1. INTRODUCTION  
Processing in Memory (PIM) was first proposed decades ago for reducing the overhead of data movement between core and memory. With the advances in 3D-stacking technologies, recently PIM architectures have regained researchers’ attentions. Several fully-programmable PIM architectures as well as programming models were proposed in previous literature. Meanwhile, memory industry also starts to integrate computation units into Hybrid Memory Cube (HMC). In HMC 2.0 specification, a number of atomic instructions are supported. Although the instruction support is limited, it enables us to offload computations at instruction granularity. In this paper, we present a preliminary study of instruction offloading on HMC 2.0 using graph traversals as an example. By demonstrating the programmability and performance benefits, we show the feasibility of an instruction-level offloading PIM architecture.

2. BACKGROUND  
Starting from HMC 2.0 specification, atomic requests will be supported by standard HMC design [3]. The atomic requests include three steps, reading 16 bytes of data from DRAM, performing an operation on the data, and then writing back the result to the same DRAM location. All three steps occur in an atomic way. As shown in Table 1, several atomic requests are supported, including arithmetic, bitwise, boolean, and comparison operations. Since only one memory location operand is allowed, all operations have to be performed between an immediate and a memory operand. Meanwhile, a response message will be returned with the original value and an atomic flag indicating if the atomic operation is successful or not.

3. GRAPH TRAVERSAL AND ITS OPERATION OFFLOADING  
The atomic instruction support in HMC 2.0 contains obvious limitations because of its one-memory-operand constraint. However, as a practical design happening in upcoming products, utilizing the atomic instructions as an instruction-level offloading method can be meaningful. We observe that because of the instruction limitations, in terms of both performance and programmability, it is well suitable for irregular graph workloads. In this paper, we take graph traversal as an example for a case study.

3.1 Graph traversal behaviors  
Graph traversal is known to be one of the most fundamental and important algorithms in graph computing [8]. Although different traversal methods contain variations in details, their implementations and behaviors are similar. Hence, in this section, we choose the breadth-first-search (BFS) as the object of our analysis.

A code example of parallel BFS is shown in Figure 1. The code goes through a loop that iterates over the steps in a synchronized way. Each step processes the vertices within the working set, which covers the given level of the graph. For each vertex, it checks its neighbors’ depth to see if it is visited. If not, the level value and the parent vertex information are updated. Meanwhile, a response message will be returned with the original value and an atomic flag indicating if the atomic operation is successful or not.
In the CPU BFS example in Figure 1, although the loop can be unrolled, the execution is still synchronized between the host side and the HMC side because the outcome of the atomic CAS is required. However, the topology-driven design of graph traversal on GPUs can relax such requirement. In the GPU implementation, since there is no concept of working sets, the property processing can be achieved in an asynchronous way, which will likely increase the performance benefits. In this case, all CAS requests can be sent asynchronously. The only boundary would be kernel launch/finish.

### 3.3 Bandwidth Modeling

To estimate the performance benefit of the proposed offloading method, we present an analytical model for the off-chip bandwidth.

**Notation:**
- \(V_i/E_i\): number of traversed vertices/edges in step \(i\)
- \(M\): meta data size in step \(i\)
- \(H_m\): meta data/graph property cache hit rate
- \(D_i\): average vertex degree in step \(i\)

**Conventional BFS bandwidth:** With the notations above and assuming 64-byte cache lines, we can have the bandwidth consumption when accessing each data structure as following.

\[
BW(\text{meta\ data}) = M \times (1 - H_m)
\]

\[
BW(\text{graph\ structure}) = V_i \times D_i / 64 \times 64 = V_i \times D_i = E_i
\]

\[
BW(\text{graph\ property\ read}) = 2 \times E_i \times 64 \times (1 - H_p)
\]

\[
BW(\text{graph\ property\ writeback}) = 2 \times E_i \times 64 \times (1 - H_p)
\]

Meanwhile, since the \(H_m\) is close to 100%, we can have the approximate bandwidth of CPU and GPU BFS as following.

\[
\text{Bandwidth(parallel\ BFS)} \approx 256 \times E_i \times (1 - H_p) + E_i
\]

In the context of HMC, the read/write request will be split into 128-bit packets, named as FLIT. Each 64-byte read includes a 1-FLIT request and a 5-FLIT response. A 64-byte write has a 5-FLIT request and a 1-FLIT response. Thus, we can have the bandwidth in FLITs as following.

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

**HMC-based BFS bandwidth:** With instruction offloading, the graph property operations are directly performed within the HMC via sending CAS atomic requests. Each CAS operation includes a 2-FLIT request and a 2-FLIT response. Thus, we can have the bandwidth consumption as following.

\[
\text{Bandwidth(in\ FLITs)} \approx 2 \times E_i \times (2 + 2) = 8 \times E_i
\]

**Bandwidth saving:** With the analytical model above, we can have the bandwidth consumption of BFS with HMC offloading and without offloading. As shown in Figure 3, with HMC offloading enabled, the bandwidth consumption is independent with \(H_p\), showing a constant value. Meanwhile, the bandwidth of conventional BFS is highly correlated with cache hit rate. When the hit rate is 67%, both sides reach the same bandwidth consumption. Because of the irregular access pattern of graph property, the graph property hit rate usually shows a close to 0% value. In that...
case, instruction offloading can save the memory bandwidth as much as 67%. Even when the hit rate is 30%, which is much higher than typical cases, the bandwidth saving still can be more than 50%. Moreover, because of bypassing caches for irregular data, we can also remove the latency overhead of redundant cache lookups.

