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

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

Devdutt Nadkarni, Chenrong Gu

J. Ahn, S. Hong, S. Yoo, O. Mutlu and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, USA, 2015, pp. 105-117, doi: 10.1145/2749469.2750386.

Authors



Junwhan Ahn - was SNU Phd, now at Google

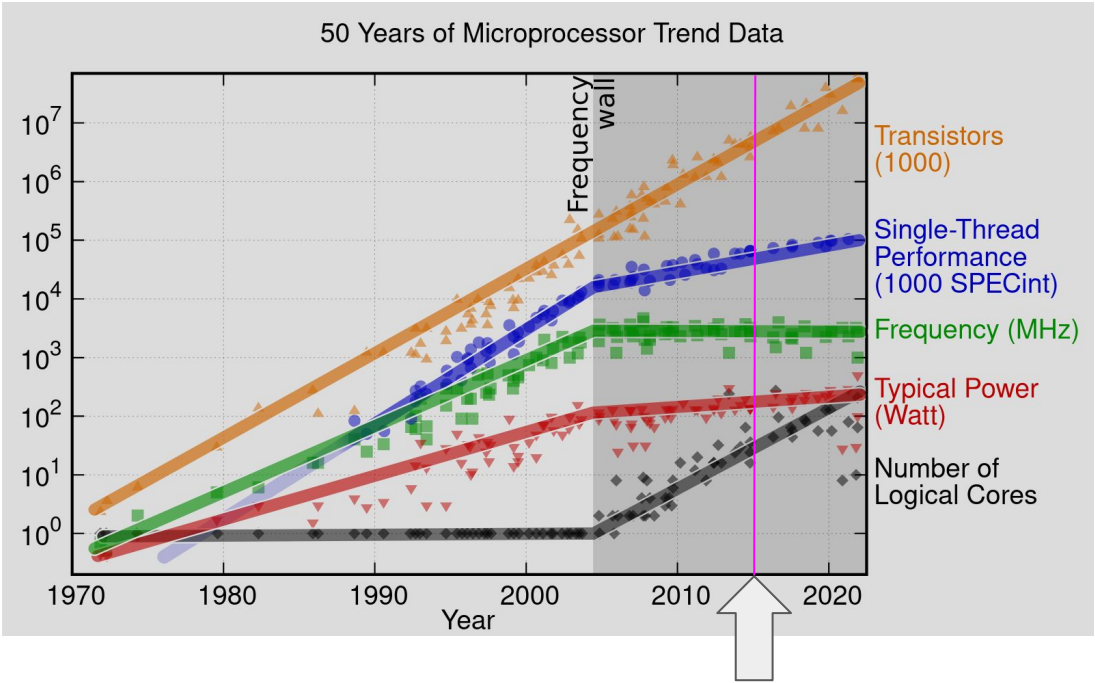
Sungpack Hong - now VP at Oracle

Sungjoo Yoo - Professor at SNU

Onur Mutlu - was at CMU, now Professor at ETH

Kiyoung Choi - Professor at SNU


Place in Timeline



Motivation

Graphs have become increasingly important in our daily lives

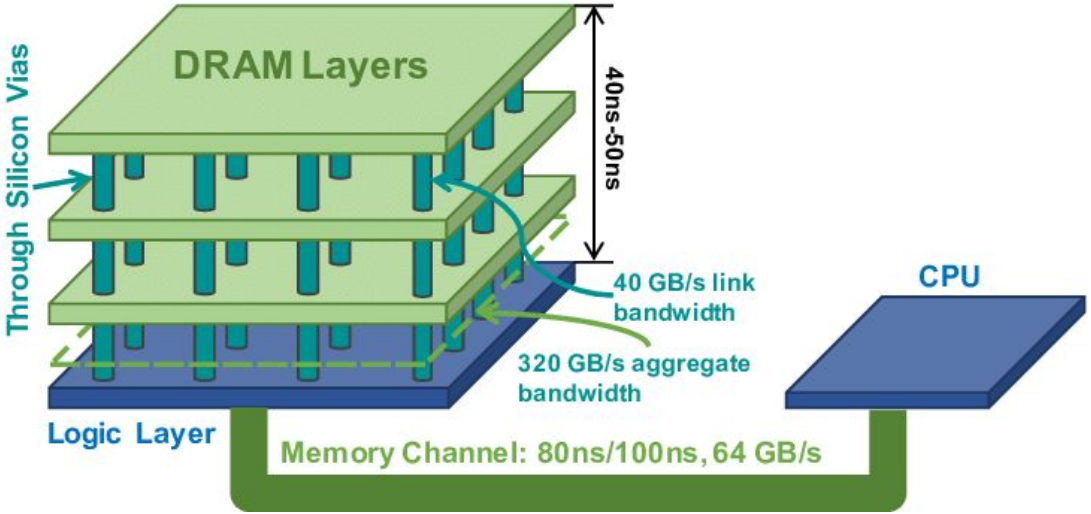
- Memory access patterns are typically hard for traditional OoO cores
- Memory footprints are large (often don’t even fit in DRAM)
- Immediate reutilization of addresses are infrequent

