GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks

Lifeng Nai, Ramyad Hadidi, Jaewoong Sim*, Hyojong Kim, Pranith Kumar, Hyesoon Kim
Georgia Tech, *Intel Labs
INTRODUCTION

Graph computing: processing big network data
- Social network, knowledge network, bioinformatics, etc.

Graph computing is inefficient on conventional architectures
- Inefficiency in memory subsystems
Processing-in-memory (PIM)

- PIM has the potential of helping graph computing performance
- PIM is being realized in real products: Hybrid memory cube (HMC) 2.0
What are the benefits of PIM for graph computing?

- Known benefits of PIM
  - Bandwidth savings, latency reduction, more computation power
- But, they are not good enough
- **We explore something more!**

How to enable PIM for graph in a practical way?

- Minor hardware/software change
- No programmer burden
OVERVIEW

GraphPIM: a **PIM-enabled** graph framework

- We identify a new benefit of PIM offloading
- We determine PIM offloading targets
- We enable PIM without user-application change
KNOWN PIM BENEFITS

More computation power
- Extra computation units in memory

Bandwidth savings
- Pulling data vs. pushing computation command

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-byte READ</td>
<td>1 FLIT (addr)</td>
<td>5 FLITs (data)</td>
</tr>
<tr>
<td>64-byte WRITE</td>
<td>5 FLITs (addr, data)</td>
<td>1 FLIT (ack)</td>
</tr>
<tr>
<td>CPU rd-modify-wr (rd-Miss; wr-evict)</td>
<td>6 FLIT</td>
<td>6 FLIT</td>
</tr>
<tr>
<td>PIM rd-modify-wr</td>
<td>2 FLITs (addr, imm)</td>
<td>1 FLIT (ack)</td>
</tr>
</tbody>
</table>

(FLIT: 16 byte, basic flow unit)
PERFORMANCE BENEFITS

More computation power?
- Limited # of FUs in memory

Bandwidth savings?
- Not BW saturated

Latency reduction?
- Yes, but small
Atomic overhead reduction

- Atomic instructions on CPUs have substantial overhead [Schweizer’15]

RMW: read-modify-write
- Cache operation: cache-line invalidation, coherence traffic etc.
- Data ordering: write buffer draining, pipeline freeze etc.

Because of the characteristics of graph programming model, PIM offloading can avoid the atomic overhead

---

H. Schweizer et al., “Evaluating the Cost of Atomic Operations on Modern Architectures,” PACT’15
ATOMIC OVERHEAD REDUCTION

CPU

Pipeline Stall (Serialization)

Data Ordering | Cache Operation | RMW

Retire

Offload

Continue Execution

CPU

Offload

RMW

PIM

Atomic

ACK

Serializaiton in PIM

Georgia Tech

comparch
Atomic overhead experiments on a Xeon E5 machine

- Atomic RMW $\rightarrow$ regular load + compute + store

Atomic instructions incur 30% performance degradation

(Non-Atomic: artificial experiment, not precise estimation)
PERFORMANCE BENEFITS

More computation power?
- Limited # of FUs in memory

Bandwidth savings?
- Not BW saturated

Latency reduction?
- Yes, but small

Atomic overhead reduction?
- Yes and significant!
  - Main source of PIM benefit for graph
OFFLOADING TARGETS

Code snippet: Breadth-first search (BFS)

```java
F ← {source}
while F is not empty
    F' ← {∅}
    for each u ∈ F in parallel
        d ← u.depth + 1
        for each v ∈ neighbor(u)
            ret ← CAS(v.depth, inf, d)
            if ret==success
                F' ← F' ∪ v
    endfor
endfor
barrier()
F ← F'
endwhile
```

- **F**: frontier vertex set of current step
- **F'**: frontier vertex set of next step
- **u.depth**: depth value of vertex u
- **neighbor(u)**: neighbor vertices of u
- **CAS(v.depth, inf, d)**: atomic compare and swap operation

- **Line 4-5, 8-10**: accessing meta data
- **Line 6**: accessing graph structure

**Offload atomic operations on graph property**

**Cache Unfriendly + Atomic**

**Cache Friendly**
How to indicate offloading targets?

Option #1: Mark instructions
- New instructions in ISA
- Requires changes in user-level applications

Option #2: Mark memory regions
- Special memory region for offloading data
- Can be transparent to application programmers
Graph computing is **framework**-based