### 3.4 Applicability

Although HMC 2.0 supports various atomic operations, several limitations exist. These limitations further impose constraints on the applicability of HMC instruction offloading. First, floating point operations are not supported. Thus, applications with heavy floating operations become unfeasible. Second, only one memory operand is allowed. Complex operations involving multiple memory locations have to be separated into multiple requests, leading to unnecessary performance overheads.

Besides the constraints of integer operation and one-memory operand, to apply HMC instruction offloading with minor efforts, the target workloads should contain a significant amount of irregular memory accesses triggered by code sections that can be easily spotted. Many graph traversal applications, such as our example BFS, are exactly this case. In these applications, a large amount of irregular memory accesses happen and these accesses mostly come from a few lines of code that can effortlessly converted into HMC atomic operations.

**Table 2: Summary of HMC atomic applicability with GraphBIG Workloads.** (CompStruct: computation on graph structure (graph traversals). CompDyn: computation on dynamic graph. CompProp: computation on graph properties.)

<table>
<thead>
<tr>
<th>Workload</th>
<th>Computation Type</th>
<th>Applicable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-first search</td>
<td>CompStruct</td>
<td>✓</td>
</tr>
<tr>
<td>Depth-first search</td>
<td>CompStruct</td>
<td>✓</td>
</tr>
<tr>
<td>Degree centrality</td>
<td>CompStruct</td>
<td>✓</td>
</tr>
<tr>
<td>Betweenness centrality</td>
<td>CompStruct</td>
<td>✓</td>
</tr>
<tr>
<td>Shortest path</td>
<td>CompStruct</td>
<td>✓</td>
</tr>
<tr>
<td>K-core decomposition</td>
<td>CompStruct</td>
<td>✓</td>
</tr>
<tr>
<td>Connected component</td>
<td>CompStruct</td>
<td>✓</td>
</tr>
<tr>
<td>Graph construction</td>
<td>CompDyn</td>
<td>✓</td>
</tr>
<tr>
<td>Graph update</td>
<td>CompDyn</td>
<td>✓</td>
</tr>
<tr>
<td>Topology morphing</td>
<td>CompDyn</td>
<td>✓</td>
</tr>
<tr>
<td>Triangle count</td>
<td>CompProp</td>
<td>✓</td>
</tr>
<tr>
<td>Gibbs inference</td>
<td>CompProp</td>
<td>✓</td>
</tr>
</tbody>
</table>

To summarize the applicability of HMC atomic on graph workloads, we performed code conversion of all graph workloads from GraphBIG, a comprehensive graph benchmark suites [7]. As shown in Table 2, most of the graph traversal oriented workloads, marked as CompStruct type, are applicable to be effortlessly converted with HMC atomic operations, except betweenness centrality, which involves lots of floating point operations. On the contrary, the graph workloads for dynamic graphs perform heavy graph structure/property updates and involve complex code structure and access patterns. Therefore, it is non-trivial to convert these workloads. For CompProp workloads, the triangle count is convertible. However, the performance benefit is an issue here. In CompProp workloads, most computations happen within one vertex’s property. The irregular accesses patterns of graph properties don’t exist. Hence, as a sum, out of the seven graph traversal based workloads, six applications can utilize HMC atomic operations effortlessly. For other graph computation types, HMC atomic is not applicable, even though triangle count still can be converted.

### 4. RELATED WORK

Processing-in-memory (PIM) as a concept has been studied since decades ago. Recently, the advances in 3D stacking technology re-initiated the interest in PIM architectures. As an example, the HMC technology has already demonstrated the feasibility of 3D stacking and PIM approach.

In previous literature, multiple PIM architectures have been proposed. The proposals can be grouped into two major categories, fully-programmable PIM and fix-function PIM [6]. For example, a scalable PIM accelerator was proposed in [1]. It achieves fully-programmable in-memory computation with multiple memory partitions. An instruction-level fix-function PIM architecture was also proposed in [2], in which instructions can be dynamically allocated to host processor or memory side. The existing work all focus on proposing new architectures to enable PIM for performance or power benefits. However, in our study, instead of proposing new architecture prototypes, we utilize an industrial proposal, which is becoming real-world products. In the context HMC 2.0 standard, we analyzed its bandwidth potentials and its applicability on graph computing applications.

### 5. CONCLUSION AND FUTURE WORK

In this position paper, we represented a preliminary study of an instruction-level-offloading method. By utilizing the atomic instructions supported by HMC 2.0 specification, we demonstrated that graph traversals on both CPUs and GPUs can be offloaded via a simple code modification. Meanwhile, in our analytical model, we showed a significant reduction of memory bandwidth consumption. As a case study, this paper illustrated the feasibility of the less-programmable PIM architectures and its potential in the context of HMC 2.0 standard. In our future work, we will perform detailed timing simulations to measure the benefits of graph algorithms on HMC 2.0.

### Acknowledgment

We gratefully acknowledge the support of National Science Foundation (NSF) XPS 1337177. We would like to thank other HPArch members and the anonymous reviewers for their comments and suggestions. Any opinions, findings and conclusions or recommendations expressed in this material...
are those of the authors and do not necessarily reflect those of NSF.

6. REFERENCES


