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

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

Seoul National University       *Oracle Labs     +Carnegie Mellon University

International Symposium on Computer Architecture 2015
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

Graph Processing Workloads

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

1 for (v: graph.vertices):
2 for (u: v.successors):
3 u.new_rank = v.rank * weight
4 for (v: graph.vertices):
5 v.rank = v.new_rank
6 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):
3. u.new_rank = v.rank * weight

Each successor might lead you to a whole new subgraph
Simple multiplication computation
Graph Processing on Conventional Systems

Page Rank performance comparison has shown the following:

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

**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)
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

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

-A network of HMC cubes
-Memory mapped accelerator interface, non-cacheable, and no support for virtualization
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
Data needed by a Tesseract core might be present in another vault's 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 * v.rank; })
    barrier()

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

Send function address and arguments to the remote core
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.
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
3 real world graphs:
- ljournall-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

- 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
Evaluation Results

Average Performance

![Bar chart showing speedup for DDR3-OoO, HMC-OoO, HMC-MC, Tesseract, Tesseract LP, and Tesseract LP + MTP with speedup percentages and multipliers.]

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

Average Bandwidth Utilization

<table>
<thead>
<tr>
<th></th>
<th>DDR3-OoO</th>
<th>HMC-OoO</th>
<th>HMC-MC</th>
<th>Tesseract</th>
<th>Tesseract LP</th>
<th>Tesseract LP + MTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Bandwidth (TB/s)</td>
<td>190GB/s</td>
<td>243GB/s</td>
<td>1.3TB/s</td>
<td>2.2TB/s</td>
<td>2.9TB/s</td>
<td></td>
</tr>
<tr>
<td>Average Bandwidth Utilization</td>
<td>0.5</td>
<td>1</td>
<td>1.5</td>
<td>2</td>
<td>2.5</td>
<td>3</td>
</tr>
<tr>
<td>Memory Bandwidth (TB/s)</td>
<td>80GB/s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluation Results

Average Memory Energy Consumption

![Chart showing average memory energy consumption comparing HMC-OoO and Tesseract LP + MTP technologies, with a normalized energy reduction of -87%.]
### 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 |
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
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?
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 offload to the GPU-PIM accelerator: Bandwidth gain

2. How to map the data and schedule the computation to benefit the most:
   Subsequent accesses have a certain offset, thus we can map them together

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

SM Access Breakdown over Vaults (MUMer GPU)

Vault ID

0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Percentage of Accesses

SM ID

0 2 4 6 8 10 12 14 16
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 dense ReRAM crossbars, they do graph computations.

With ReRAMs you can do analog computation.

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


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

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

Seoul National University       *Oracle Labs       +Carnegie Mellon University

International Symposium on Computer Architecture 2015
Backup Slides
Backup Slides

- Normal Mode
- Interrupt Mode
- Interrupt Switching
- Network
- Barrier

<table>
<thead>
<tr>
<th></th>
<th>AT.LJ</th>
<th>CT.LJ</th>
<th>PR.LJ</th>
<th>SP.LJ</th>
<th>VC.LJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>75%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

35
Backup Slides

![Bar Chart](chart.png)

- **AT.LJ**
- **CT.LJ**
- **PR.LJ**
- **SP.LJ**
- **VC.LJ**

- 32 Cores (8 GB)
- 128 Cores (32 GB)
- 512 Cores (128 GB)

**Speedup**

0 4 8 12 16

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()