>8.5 billion searches per day

>100 million photos shared per day

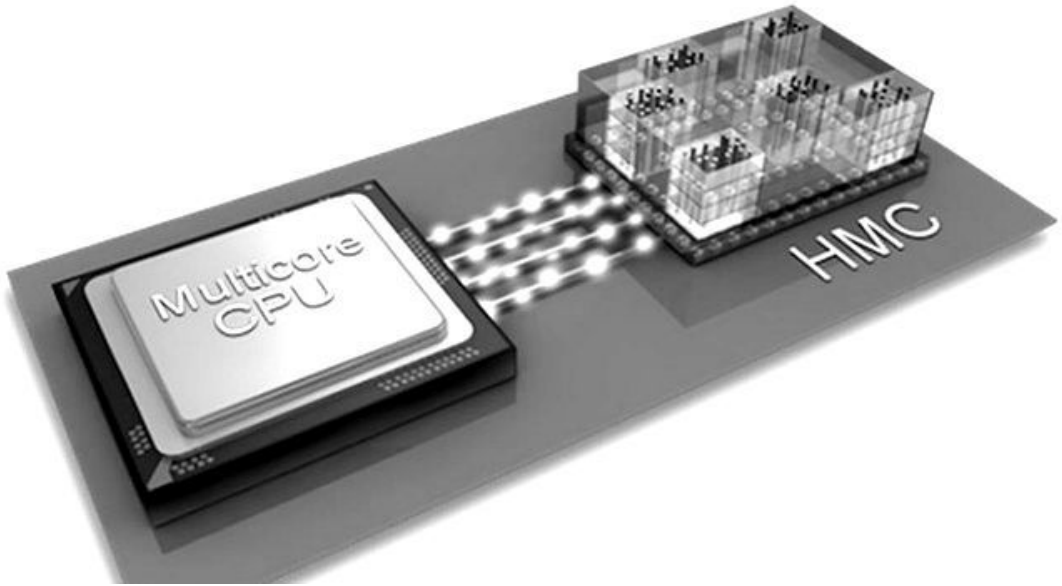
3D integration (and HMC)

- Enable stacking logic and memory dies in a single package



Modern 3D packaging in DRAM

Difficulties with HMC

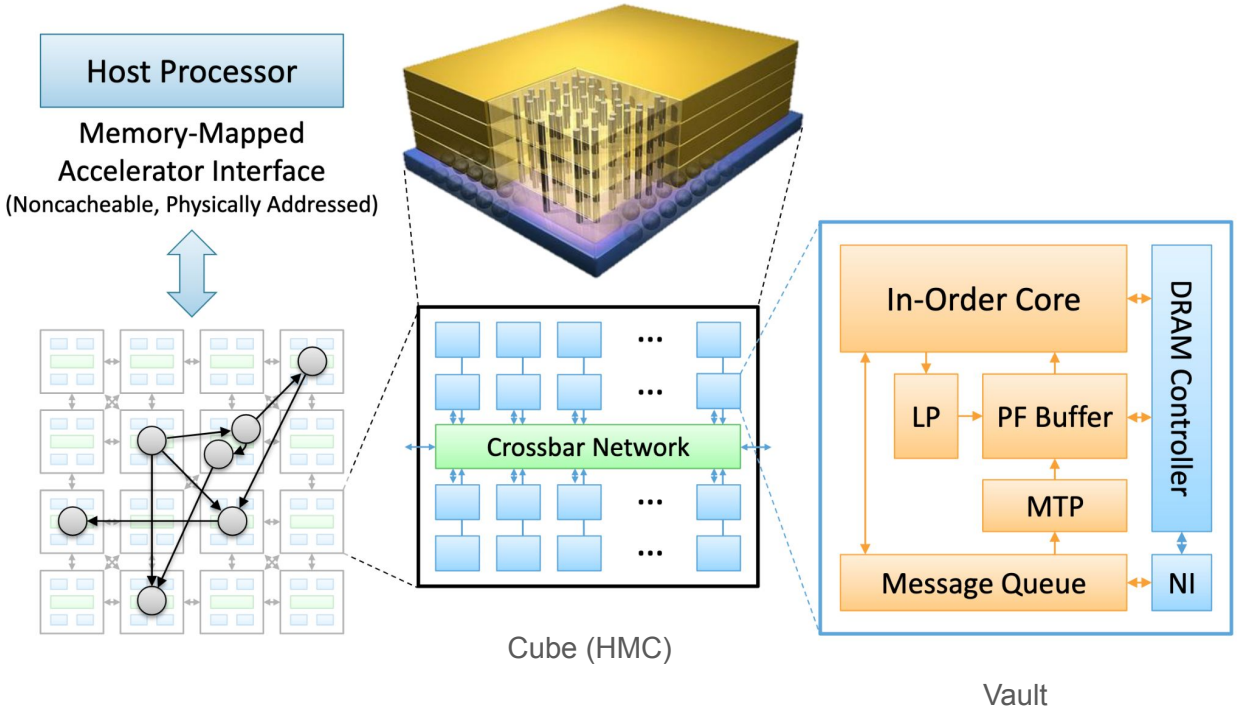


Bandwidth + Scaling (pin count)

Tesseract Architecture

Hybrid Memory Cube

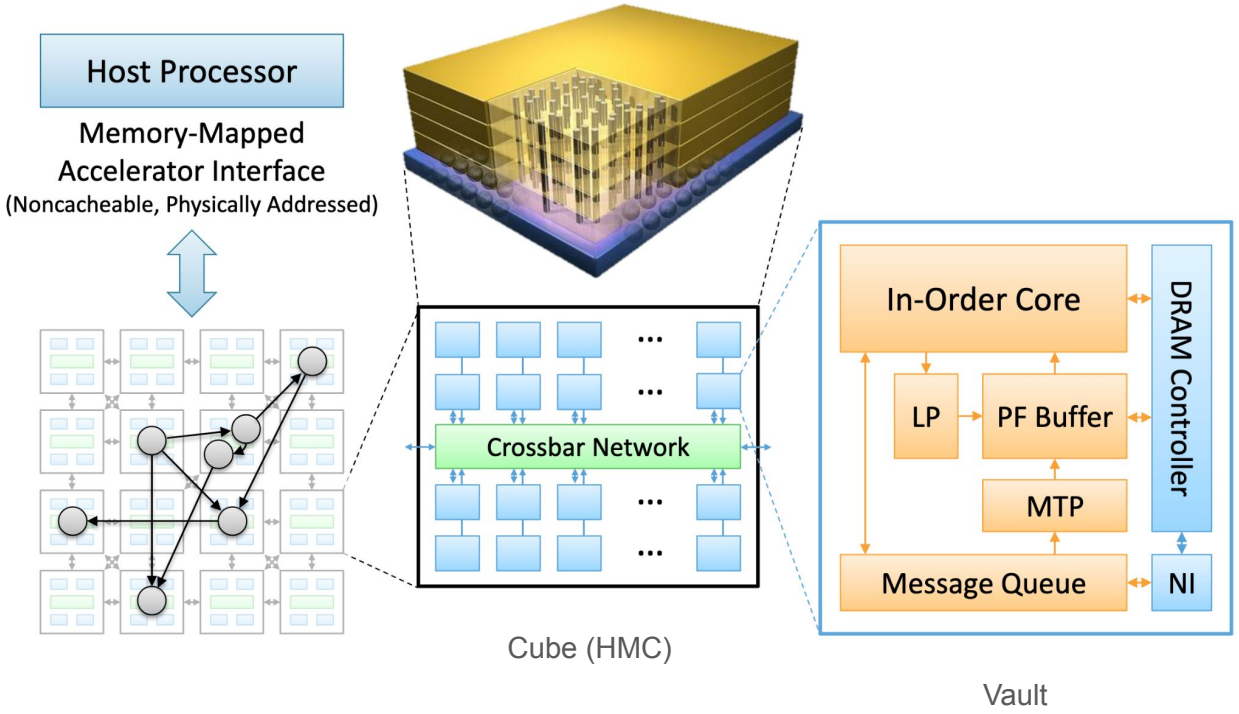
- 8 8Gb DRAM layers
- 8 40GB/s off-chip serial link interface = 320GB/s
- 32 vertical slices (vaults)
- In each vault:
  - 16-bank DRAM partition
  - 16GB/s internal bandwidth
  - Memory controller