- User application is designed on top of framework interfaces
- Data is managed within the framework

```c
G = load_graph(“Fruit”);
V1 = G.find_vertex(“Apple”);
V1.property().price = 5;
V1.add_neighbor(“Orange”);
```

**Graph Lab**

**GraphX**

IBM System G
ENABLE PIM IN GRAPH FRAMEWORK

User Application

Graph Framework

Graph API

Middleware

Graph Data Management

OS

Hardware Architecture

User Application

Middleware

Graph Data Management

OS

Hardware Architecture

load_graph

G_structure = malloc(size1);
G_property = pmr_malloc(size2);
Open file & load data

malloc() → pmr_malloc()

Graph Property

Host Processor

Core

PIM Offloading Unit

Georgia Tech

comparch
FRAMEWORK CHANGE

PIM memory region (PMR)
- Uncacheable memory region in virtual memory space
- Utilizing existing uncacheable (UC) support in X86

Framework change
- `malloc() → pmr_malloc()`
- `pmr_malloc()`: customized malloc function that allocates mem objects in PMR
ARCHITECTURE CHANGE

PIM offloading unit (POU)
- Identifies **atomic** instructions that are accessing **PIM Memory Region**
- Offloads them as PIM instructions
CHANGES

Software changes:

- No user application change
- Minor change in framework: malloc() → pmr_malloc()

Hardware changes:

- PIM memory region: utilizes existing uncachable (UC) support
- PIM offloading unit (POU): identifies offloading targets

No burden on programmers + Minor HW/SW change
EVALUATION

Simulation Environment

- SST (framework) + MacSim (CPU) + VaultSim (HMC)

Benchmark

- GraphBIG benchmark suite [Nai’15] (https://github.com/graphbig)
- LDBC dataset from Linked Data Benchmarking Council (LDBC)

Configuration

- 16 OoO cores, 2GHz, 4-issue
- 32KB L1/256KB L2/16MB shared L3
- HMC 2.0 spec, 8GB, 32 vaults, 512 banks, 4 links, 120GB/s per link

L. Nai et al. “GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions,” SC’15
EVALUATION: PERFORMANCE

Baseline: No PIM offloading

U-PEI: Performance upper-bound of PIM-enabled instructions [Ahn’15]

Baseline, U-PEI, GraphPIM, %PIM-Atomic

Speedup over Baseline

Up to 2.4X speedup
On average 1.6X speedup

EVALUATION: EXECUTION TIME BREAKDOWN

Breakdown of normalized execution time

- Atomic-inCore: **atomic overhead** of offloading targets (atomic inst.)
- Atomic-inCache: **cache-checking overhead** of offloading targets

![Normalized Execution Time Breakdown](chart)
CONCLUSION

Graph computing is inefficiency on conventional architectures

GraphPIM enables PIM in graph computing frameworks

- Explores a new benefit of PIM offloading: atomic overhead reduction
- Identifies atomic operations on graph property as the offloading target
- Requires no user-application change and only minor change in framework and architecture
THANK YOU!
BACKUP SLIDES
GraphPIM marks memory region statically

- Cons: cannot be **adaptive** to the working set sizes
- But, property accesses to graphs have very high cache misses regardless of graph inputs except for really small graph sizes [JPDC’16, SC’15]
- Pros: **coherence** support between memory and processor-cache is not required
PIM offloading for atomic instructions works fine because...

- The programming model of graph applications naturally avoids consistency issues: *all PIM writes are done before reads*
- Graph applications require only atomicity from atomic instructions
- But, atomic instructions in CPUs don’t allow to specify atomicity without fence

- We also have a follow-up work discussing the consistency issue for PIM instructions in the context of **general applications** [MEMSYS’17]
Graph applications with BSP model naturally avoids consistency issues

- Barriers ensures all PIM writes are done before reads

<table>
<thead>
<tr>
<th>Program Phases</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop: foreach vertex in task queue:</td>
<td>// Reads</td>
</tr>
<tr>
<td>read property</td>
<td>// HMC Inst.</td>
</tr>
<tr>
<td>fetch neighbor list</td>
<td></td>
</tr>
<tr>
<td>foreach neighbor:</td>
<td></td>
</tr>
<tr>
<td>update neighbor property</td>
<td></td>
</tr>
<tr>
<td>update next-iter task queue</td>
<td></td>
</tr>
<tr>
<td>barrier</td>
<td></td>
</tr>
</tbody>
</table>
GraphPIM

- Considers the **separation** of framework and user application
- Proposes a **full-stack** solution: SW framework + HW architecture
- Requires **no** application programmers’ efforts

Users can easily enable/disable GraphPIM by switching between different framework libraries.
Graph on CPUs are not very sensitive to BW changes
- Speedup over baseline system with different HMC link bandwidth
## APPLICABILITY?

