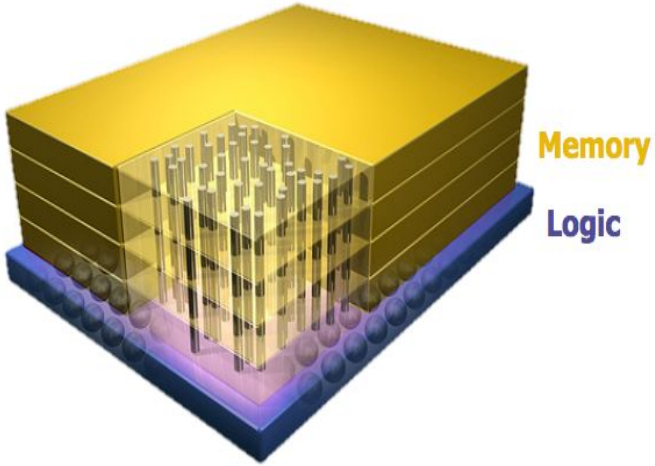
A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing

Junwhan Ahn Sungpack Hong[§] Sungjoo Yoo Onur Mutlu[†] Kiyoung Choi
junwhan@snu.ac.kr, sungpack.hong@oracle.com, sungjoo.yoo@gmail.com, onur@cmu.edu, kchoi@snu.ac.kr
Seoul National University [§]Oracle Labs [†]Carnegie Mellon University

Reviewed by: Hanjing Gao

Why PIM



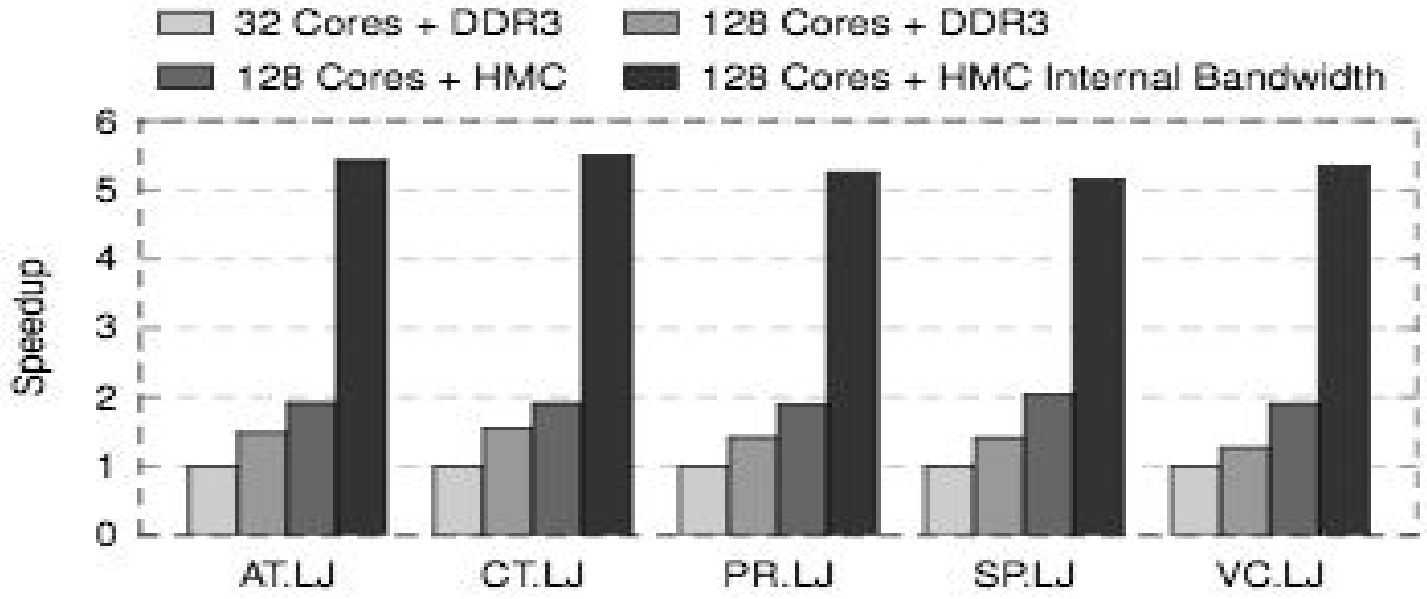
- Not a new idea
- Application demand and requirement
  - Large scale graph processing Everywhere
  - Moving from second-tier storage to memory
- Inefficiency in conventional systems
  - Beefy cores and large cache
  - Specialized on-chip accelerators are not enough
  - Memory bandwidth
- 3D Integration technology
- Think big -- “M” can be anything!
- Think critically -- limitations of the design and what’s the principle that drives the new design