Tesseract Architecture

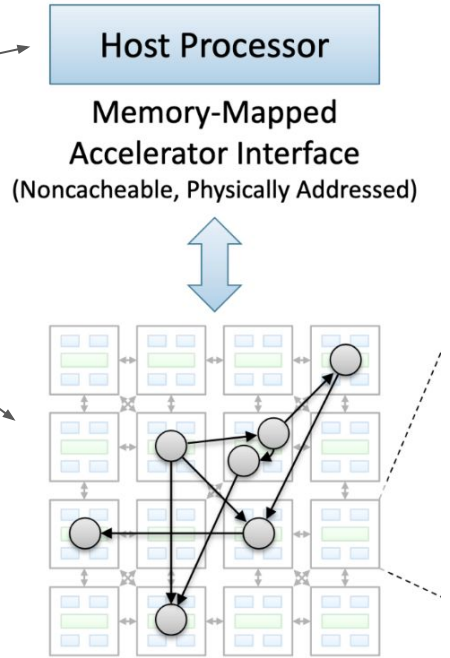
Hybrid Memory Cube

- 8 8Gb DRAM layers
- 8 40GB/s off-chip serial link interface = 320GB/s
- 32 vertical slices (vaults)
- In each vault:
  - 16-bank DRAM partition
  - 16GB/s internal bandwidth
  - Memory controller
  - **ARM Cortex-A5 in-order processor**
  - **With an area overhead 9.6%**



Host-Tesseract Interface

- **Host Processor** has its own memory without PIM
- **Tesseract** 3D memory is noncacheable
- No need for cache coherence between **host processor** and **Tesseract**
- **Tesseract** has no support for virtual memory
- **Host processor** distributes graphs across HMC vaults



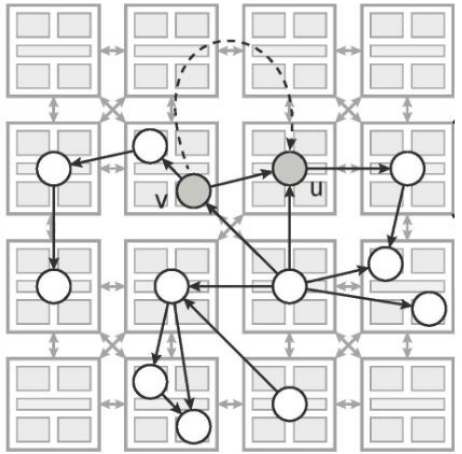
Message passing

- Each Tesseract core accesses its own DRAM partition
- No remote memory access
- Need to send computation to the remote core with right data
- How to communicate between each other?

```
for (v: graph.vertices) {
    for (w: v.successors) {
        w.next_rank += weight * v.rank;
    }
}
```

Message passing

- Each Tesseract core accesses its own DRAM partition
- How to communicate between each other?
- v can remotely update u by sending a **message** (target id and computation)
  - Avoid coherence between Tesseract core caches
  - Guarantee atomic updates of shared data
  - Hide latency through asynchronous messaging



Blocking remote function call

1. Local core sends a packet (function address and argument) to remote core
2. Network interface interrupts remote core
3. Remote core executes function in *interrupt mode*
4. Remote core switches back to normal execution mode
5. Remote core sends return value back to local core

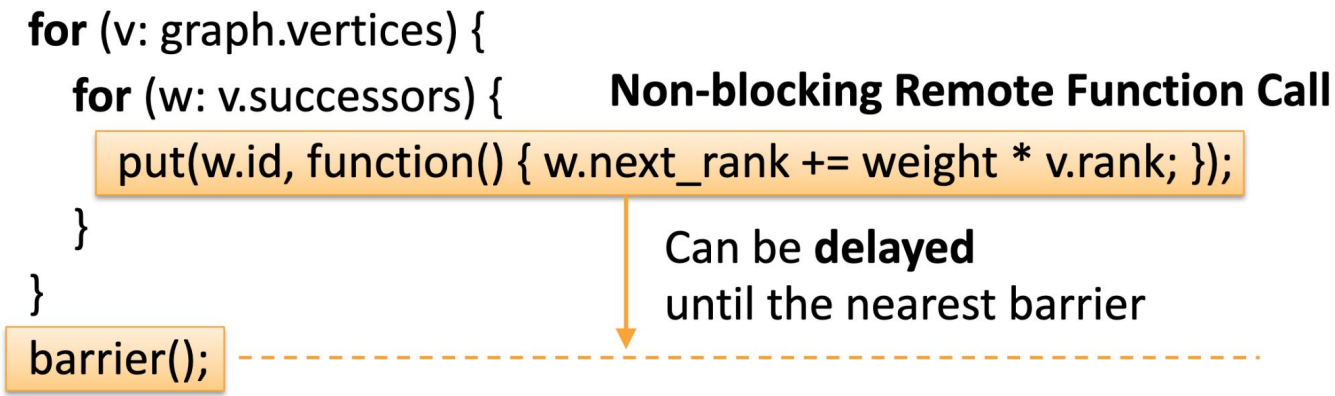
- Interrupt can be disabled during the process
- Used for checking conditions (*e.g. “diff > e”*)

Performance?