<table>
<thead>
<tr>
<th>Category</th>
<th>Workload</th>
<th>Applicable?</th>
<th>Offloading Target</th>
<th>PIM Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Graph Traversal</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Breadth-first search</td>
<td>✔</td>
<td>lock cmpxchg</td>
<td>CAS if equal</td>
</tr>
<tr>
<td></td>
<td>Degree centrality</td>
<td>✔</td>
<td>lock addw</td>
<td>Singed add</td>
</tr>
<tr>
<td></td>
<td>Betweenness centrality</td>
<td>✘Audit</td>
<td>(Floating point add)</td>
<td>(FP add)</td>
</tr>
<tr>
<td></td>
<td>Shortest path</td>
<td>✔</td>
<td>lock cmpxchg</td>
<td>CAS if equal</td>
</tr>
<tr>
<td></td>
<td>K-core decomposition</td>
<td>✔</td>
<td>lock subw</td>
<td>Singed add</td>
</tr>
<tr>
<td></td>
<td>Connected component</td>
<td>✔</td>
<td>lock cmpxchg</td>
<td>CAS if equal</td>
</tr>
<tr>
<td></td>
<td>Page rank</td>
<td>✘Audit</td>
<td>(Floating point add)</td>
<td>(FP add)</td>
</tr>
<tr>
<td><strong>Dynamic Graph</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Graph construction</td>
<td>✘Audit</td>
<td>(Complex operation)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Graph update</td>
<td>✘Audit</td>
<td>(Complex operation)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Topology morphing</td>
<td>✘Audit</td>
<td>(Complex operation)</td>
<td></td>
</tr>
<tr>
<td><strong>Rich Property</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Triangle count</td>
<td>✔</td>
<td>lock add</td>
<td>Singed add</td>
</tr>
<tr>
<td></td>
<td>Gibbs inference</td>
<td>✘Audit</td>
<td>(Compute intensive)</td>
<td></td>
</tr>
</tbody>
</table>
GraphPIM: EVALUATION

GraphPIM speedup over baseline with different dataset sizes

- BFS
- CComp
- DC
- kCore
- SSSP
- TC
- BC
- PRank

Dataset sizes:
- LDBC-1M
- LDBC-100k
- LDBC-10k
- LDBC-1k
Normalized uncore energy consumption

On average, GraphPIM saves 37% of uncore energy because of reduction in cache accesses and memory bandwidth.
Normalized bandwidth consumption with request/response breakdown
Performance and energy results of two real-world applications

- Based on an analytical model
- FD: Financial fraud detection; RS: Recommender system

GraphPIM: Evaluation

Normalized Energy Breakdown

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>GraphPIM</th>
<th>Baseline</th>
<th>GraphPIM</th>
<th>Baseline</th>
<th>GraphPIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>FD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Caches | HMC Link | HMC Other

Speedup over baseline
PIM Memory Region (PMR)
- A uncachable memory region in virtual memory space
- Utilizing existing uncachable (UC) support in X86

PIM Offloading Unit (POU)
Profiling using HW performance counters

- Execution cycle breakdown: top-down methodology from Intel

**Bottleneck caused by backend stalls**

**High number of cache misses**
BACKGROUND: PIM OFFLOADING IN HMC 2.0

Hybrid Memory Cube (HMC) 2.0

- One of the first industrial PIM proposals
- Instruction-level PIM offloading

- 1 logic die + 4/8 DRAM dies
- 32 Vaults
- 4 serial links
BACKGROUND: PIM OFFLOADING IN HMC 2.0

Packet-based protocol

<table>
<thead>
<tr>
<th>Request</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-byte READ</td>
<td>1 FLIT</td>
</tr>
<tr>
<td>64-byte WRITE</td>
<td>5 FLITs</td>
</tr>
</tbody>
</table>

8 byte | 0~256 byte | 8 byte

Tail | Payload | Header

Regular READ/WRITE

- FLIT: 16-byte; basic flow unit
PIM Instruction: **read-modify-write** (RMW) operation

- Similar as regular READ/WRITE, just different **CMD** in the **Header**
- DRAM bank is locked during the whole RMW for atomicity