Challenges

- Large amount of random memory accesses
  - Neighbor traversal
  - Poor locality
- Little computation

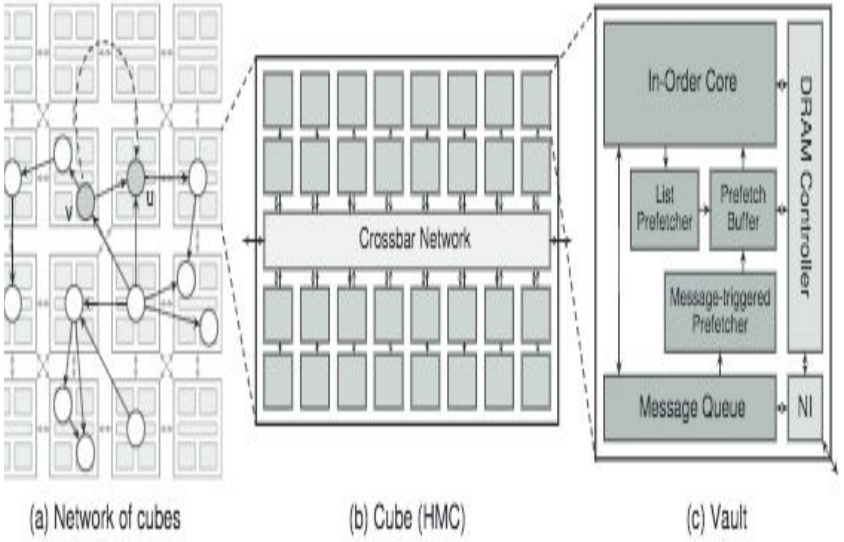
```
 8     for (v: graph.vertices) {
 9       value = 0.85 * v.pagerank / v.out_degree;
10       for (w: v.successors) {
11         w.next_pagerank += value;
```

