Boosting the Performance of FPGA-based Graph Processor using Hybrid Memory Cube: A Case for Breadth First Search

Jialiang Zhang, Soroosh Khoram and Jing Li
Department of Electrical and Computer Engineering
University of Wisconsin-Madison
jialiang.zhang@ece.wisc.edu, khoram@wisc.edu, jli@ece.wisc.edu

Abstract

Large graph processing has gained great attention in recent years due to its broad applicability from machine learning to social science. Large real-world graphs, however, are inherently difficult to process efficiently, not only due to their large memory footprint, but also that most graph algorithms entail memory access patterns with poor locality and a low compute-to-memory access ratio. In this work, we leverage the exceptional random access performance of emerging Hybrid Memory Cube (HMC) technology that stacks multiple DRAM dies on top of a logic layer, combined with the flexibility and efficiency of FPGA to address these challenges.

To our best knowledge, this is the first work that implements a graph processing system on a FPGA-HMC platform based on software/hardware co-design and co-optimization. We first present the modifications of algorithm and a platform-aware graph processing architecture to perform level-synchronized breadth first search (BFS) on FPGA-HMC platform. To gain better insights into the potential bottlenecks of proposed implementation, we develop an analytical performance model to quantitatively evaluate the HMC access latency and corresponding BFS performance. Based on the analysis, we propose a two-level bitmap scheme to further reduce memory access and perform optimization on key design parameters (e.g. memory access granularity). Finally, we evaluate the performance of our BFS implementation using the AC-510 development kit from Micron. We achieved 166 million edges traversed per second (MTEPS) using GRAPH500 benchmark on a random graph with a scale of 25 and an edge factor of 16, which significantly outperforms CPU and other FPGA-based large graph processors.

1. INTRODUCTION

The explosion of data poses new challenges to emerging data-intensive workloads ranging from social network analysis to bioinformatics and neural networks. Large sparse graph, which usually contains millions of vertices and billions of edges, is one common data representation in these applications. Among all graph algorithms, breadth first search (BFS) is the most widely used one that serves as a basis of many other more complex algorithms. For instance, BFS is a key kernel in GRAPH500 [1], which is a widely used benchmark suite to measure the performance of super computers for data-intensive applications.

FPGA '17, February 22-24, 2017, Monterey, CA, USA

DOI: http://dx.doi.org/10.1145/3020078.3021737

In traditional CPU-DRAM systems, efficient parallel large graph processing is challenging. Due to the *random* and *data-dependent* memory access pattern requirement of large graph workloads, it is difficult to exploit spatial or temporal locality in on-chip cache. As a result, the system performance is typically bounded by the throughput of external DRAM. However, traditional DDR DRAM suffers from poor random access performance due to the lack of memory level parallelism [2]. To make the situation worse, the high data transfer cost between DRAM and CPU makes it more challenging to parallelize large graph workloads efficiently on such systems, as the synchronization and locking between parallel kernels have become key performance bottlenecks [3].

To address the issues in traditional systems, in recent years, FPGA has been increasingly popular in accelerating graph workloads due to its flexibility, high performance and energy efficiency. Many existing works [4–6] have proposed different architectures to implement BFS on FPGA but are all based on one common scheme. By placing some key data – those used for synchronization between parallel kernels – on the on-chip block ram (BRAM) to alleviate the pressure on accessing external DRAM, the efficiency of processing sparse graph can be significantly improved. However, this scheme does not scale well with large graphs, as the on-chip storage capacity of FPGA is still very limited. As a consequence, these solutions unavoidably suffer from the DRAM bottleneck once the key data of a graph is too large to be fit in the FPGA's on-chip storage.

In this work, we propose a new scheme based on software/hardware co-design and co-optimization, to address the inefficiency of current BFS implementation on FPGA. It effectively combines the emerging hybrid memory cube (HMC) technology, which stacks multiple DRAM dies on top of a base logic layer, with FPGA to effectively accelerate large scale parallel graph workloads. HMC has much better random access performance than traditional DDR DRAM, due to its higher memory-level parallelism [2]. The parallelism mainly comes from two-folds: 1) bank level parallelism: It has a much smaller bank size compared to traditional DRAM, and therefore can fit more banks in a single chip. 2) vault level parallelism: The 3D stacking structure provides additional coarser-grained parallelism at the vault level, as will be explained in section 2.1. Furthermore, HMC supports near-memory operations, such as read-modify-write, locking, *etc.*, on the base logic layer. With all these properties, HMC provides a great opportunity for improving the efficiency of parallel BFS implementation despite of the limitation of FPGA's on-chip BRAM capacity. To leverage HMC's high memory level parallelism and near-memory operation, we propose an improved BFS implementation by taking full advantage of the HMC-FPGA platform, which includes modifications to the BFS algorithm and development of a platform-aware graph processing

architecture. More specifically, we need to change the original BFS execution flow and the data structure to enable the use of near memory operation, as will be explained in section 3.1. Also, by leveraging the parallel processing capability of FPGA-HMC platform, we introduce a Map-Reduce-like framework and present its FPGA implementation in section 3.2.

To achieve an optimal design, we further explore the design space of the FPGA-HMC based graph processing system through theoretical analysis and real hardware experiments. We use GRAPH500 [1] (i.e. BFS algorithm) as the benchmark to evaluate the performance and develop an analytical model to help us identify the potential performance bottlenecks as well as choosing the optimal design parameters. Based on the analysis results, we further propose to use a two-level bitmap to reduce the unnecessary HMC access. Finally, we perform both simulation and practical hardware implementation to validate our design choices.

