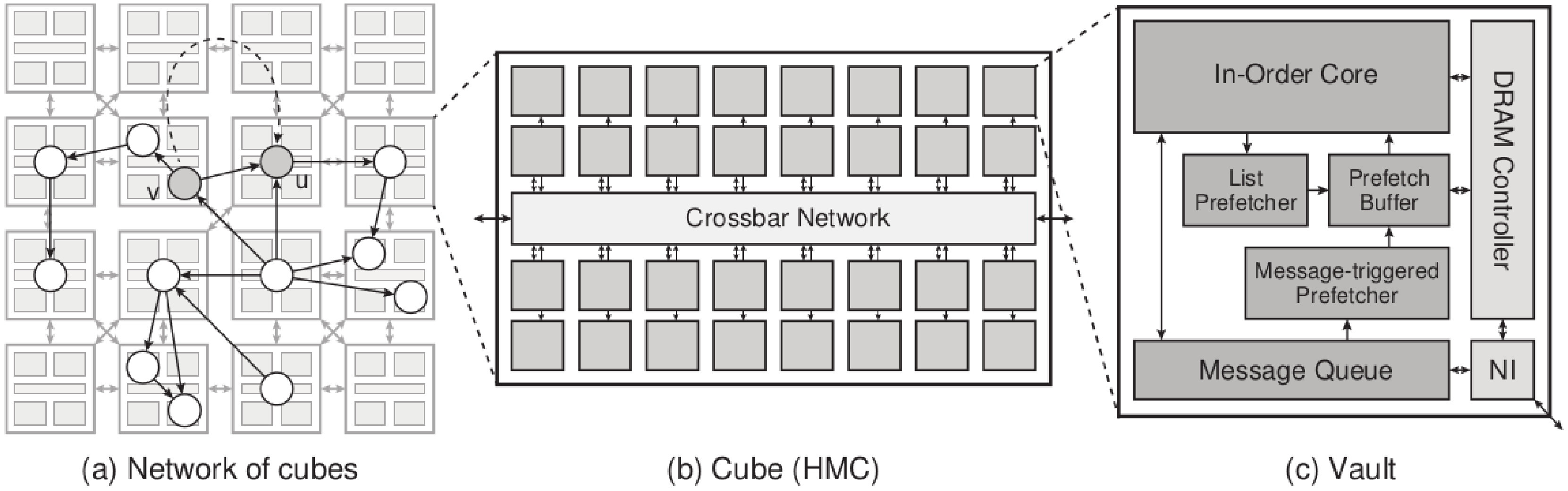


Figure 3: Tesseract architecture (the figure is not to scale).

Key Ideas – Programming Model

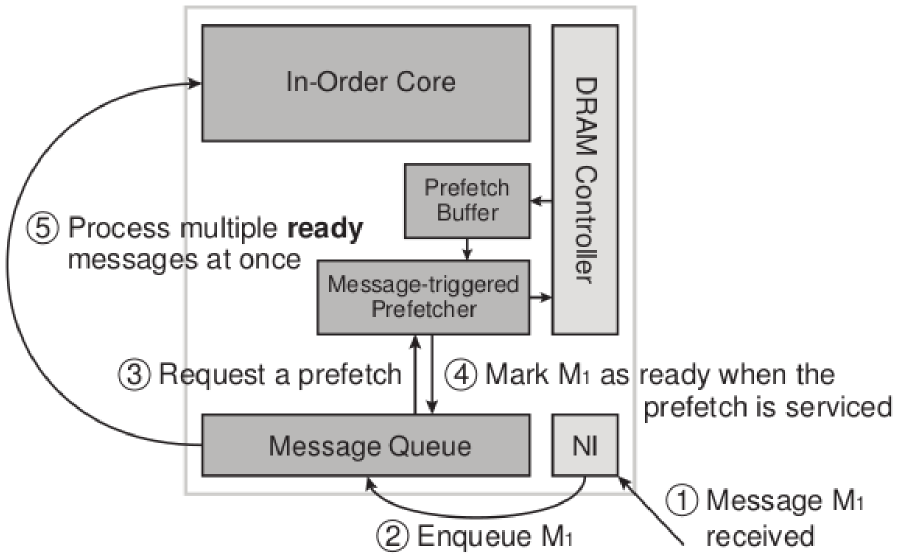
- Apply vertex-focused programming model to PIM
- Use a message passing mechanism to exploit data parallelism