Reality today

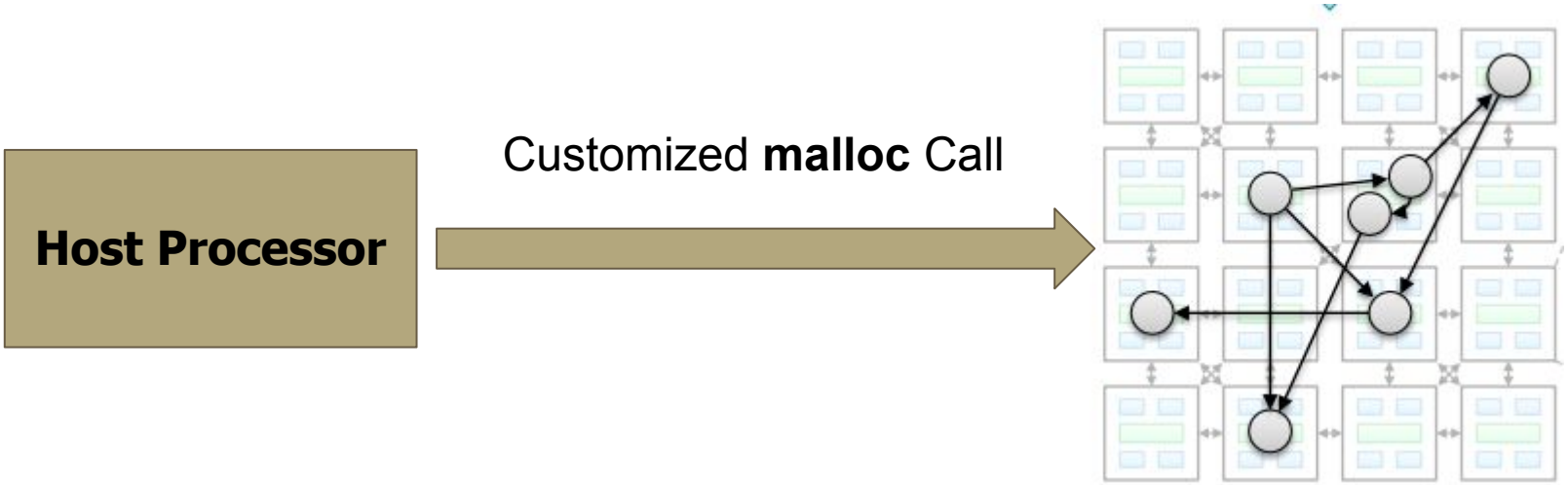


Tesseract Architecture

- 32 vaults
  - 32 single-issue in-order core
  - Dedicated memory controller
- Host processor
  - Memory mapped
    - Non-cacheabale
    - Physically Addressed
- Host to distribute the graphs
  - Customized malloc call
  - Spoiler: Balanced graph distribution
- Message passing
- Prefetching
- Programming interface

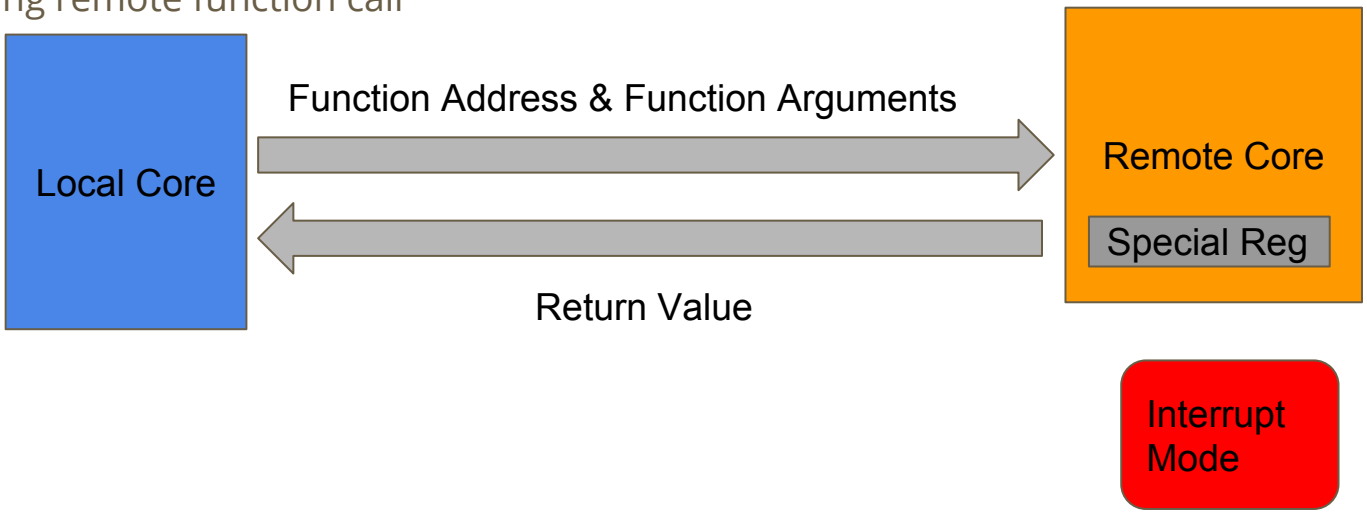


Interface



Message Passing

- Moves computation to where data resides
- Two types of function calls
  - Blocking remote function call