- Local cores are blocked until function return
- Each function call emits an interrupt

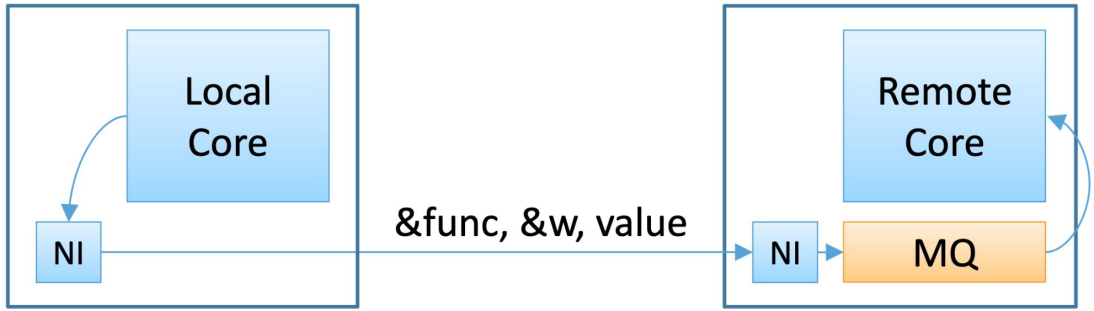
Non-blocking remote function call

- No return values, for updating remote data
- Local core is not blocked
  - Non-blocking function calls do not cross synchronization barriers
  - Results viable after a barrier
- Can be delayed and executed together with a single interrupt



Non-blocking remote function call

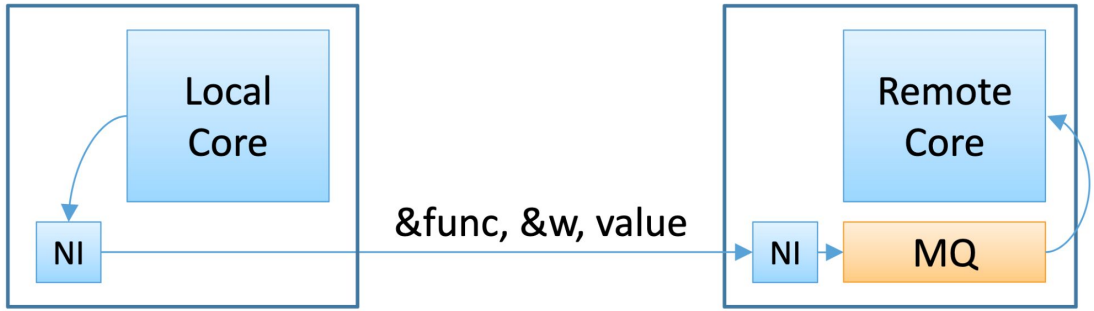
- Message queue
  - Remote core stores the incoming message to the message queue
  - Flush the queue when certain conditions are met with an single interrupt to the core



put(w.id, function() { w.next_rank += value; })

Non-blocking remote function call

- Hide latency because local cores are not blocked
- No off-chip traffic due to remote function call
- Synchronization - atomic function calls, and only current core can access current data
- Prefetching



put(w.id, function() { w.next_rank += value; })

Prefetching

- Each Tesseract core has 16GB/s internal memory bandwidth
- How to enable in-order cores to utilize large bandwidth?
- List prefetching
- Message-triggered prefetching
- Prefetch buffer

List prefetching

- Sequential memory accesses when traversing a list
- Reference prediction table
- For each list:
  - Software provide start address, size, and stride

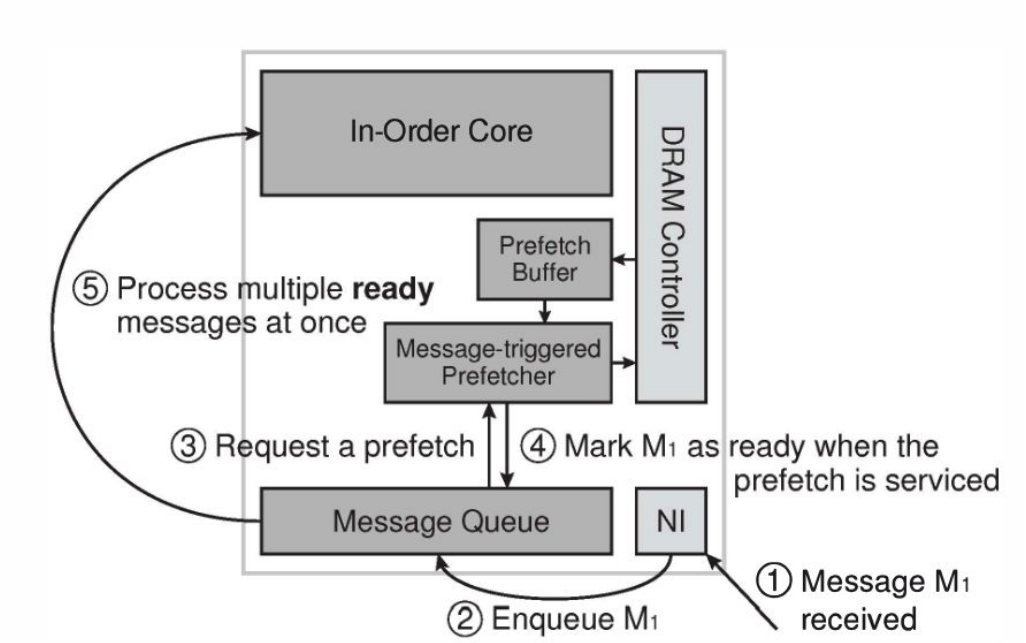
```
for (v: graph.vertices) {
    for (w: v.successors) {
        w.next_rank += weight * v.rank;
    }
}
```