Programming Interface

- Blocking vs. non-blocking remote function call
- Message queue, interrupts for batch processing of messages

Prefetch Mechanisms

- Stride prefetcher for traversal of vertex list or list of edges per vertex
- Message-triggered prefetcher to hide access latency



In-Order Core
DRAM Controller
Prefetch Buffer
⑤ Process multiple **ready** messages at once
Message-triggered Prefetcher
③ Request a prefetch
④ Mark M1 as ready when the prefetch is serviced
Message Queue
NI
① Message M1 received
② Enqueue M1

Figure 4: Message-triggered prefetching mechanism.

Example - PageRank

```
1   for (v: graph.vertices) {
2     v.pagerank = 1 / graph.num_vertices;
3     v.next_pagerank = 0.15 / graph.num_vertices;
4   }
5   count = 0;
6   do {
7     diff = 0;
8     for (v: graph.vertices) {
9       value = 0.85 * v.pagerank / v.out_degree;
10      for (w: v.successors) {
11        w.next_pagerank += value;
12      }
13    }
14    for (v: graph.vertices) {
15      diff += abs(v.next_pagerank - v.pagerank);
16      v.pagerank = v.next_pagerank;
17      v.next_pagerank = 0.15 / graph.num_vertices;
18    }
19  } while (diff > e && ++count < max_iteration);
```

Figure 1: Pseudocode of PageRank computation.

```
...
count = 0;
do {
  ...
  list_for (v: graph.vertices) {
    value = 0.85 * v.pagerank / v.out_degree;
    list_for (w: v.successors) {
      arg = (w, value);
      put(w.id, function(w, value) {
        w.next_pagerank += value;
      }, &arg, sizeof(arg), &w.next_pagerank);
    }
  }
  barrier();
  ...
} while (diff > e && ++count < max_iteration);
```

Figure 5: PageRank computation in Tesseract (corresponding to lines 8–13 in Figure 1).