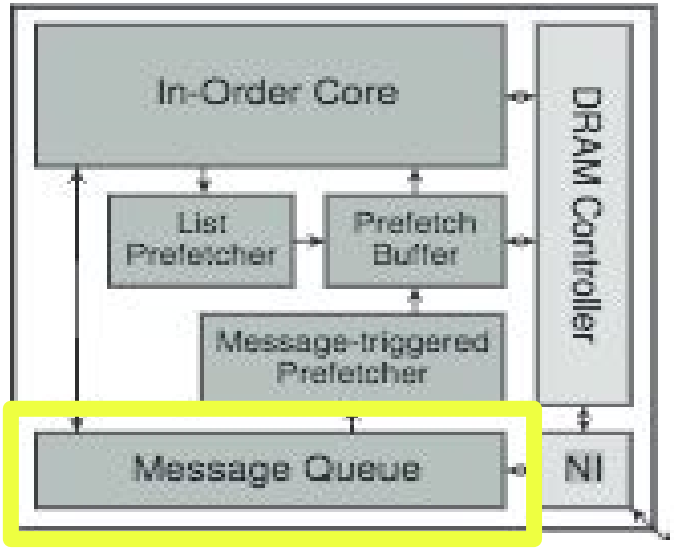
```
1  for (v: graph.vertices) {
2    v.pagerank = 1 / graph.num_vertices;
3    v.next_pagerank = 0.15 / graph.num_vertices;
4  }
5  count = 0;
6  do {
7    diff = 0;
8    for (v: graph.vertices) {
9      value = 0.85 * v.pagerank / v.out_degree;
10     for (w: v.successors) {
11       w.next_pagerank += 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);
```

- Global state checking
- Block local core
- Overhead at the remote call

Message Passing: Non-blocking function call

- Non-blocking function calls
  - Useful for updating remote data
  - No return values
  - Synchronization barrier
  - Batch processing
    - Avoid CS overhead

```
8    for (v: graph.vertices) {
9      value = 0.85 * v.pagerank / v.out_degree;
10     for (w: v.successors) {
11       w.next_pagerank += value;
```



Prefetching

- List Prefetching
  - Sequential accesses
  - Prefetch cache blocks
  - Accept user information
- Message-triggered Prefetching
  - Random access patterns
    - Edges -> random vertices
  - Mostly on remote accesses
  - During Non-blocking remote call

```
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);
```