Message-triggered prefetching

- Besides sequential memory accesses, how to prefetch random memory accesses?
- We know target data address when doing remote function calls
- Non-blocking function calls have time slack before execution
- We can track non-blocking function calls:
  - Add target memory address in function call
  - Prefetch when remote core receives the function call
  - Process only **ready** messages in the message queue

```
put(w.id, function() { w.next_rank += value; }, &w.next_rank)
```

Message-triggered prefetching



put(w.id, function() { w.next_rank += value; }, &w.next_rank)

Prefetch buffer

- Store prefetch data into a buffer
- Avoid prefetch data being evicted in L1 cache during the waiting time

Programming interface

- Blocking function calls:

get (id, A func, A arg, S arg_size, A ret, S ret_size)

- Non-blocking function calls:

put (id, A func, A arg, S arg_size, A prefetch_addr)

- Nonpreemption:

disable_interrupt() / **enable_interrupt**()

- Copy data from local vault to a remote vault when transfer size exceeds function argument:

copy(id, A local, A remote, S size)

- List prefetching:

list_begin(A address, S size, S stride) / **list_end**(A address, S size, S stride)

- Synchronization barrier:

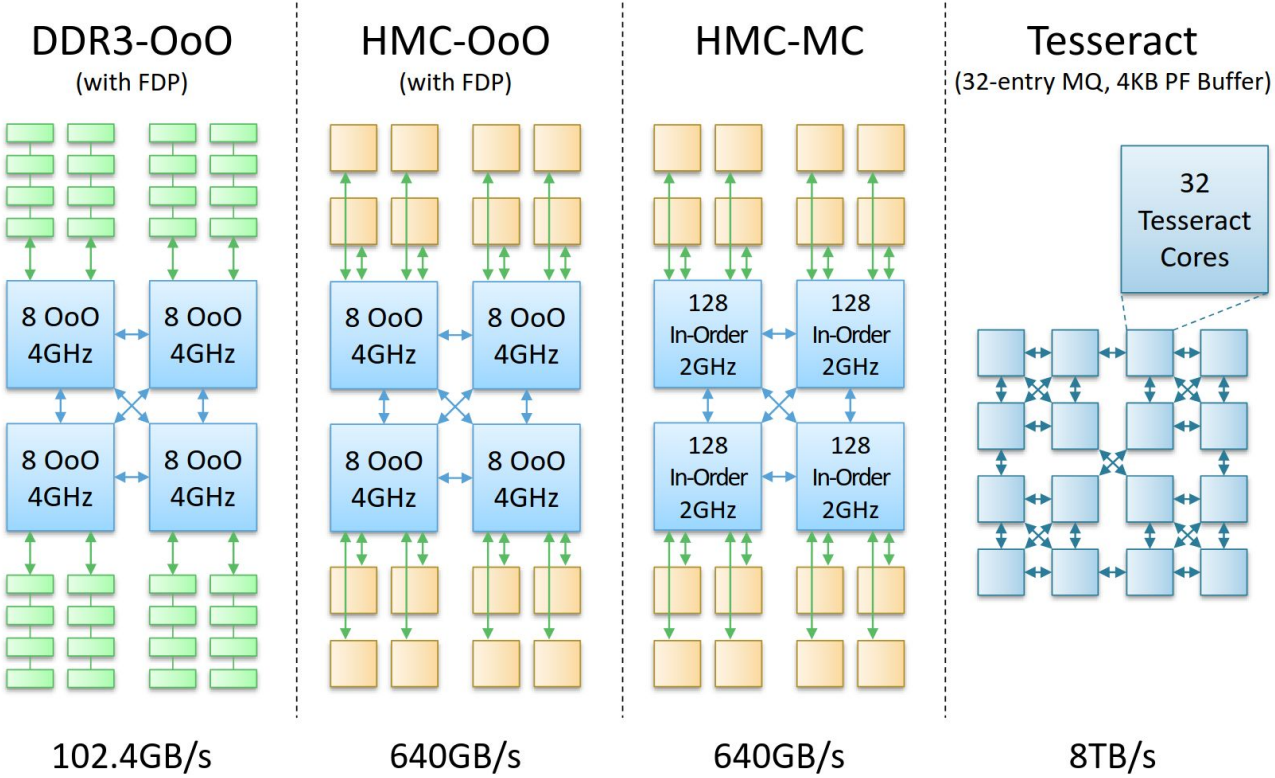
barrier( )

PageRank

```
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  }
```

```
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);
```

Evaluated Systems



Benchmark Caveats

- Performance is heavily dependent on input graph and size
- Since graphs are large, simulation is prohibitively time consuming
  - PR, VC: 1 iteration
  - SS: 4 iterations