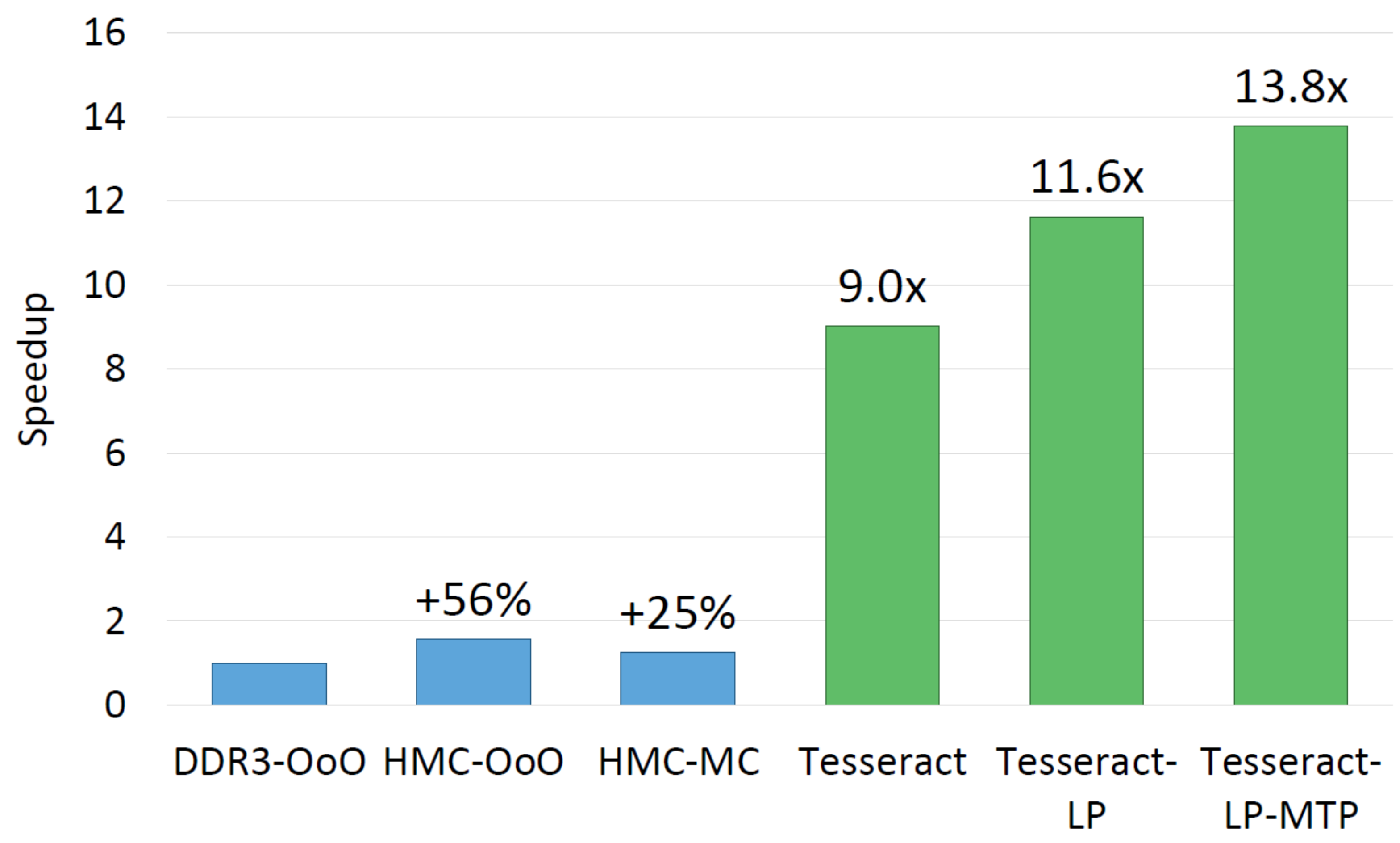
Evaluation

- Design is evaluated in simulation against conventional DDR3-based and HMC-based architectures
- 16 HMC's are used yielding 128 GB main memory
- DDR3-based: 102.4 GB/s
- HMC-based: 640 GB/s
- Tesseract: 8 TB/s