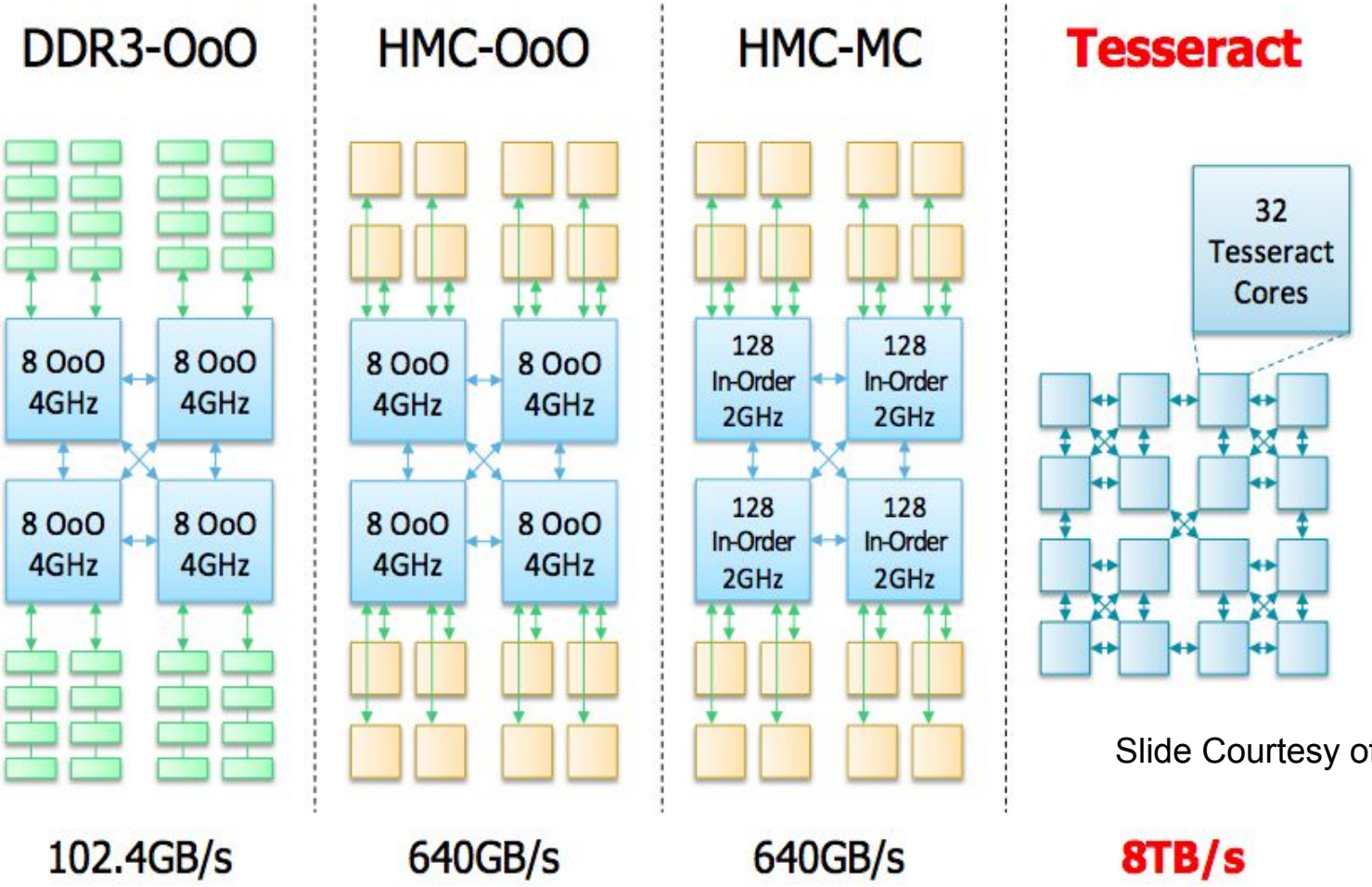
Programming interface & Application Mapping

- Get and put for blocking and non-blocking remote function calls
-
```
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)
```

