INSTRUCTION OFFLOADING WITH HMC 2.0 STANDARD
--- A CASE STUDY FOR GRAPH TRAVERSALS

Lifeng Nai
Hyesoon Kim
Graph computing is important

- however it doesn’t perform well on conventional architectures
- Processing-in-memory can help

Nice Feature from HMC 2.0 specification

- Atomic instruction $\rightarrow$ processing-in-memory
- By utilizing HMC atomic instructions, we can improve graph computing applications
Big data is Big Linked Data

Graph computing is important for processing network data.

Big Data includes all sorts of Networks

- Social/Economic/Political Network
- Information/Knowledge Network
- Nature/Bio/Cognitive Network
- Man-Made Technology Network
Graph computing doesn’t perform on conventional architectures.

**IPC of graph benchmarks on Intel Xeon machine**

- BFS
- kCore
- CComp
- SPath
- DCentr
- TC
- GU

<table>
<thead>
<tr>
<th>IPC</th>
<th>twitter</th>
<th>knowledge</th>
<th>watson</th>
<th>roadnet</th>
<th>LDBC</th>
</tr>
</thead>
</table>

**GraphBIG Benchmark Suite** ([https://github.com/graphbig/graphBIG](https://github.com/graphbig/graphBIG))

“**GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions**” *(to appear in SC’15)*
Problem: cache performance [GraphBIG SC’15]

<table>
<thead>
<tr>
<th>L1D Hit Rate</th>
<th>twitter</th>
<th>knowledge</th>
<th>watson</th>
<th>roadnet</th>
<th>LDBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>100%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2 Hit Rate</th>
<th>BFS</th>
<th>kCore</th>
<th>CComp</th>
<th>SPath</th>
<th>DCentr</th>
<th>TC</th>
<th>GUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>60%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L3 Hit Rate</th>
<th>BFS</th>
<th>kCore</th>
<th>CComp</th>
<th>SPath</th>
<th>DCentr</th>
<th>TC</th>
<th>GUp</th>
</tr>
</thead>
<tbody>
<tr>
<td>50%</td>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Graph computing doesn’t perform on conventional architectures.

Problem: cache performance

Offloading -> Processing-in-memory

HMC 2.0 standard provides that capability
Hybrid Memory Cube Specification 2.0

- Released @ 2014 by Hybrid Memory Cube Consortium
- (although hardware is not available yet)

New feature: Atomic Request Commands

- In-memory processing capability
- Read-Update-Write atomic operation
- 16 byte/8 byte granularity
- Integer data only
# HMC ATOMIC INSTRUCTION

<table>
<thead>
<tr>
<th>Type</th>
<th>Data Size</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>Dual 8B</td>
<td>Add immediate</td>
</tr>
<tr>
<td></td>
<td>Single 8B</td>
<td>Add immediate</td>
</tr>
<tr>
<td></td>
<td>8-byte</td>
<td>Increment</td>
</tr>
<tr>
<td>Bitwise</td>
<td>16-byte</td>
<td>Swap</td>
</tr>
<tr>
<td></td>
<td>8-byte</td>
<td>Bit write</td>
</tr>
<tr>
<td>Boolean</td>
<td>16-byte</td>
<td>AND</td>
</tr>
<tr>
<td></td>
<td>16-byte</td>
<td>NAND</td>
</tr>
<tr>
<td></td>
<td>16-byte</td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td>16-byte</td>
<td>XOR</td>
</tr>
<tr>
<td>Comparison</td>
<td>8-byte</td>
<td>CAS/if equal</td>
</tr>
<tr>
<td></td>
<td>16-byte</td>
<td>CAS/if zero</td>
</tr>
<tr>
<td></td>
<td>8/16-byte</td>
<td>CAS/if greater</td>
</tr>
<tr>
<td></td>
<td>8/16-byte</td>
<td>CAS/if less</td>
</tr>
</tbody>
</table>
HMC ATOMIC INSTRUCTION

HMC Atomic Instruction
- Instruction-level offloading

Limitations
- One-memory location
- No floating-point
- No indirect access
- Simple operations only

How to utilize this feature for graph?
Q-in.insert(source);

while(! Q-in.empty()){
    Q-out.clear(); step++;
    for all u in Q-in in parallel
        for each v in u.neighbors {
            if (level[v] == infinity){
                level[v] = step;
                parent[v] = u;
                Q-out.insert(v);
            }
        }
    barrier();
    swap(Q-in, Q-out);
}

Meta data: small size

Graph structure: good locality

Operation: simple integer check & set

Graph property: large size, no locality

BFS Sample Code
Incorporating HMC Atomic Instruction needs only little effort

```c
bool flag[SZ]={false};
for each v in u.neighbors{
    if (level[v] == infinity){
        level[v] = step;
        parent[v] = u;
        Q-out.insert(v);
    }
}

CAS_equal(ptr, T, N) :
if (*ptr == T)
    {*ptr = N; return true;}
else return false;
```
DISCUSSIONS