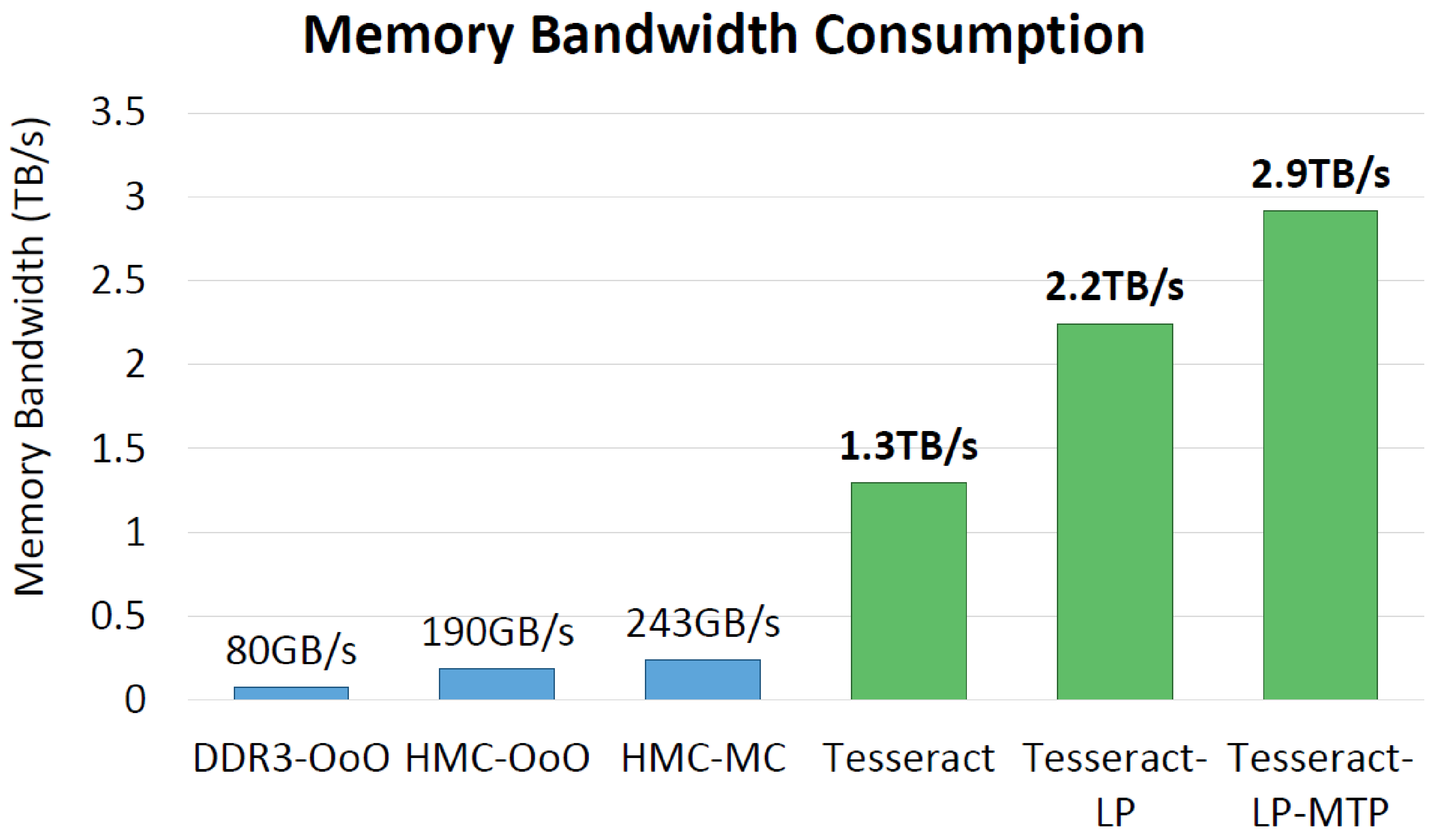
Workloads

- Five standard graph algorithms including PageRank
- Data sets are obtained from applications in the internet context including Wikipedia
- Input graphs contain a couple of million nodes, 100-200 millions of edges and are 3-5 GB