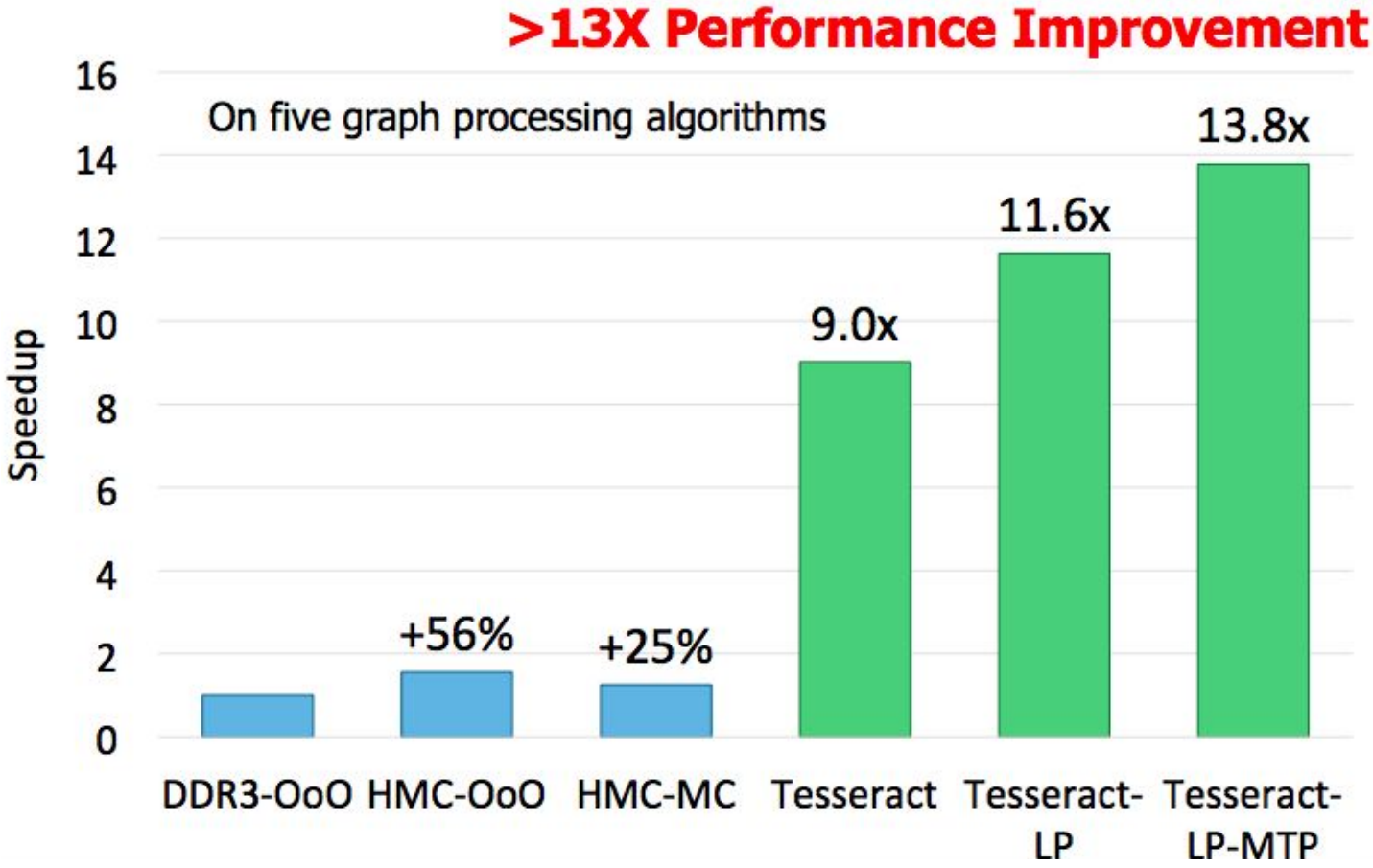
```
1  ...
2  count = 0;
3  do {
4    ...
5    list_for (v: graph.vertices) {
6      value = 0.85 * v.pagerank / v.out_degree;
7      list_for (w: v.successors) {
8        arg = (w, value);
9        put(w.id, function(w, value) {
10         w.next_pagerank += value;
11       }, &arg, sizeof(arg), &w.next_pagerank);
12     }
13   }
14   barrier();
15   ...
16 } while (diff > e && ++count < max_iteration);
```

