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

Junwhan Ahn  Sungpack Hong§  Sungjoo Yoo  Onur Mutlu†  Kyoung 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

```java
for (v: graph.vertices) {
    value = 0.85 * v.pagerank / v.out_degree;
    for (w: v.successors) {
        w.next_pagerank += value;
    }
}
```
Reality today
Tesseract Architecture

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

Host Processor $\xrightarrow{\text{Customized \textbf{malloc} Call}}$
Message Passing

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

![Diagram showing message passing](https://via.placeholder.com/150)
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);
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

```java
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

```c
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);
```
Programming interface & Application Mapping

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

```plaintext
... 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);
```
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
>13X Performance Improvement

On five graph processing algorithms

- DDR3-OoO: +56%
- HMC-OoO: +25%
- HMC-MC: 9.0x
- Tesseract: 11.6x
- Tesseract-LP: 13.8x

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