Instruction Offloading with HMC 2.0 Standard — A Case Study for Graph Traversals

Lifeng Nai
Georgia Institute of Technology
Atlanta, GA 30332
lnai3@gatech.edu

Hyesoon Kim
Georgia Institute of Technology
Atlanta, GA 30332
hyesoon@cc.gatech.edu

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.

Table 1: Summary of Atomic Request Commands

Type	Data Size	Operation	Request	Response
Arithmetic	dual 8B	add immediate	2 FLITs	2 FLITs
	single 16B	add immediate	2 FLITs	2 FLITs
	8-byte	increment	1 FLIT	1 FLIT
Bitwise	16-byte	swap	2 FLITs	2 FLITs
	8-byte	bit write	2 FLITs	2 FLITs
Boolean	16-byte	AND	2 FLITs	2 FLITs
	16-byte	NAND	2 FLITs	2 FLITs
	16-byte	OR	2 FLITs	2 FLITs
	16-byte	XOR	2 FLITs	2 FLITs
Comparison	8-byte	CAS/if equal	2 FLITs	2 FLITs
	16-byte	CAS/if zero	2 FLITs	2 FLITs
	8/16-byte	CAS/if greater	2 FLITs	2 FLITs
	8/16-byte	CAS/if less	2 FLITs	2 FLITs

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, the newly visited vertices will form up the working set of the next iteration. The GPU implementation is slightly different. A CUDA kernel is launched for each BFS step [5]. In each step, there is no concept of working sets. Instead, a thread is started for each vertex. If the vertex is in the corresponding frontier, its neighbors will be checked and updated in the same way as the parallel BFS.

The memory accesses of BFS have three categories, meta data, graph structure, and graph property. The meta data includes local variables and task queues. It is frequently accessed and usually stays inside the cache. The graph struc-

MEMSYS '15 October 05-08, 2015, Washington DC, DC, USA

ACM ISBN 978-1-4503-3604-8/15/10.
DOI: 10.1145/1235

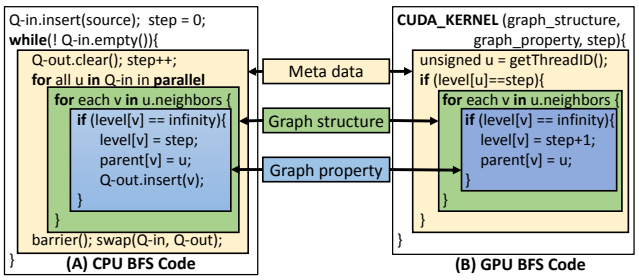

Figure 1: Code Example of BFS on CPU and GPU

ture is accessed for retrieving the neighbor lists. Since the neighbors are typically stored together in a compact way, the graph structure access is not an issue for medium-degree or large-degree graphs. However, the graph property access is the key problem. The property list involves spatially incoherent access spreading across the whole graph [4]. Meanwhile, because of the huge size of graph data, only a small portion of the graph property can stay in the cache. Hence, a property access usually leads to a cache miss and brings in a cache-line (64-bytes) for each edge traversed. Even worse, these cache-lines mostly will be evicted and written back before any further references. Such behaviors result in a high LLC MPKI and a high memory bandwidth consumption.

3.2 Instruction offloading with HMC

In graph traversals, because of the irregular access pattern of graph property, each traversed edge can trigger a 64-byte read from memory, an update operation, and later a 64-byte writeback. However, only 4-byte of the cache-line is used and there is mostly no further references to this cache-line. Therefore, offloading the property computation into the memory is a natural choice. If the property computation is done within the memory, instead of 128-byte data transfer, there will be only one request and one response. It can significantly reduce memory bandwidth consumption and meanwhile the access latency is reduced because of bypassing the cache for cache-unfriendly data.

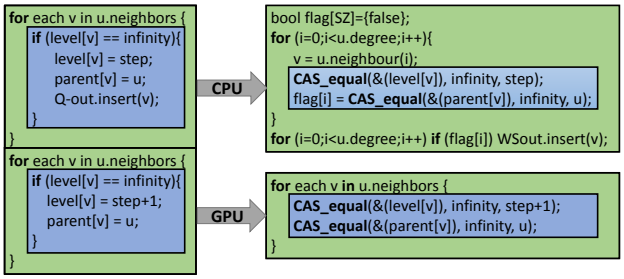

Figure 2: Illustration of HMC Offloading

To illustrate the implementation of instruction offloading for graph traversals, a sample code is shown in Figure 2. We first mark the given graph property arrays, such as the level array and parent array, as uncacheable, so that they won't be cache-residents. Then, as shown, we replace the code section of check-and-update with two atomic CAS operations. In this way, the graph property reads and updates are directly achieved by the memory without wasting extra bandwidth and cache-check time.

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 asynchronized 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:
Vi/Ei: number of traversed vertices/edges in step i
M: meta data size in step i
Hm/Hp: meta data/graph property cache hit rate
Di: 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.

$$BW(meta\_data) = M \times (1 - Hm)$$

$$BW(graph\_structure) = Vi \times Di/64 \times 64 = Vi \times Di = Ei$$

$$BW(graph\_property\_read) = 2 \times Ei \times 64 \times (1 - Hp)$$

$$BW(graph\_property\_writeback) = 2 \times Ei \times 64 \times (1 - Hp)$$

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

$$Bandwidth(parallel\_BFS) \approx 256 \times Ei \times (1 - Hp) + Ei$$

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.

$$Bandwidth(in\_FLITs) \approx 24 \times Ei \times (1 - Hp)$$

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.

$$Bandwidth(in\_FLITs) \approx 2 \times Ei \times (2 + 2) = 8 \times Ei$$

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 Hp, 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

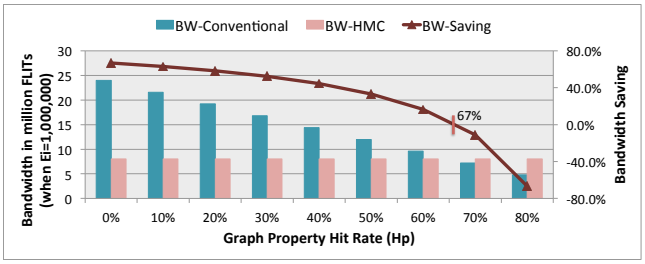

Figure 3: Bandwidth Saving with HMC Offloading

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.)

Workload	Computation Type	Applicable?
readth-first search	CompStruct	√
Depth-first search	CompStruct	√
Degree centrality	CompStruct	√
Betweenness centrality	CompStruct	
Shortest path	CompStruct	√
K-core decomposition	CompStruct	√
Connected component	CompStruct	√
Graph construction	CompDyn	
Graph update	CompDyn	
Topology morphing	CompDyn	
Triangle count	CompProp	√
Gibbs inference	CompProp	

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 convertable. However, the performance benefit is an issue here. In CompProp worklaods, 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 a 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

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA, 2015. ACM.

[3] Altera, ARM, and et al. Hybrid memory cube specification 2.0.

[4] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. IPDPS'12.

[5] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, 2007.

[6] G. H. Loh, N. Jayasena, M. Oskin, and et al. A processing in memory taxonomy and a case for studying fixed-function pim. In *Workshop on Near-Data Processing (WoNDP)*, 2013.

[7] L. Nai, Y. Xia, I. Tanase, H. Kim, and C.-Y. Lin. GraphBIG: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15.

[8] I. Tanase, Y. Xia, L. Nai, Y. Liu, W. Tan, J. Crawford, and C.-Y. Lin. A highly efficient runtime and graph library for large scale graph analytics. GRADES'14.