Abbreviations used in Benchmarks

HMC- hybrid memory cube

MC - many in order cores

LP - list prefetching

MTP - Message triggered prefetching

Performance Results

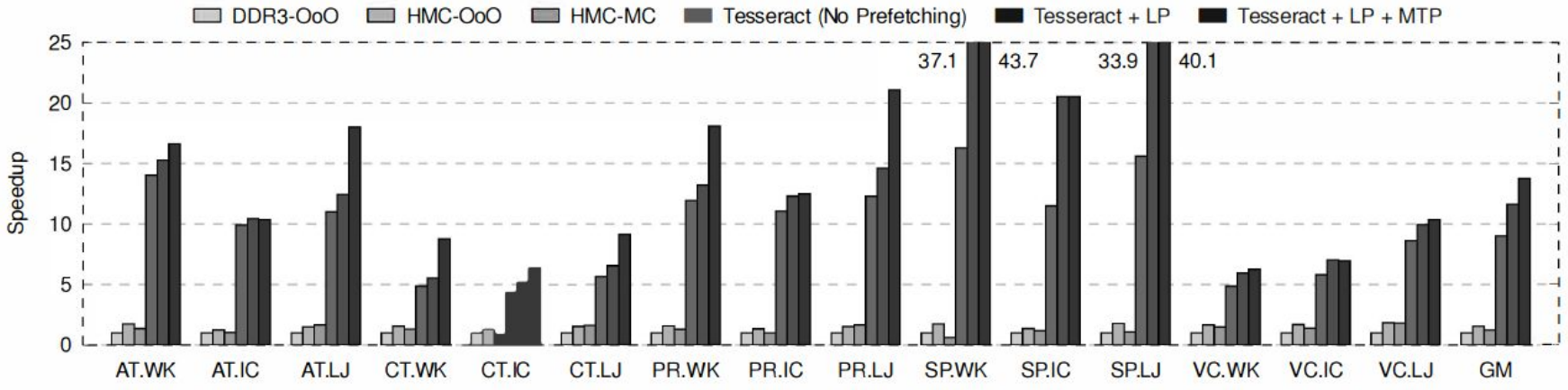
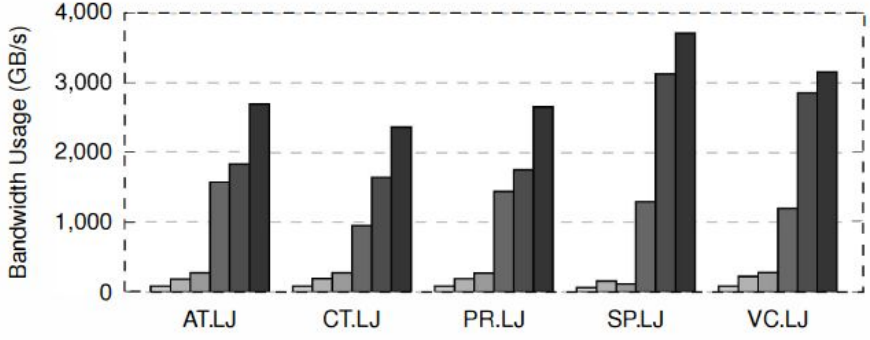
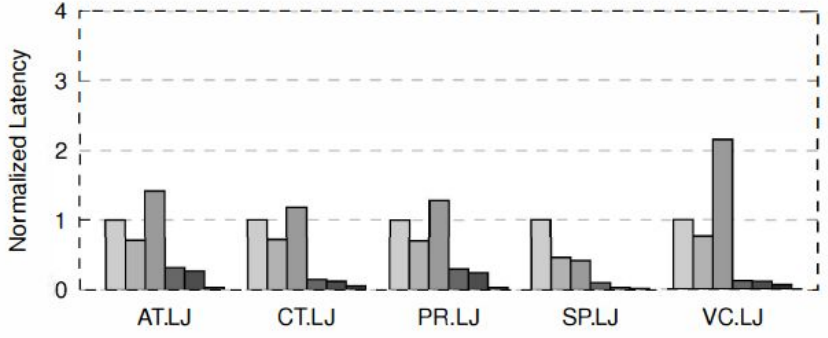
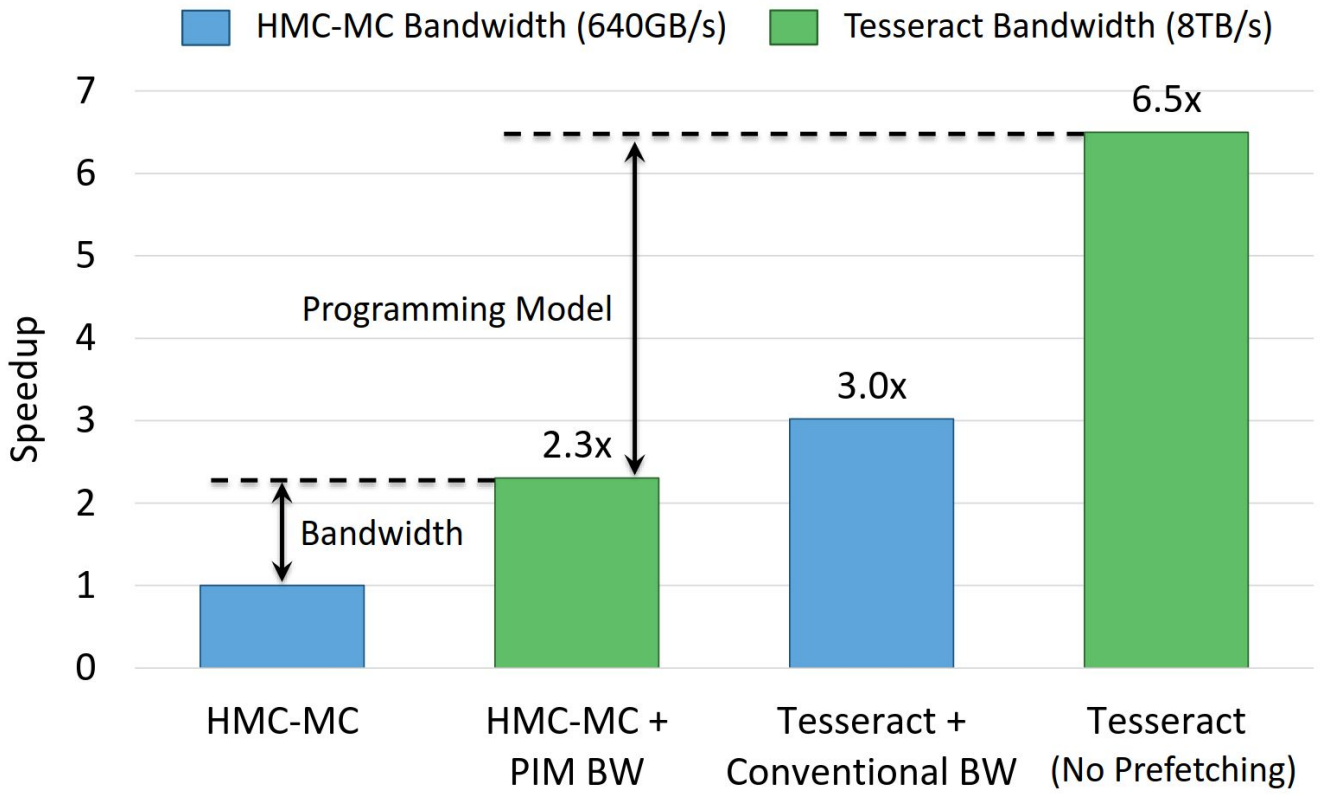


Figure 6: Performance comparison between conventional architectures and Tesseract (normalized to DDR3-OoO).