Key Results - Speedup



Key Results - Speedup



Key Results - Scalability

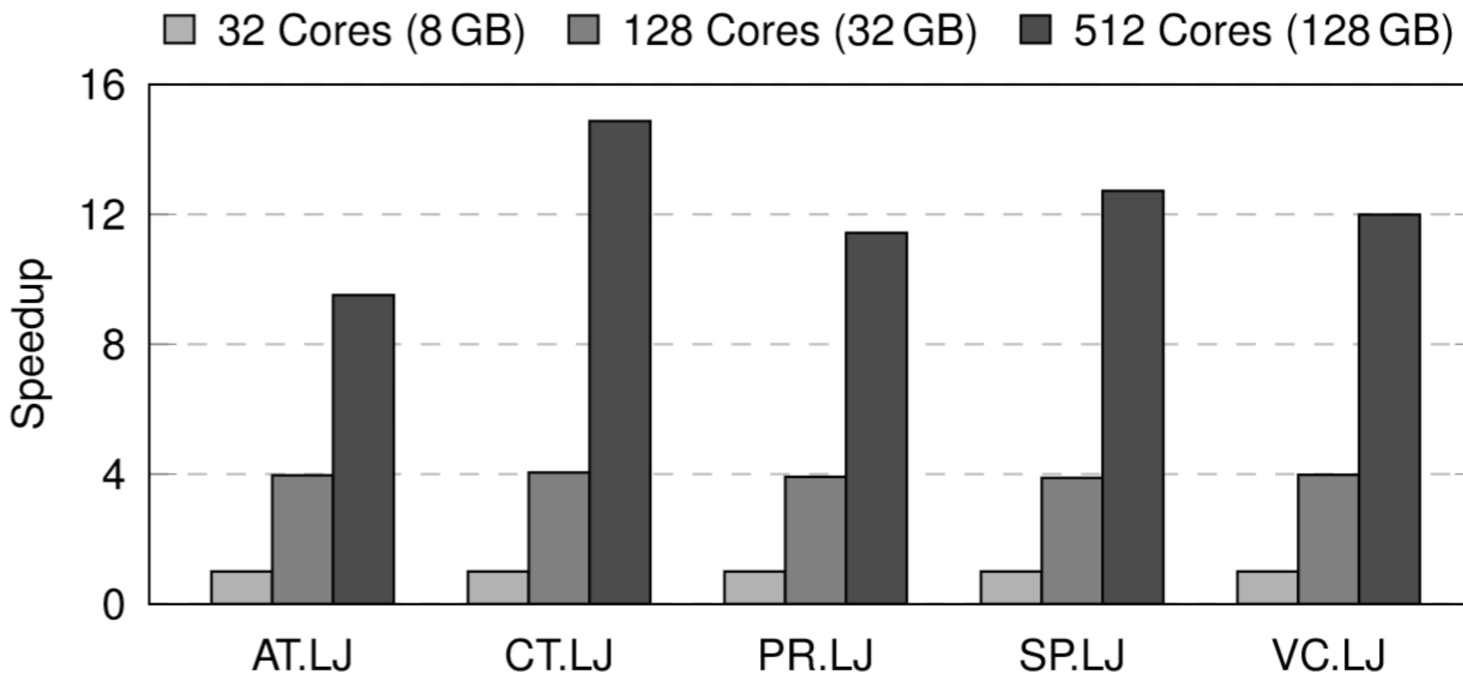


Figure 11: Performance scalability of Tesseract.

Summary

- Problem: Memory bandwidth is the bottleneck for graph processing on conventional architectures
- Goal: Ideally, performance should increase proportionally to size of stored graphs in a system
- Key Mechanism: A new Processing-in-Memory architecture which increases available memory bandwidth by 10x and a programming model to use it efficiently
- Results: In evaluation, Tesseract achieves 10x performance and 87% energy reduction over conventional architectures

Strengths

- Combines two strong ideas such that they benefit from each other: PIM and parallel programming model
- Performance analysis tries to isolate the different parts of the design
- Message-triggered prefetching is an intuitive idea with great performance benefits
- Design is not overly specific to graph workloads

Weaknesses

- Re-implemention of algorithms presents a tradeoff
- Global synchronization barrier might be problematic for imbalanced workloads across vaults
- The importance of graph distribution seems understated in the paper to me

Effects of Graph Distribution

- From the GraphP paper (1.7x speedup):

"In TESSERACT, data organization aspect is not treated as a primary concern and is subsequently determined by the presumed programming model"

Source: [3] M.Zhang, Y.Zhuo et al, "GraphP: Reducing Communication…"

Takeaways

- Processing-in-Memory can be a viable solution to the memory bottleneck
- A paradigm shift from the current conventional architectures can give great improvements by designing radically new systems
- Proven ideas from software can manifest themselves as new hardware designs

Seminar on Computer Architecture (Fall 2018)

Tesseract

A Scalable Processing-in-Memory Accelerator
for Parallel Graph Processing (ISCA'15)

presented by Mauro Bringolf

Junwhan Ahn, Sungpack Hong [§], Onur Mutlu*, Sungjoo Yoo, Kiyoung Choi

Seoul National University, [§] Oracle Labs, *Carnegie Mellon University

Discussion

- Is there a better way to handle synchronization across one HMC between vaults?
- Is this design specific to graph workloads? Can you think of scenarios where it performs poorly?
- Do you think automatic translation of algorithms is difficult?

References

(1) O. Mutlu, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing” (Slides)

(2) J. Ahn et al, “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing” (ISCA’15)

(3) M. Zhang, Y. Zhuo et al, “GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition”

get

- Retrieve data from a remote core
- Blocking remote function call

set, copy

- Store data on a remote core
- Non-blocking remote function call
- Guaranteed to be finished before next synchronization barrier

disable_interput, enable_interput

- Stop processing messages and only do local work
- Can be used to avoid data races

barrier

- A synchronization barrier across all Tesseract cores
- Can be used to avoid data races

list_begin, list_end

- Configure the prefetcher before doing a list traversal