- Bandwidth
- Applicability
- Cache policy
Bandwidth without using HMC atomic instructions

\[ \text{Bandwidth (in FLITs)} \approx 24E_i(1 - H_p) \]

Bandwidth when using HMC atomic instructions

\[ \text{Bandwidth (in FLITs)} \approx 8E_i \]

- \( E_i \) — # of traversed edges in step i
- \( H_p \) — graph property cache hit rate

\[ \text{BW (metaData)} \approx 0 \quad \text{BW (graphStructure)} \approx E_i \times 4\text{byte} \]

No-HMC: \( \text{BW (graphProp}_{\text{read}}) \approx 2E_i \times 64 \times (1 - H_p) \approx \text{BW (graphProp}_{\text{writeback}}) \)

No-HMC: \( \text{BW (total)} \approx 256E_i(1 - H_p) = 4E_i(1 - H_p) \times 6 \text{ FLIT} = 24E_i(1 - H_p) \text{ FLIT} \)

HMC: \( \text{BW (total)} \approx 2E_i \times 4\text{FLIT} = 8E_i \text{ FLIT} \)
**HMC Atomic Instruction (assuming bypass cache)**

- can save as much as 66% bandwidth
- lower Hp → higher bandwidth saving (no benefit if Hp > 67%)
<table>
<thead>
<tr>
<th>Computation Type</th>
<th>Workload</th>
<th>Applicable?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation on</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Graph Structure</strong></td>
<td>Breadth-first search</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Depth-first search</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Degree centrality</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Betweenness centrality</td>
<td>NO, floating-point operations</td>
</tr>
<tr>
<td></td>
<td>Shortest path</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>K-core decomposition</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Connected component</td>
<td>YES</td>
</tr>
<tr>
<td><strong>Computation on</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dynamic Graph</strong></td>
<td>Graph construction</td>
<td>NO, complicated code/behaviors</td>
</tr>
<tr>
<td></td>
<td>Graph update</td>
<td>NO, complicated code/behaviors</td>
</tr>
<tr>
<td></td>
<td>Topology morphing</td>
<td>NO, complicated code/behaviors</td>
</tr>
<tr>
<td><strong>Computation on</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Graph Property</strong></td>
<td>Triangle count</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Gibbs inference</td>
<td>NO, computation intensive &amp; cache friendly</td>
</tr>
</tbody>
</table>
APPLICABILITY DISCUSSION

Computation on graph structure
- Mostly applicable
- Floating Point support can help

Computation on dynamic graph
- Not applicable
- Support of indirect access and multiple mem locations can help

Computation on graph property
- Not applicable
- Computation intensive, cache friendly
CACHE POLICY DISCUSSION

HMC Atomic-Inst Accessed Data

- Data Value is updated outside the host processor → data coherence issue

Bypass Cache or Not

- Regular Rd/Wr to the same address:
  - **Bypass cache**: Mark the data as un-cacheable
  - **Cache & flush**: Cache for reads, flush when write/atomic-inst happens
  - **Cache & coherence**: Maintain host-memory coherence

- HMC Atomic-Inst:
  - **Bypass cache**: Directly send request to memory
  - **Cache check**: Check cache first, process-in-host when hit
Graph computing has significant performance degradations caused by irregular graph property accesses.

HMC 2.0 atomic instructions show promising potentials to improve graph computing performance by offloading property accesses.

Future work
- Detailed performance analysis
- Trade-offs between different design choices of host processor
- Potential extensions on existing HMC atomic instructions
QUESTIONS?
BACKUP SLIDES
Junwhan Ahn (ISCA’15) “A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing”
   - Fully-programmable in-memory core

   - General PIM instructions, some selected target applications
   - Focus on the architecture design of host processor side

Our Work
   - Strictly follow HMC 2.0 → a more realistic design
   - Focus more on application side → analyze full-spectrum of graph computing
GRAPH WORKLOAD

Zoom-in -> Details

CUDA_KERNEL (graph_structure, graph_property, step) {
  unsigned u = getThreadID();
  if (level[u] == step) {
    for each v in u.neighbors {
      if (level[v] == infinity) {
        level[v] = step + 1;
        parent[v] = u;
      }
    }
  }
}

GPU BFS Code

Meta data
Graph structure
Graph property
GRAPH WORKLOAD

Data access types
- Meta data (small size) → L1 cache
- Graph structure (good locality) → L2/L3 cache
- Graph property (large size, no locality) → Memory

Operations on graph property
- Usually as simple as check-and-set
- Most workloads don’t use floating-point