Evaluation configuration

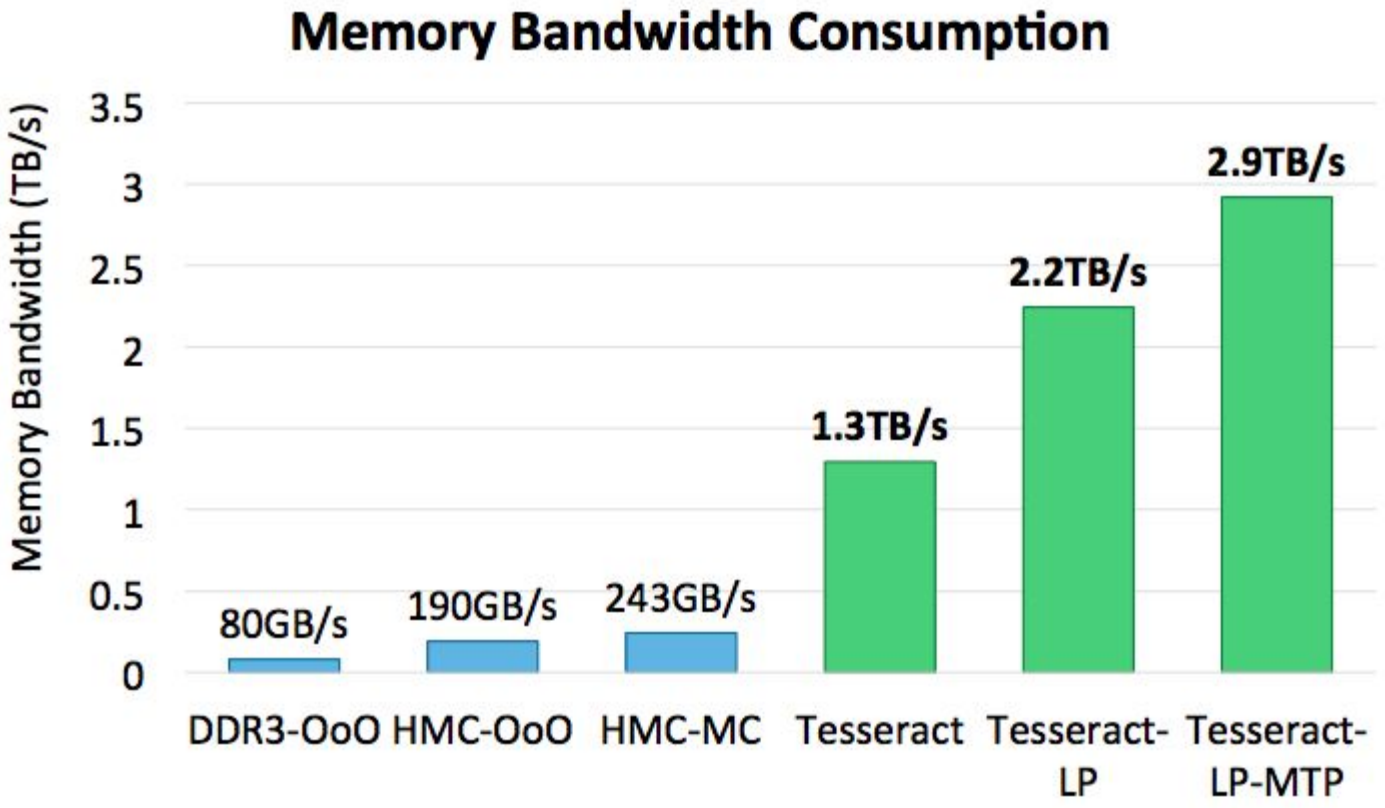
- Simulation Configuration
  - DDR3-OoO System
    - 32 4Ghz cores, DDR3-1600, 102.4 GB/s
  - HMC-OoO
    - Same processor as above, 16* 8GB of memory, 640 GB/s
  - HMC-MC
    - 512 2Ghz cores, 15*8 GB of memory, 640 GB/s
  - Tesseract System
    - 512 2Ghz cores, 8TB/s
- Workloads
  - Teenage Follower, Conductance, PageRank, Single-Source Shortest Path
  - Vertex Cover