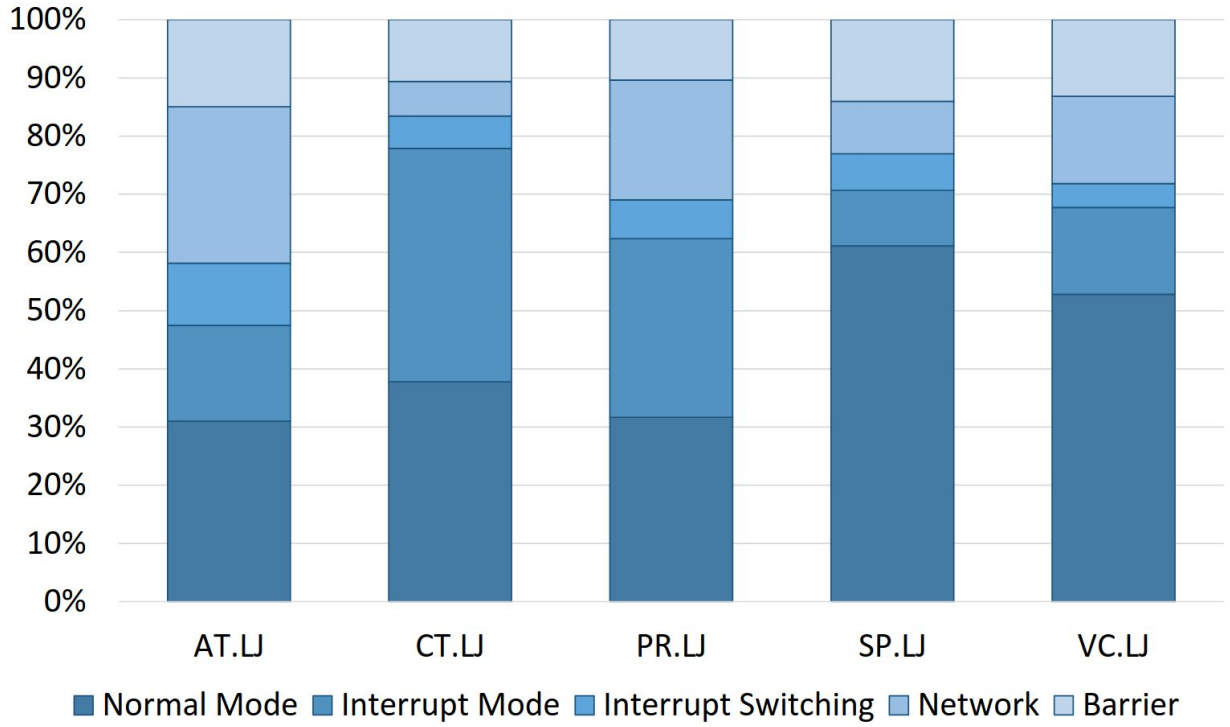




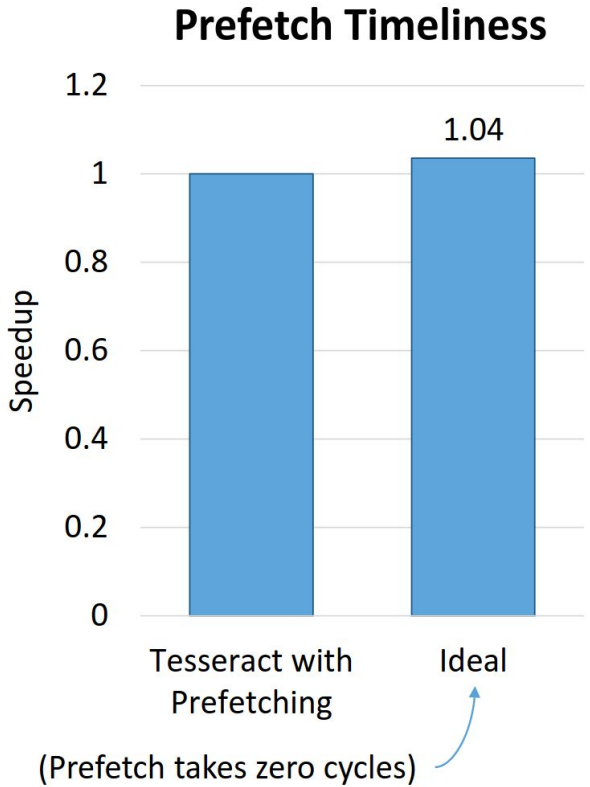
Performance given same memory bandwidth

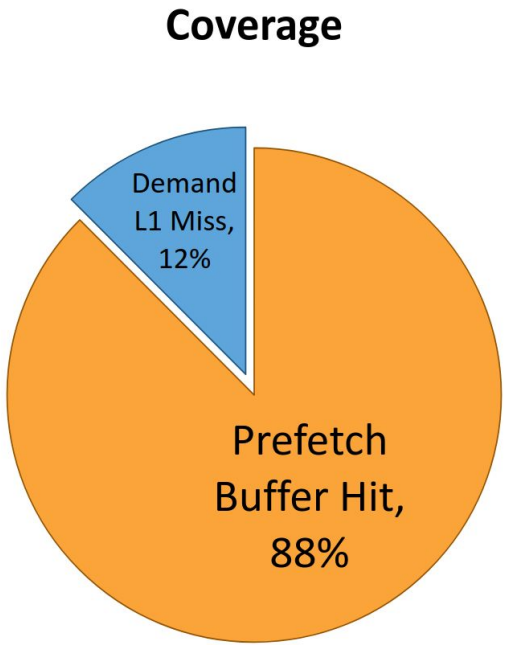


Execution time

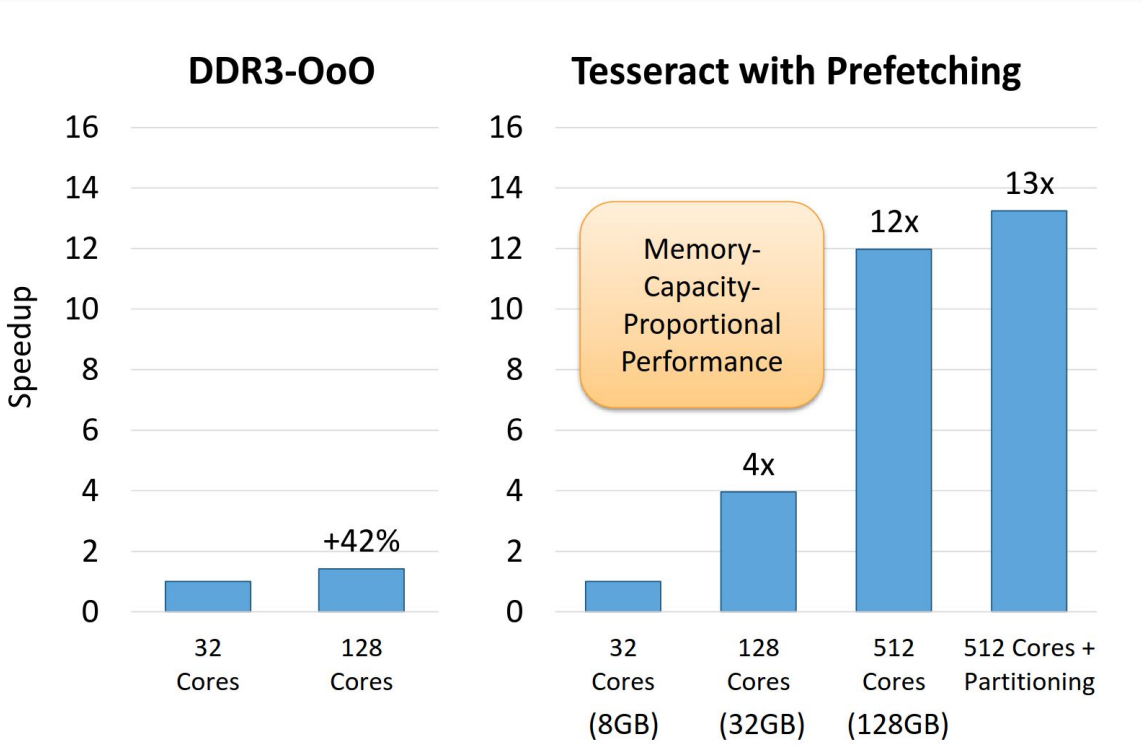


Waiting for memory accesses

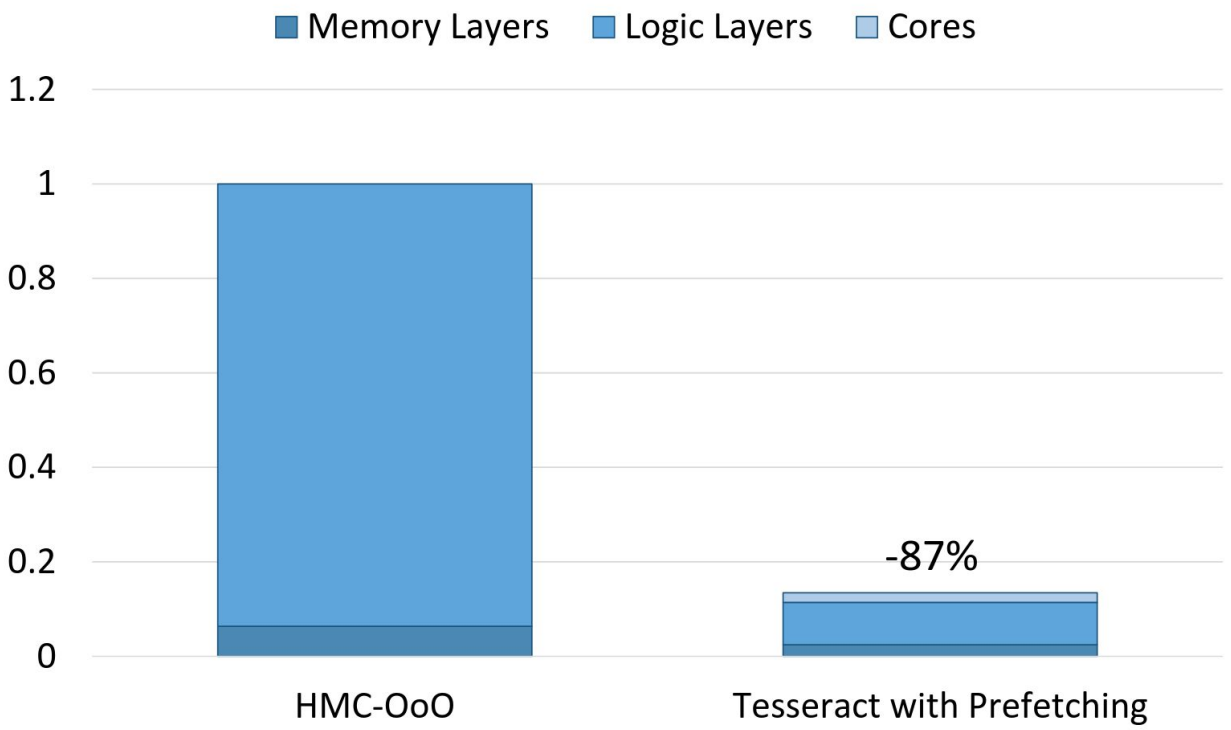




Scalability



Energy Consumption



Pitfalls

- Scaling sometimes sublinear (especially when increasing core count)
- Cost of off-chip network communication (and backpressure)
- Workload imbalance (especially with non-random scheduler)
  - Dynamic migration based load balancing could be added
- NO VIRTUAL MEMORY
- Could improve prefetching

Conclusion

- As 3D integration becomes more cost effective, PIM becomes more and more attractive
- Message passing for latency hiding can be effective, given that software engineers are willing to write code using a new programming interface
- While power consumption may go up, it currently doesn’t impact DRAM stability, and is associated with faster and lower energy performance
- Tesseract is more scalable (with respect to memory capacity) than most modern designs

Thanks