Slide Courtesy of Prof.Onur Mutlu

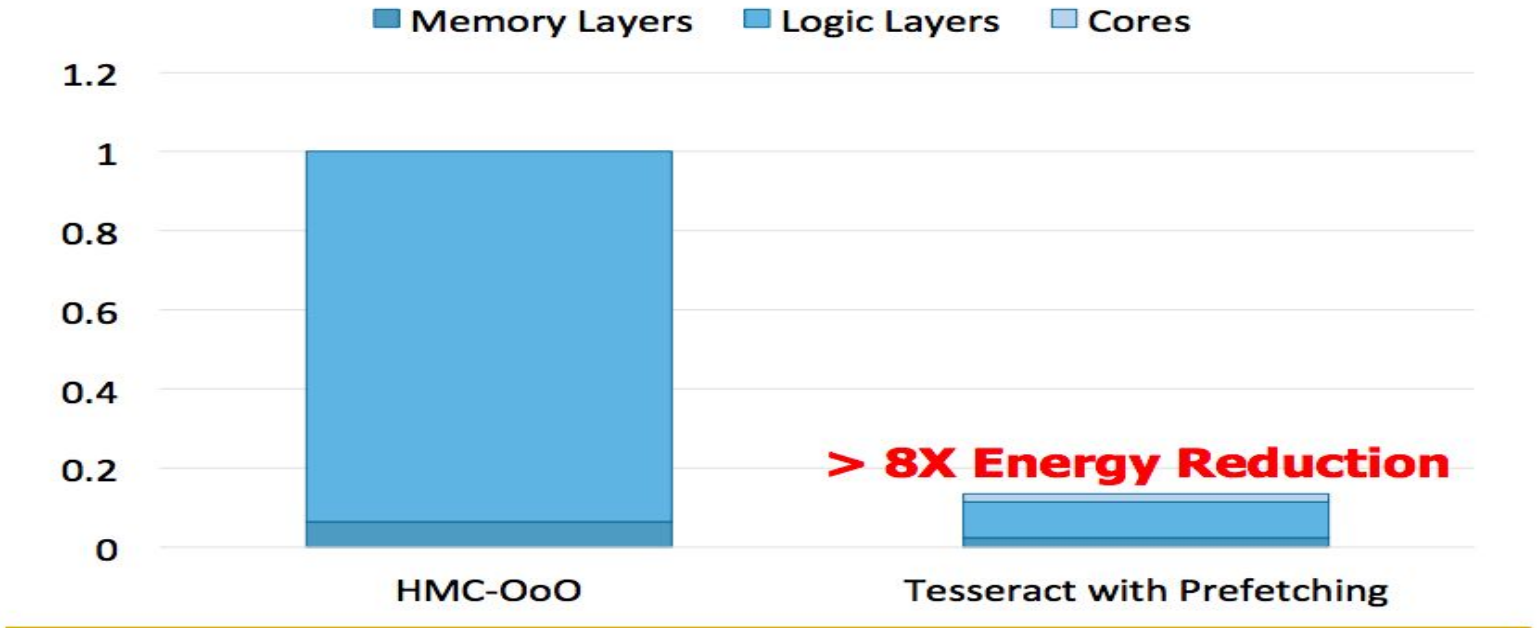


Slide courtesy of Prof. Onur Mutlu



Slide courtesy of Prof. Onur Mutlu

Energy Comparison



Slide courtesy of Prof. Onur Mutlu

Suggestions

- Clarify the bandwidth between vaults and cube interconnects
- Add examples of graph processing APIs as a reference
- Explain the pagerank code better
  - Which code is executed where
  - w.next_pagerank += value

Is it worth it?

- More complicated computation per vertex?
  - Can’t be handled by simple cores
- More properties per vertex that could overflow local memory?
- If the computation is too simple, would the data copying between host and Tesseract become a bottleneck?
- Inefficiencies in unused internal memory bandwidth (2.9 TB vs 8 TB/s)
- Depends heavily on how host distributes graphs among the cubes.


Q&A