The key contributions are summarized as follows:

- We develop a graph processing system based on software/hardware co-design and co-optimization, which comprises software modifications of level-synchronized BFS and a platform-aware graph processing architecture, to fully exploit the potential of FPGA and HMC.
- We propose an analytical performance model for our FPGA-HMC based BFS implementation. We then apply the model to perform an in-depth analysis on performance bottlenecks of the design.
- To address the bottlenecks, we propose a two-level bitmap scheme that effectively reduces the unnecessary HMC access to achieve high performance. We further apply our analytical model to perform efficient design space exploration for key design parameter optimization.
- We conduct experiments to verify the effectiveness of proposed techniques. Our implementation achieves 3× performance improvement compared to CPU and outperforms other FPGA based graph processing system.

The rest of the paper is organized as follows. Section 2 presents the background of Hybrid Memory Cube (HMC) and Breadth First Search (BFS). In Section 3, we present the software design and the system architecture of our FPGA-HMC based graph processing system. In Section 4, we present an analytical performance model and apply it to analyze the performance bottlenecks. In Section 5, we present the design, performance analysis and the implementation of proposed two-level bitmap. Section 6 presents the experimental results and validates the proposed techniques. Section 6 concludes the paper.

2. BACKGROUND

In this section, we first provide an overview of the emerging Hybrid Memory Cube (HMC) technology. We analyze its structure and unique properties compared with the traditional DRAM. Then we present the background of breadth first search (BFS) and its parallel implementation.

2.1 Hybrid Memory Cube

HMC is an emerging memory module that stacks multiple DRAM dies on top of a CMOS layer to form a cube using through-silicon-via (TSV) technology. The word "hybrid" describes the fact that HMC combines both memory and logic dies into a single stack.

The architecture of HMC is optimized for parallel memory access. Each DRAM layer is divided into multiple partitions, and

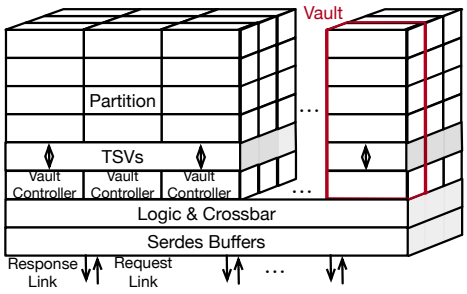

Figure 1: Architecture of Hybrid Memory Cube (HMC) [7]

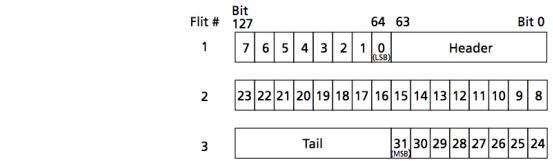

Figure 2: Example of FLIT with 32 Bytes payload

each partition comprises several memory banks. As shown in Figure 1, a vertically connected stack containing multiple partitions from different DRAM layers is called a *vault*. Moreover, each vault also contains a corresponding partition in the logic base layer, which serves as a vault controller. The vault controller manages the DRAM banks within the vault, and thus eliminates the need of off-chip memory controller in traditional DRAM module. Therefore, a HMC *vault* is analogous to the notion of a *channel* in traditional DRAM-based memory system, as it contains all components of a DRAM channel: a memory controller, several memory ranks (*partition*), and a bi-directional bus. We can therefore view HMC as a device that integrates the traditional multi-channel DRAMs into a single chip. In Table 1, we compare a 4 channel 8GB DDR DRAM memory system with a 8GB HMC. From the comparison, we show that HMC has highly fine-grained bank partitions which can be used to serve a large number of concurrent memory requests. Therefore, it offers much higher memory-level parallelism than traditional DRAM. More importantly, the page size of the HMC is only 16B, making it very suitable for random access and alleviates the over-fetch problem in traditional DRAM caused by large page size (several KB) [2]. Moreover, HMC provides an out-of-order memory access to fully exploit internal bank level and vault level parallelism. In general, HMC is expected to provide higher performance compared to traditional DRAM, especially for workloads with a large number of random accesses.

Table 1: Comparison of 8GB DDR4-2133 memory and HMC

	DDR4-2133	HMC
Total Capacity	8GB	8GB
No. of Vault (Rank)	2	32
No. Bank	256,128,64	512
Bank Capacity	32,64,128 MB	16 MB
Page size	1, 2 kB	16 B
Link Speed	19.2GB/s	up to 240GB/s

As shown in Figure 1, vault memory controllers are connected to a high-speed interface communicating with other HMCs or host devices (*e.g.* CPU, GPU, FPGA) via a crossbar switch. The high-speed interface consists of a serialized physical layer and a packetized transaction layer. The physical layer has several links, which can be used to connect to the different hosts. Each link consists of several lanes with a data rate typically higher than 10Gbps per lane. Different from the bi-directional bus of traditional DDR memory interface, the high-speed serial lane transmits data in both directions, which makes the HMC links *full-duplex*. HMC also incorporates a packetized transaction layer that differs from traditional

DDR interface but is similar to PCIe interface. Since the serial interface does not separate data bus from address bus, HMC includes the memory command, address and other information (*e.g.* tag) in the packet header, called "FLIT" (FLow unIT). As shown in Figure 2, FLIT is the smallest unit of data transmission on the high-speed interface. Each transaction may consist of several FLITs depending on the link granularity (ranging from 16 bytes to 128 bytes). *Choosing the size of data payload has significant impact on the performance of HMC*. In Section 4 , we will present the methodology for choosing an optimal data payload size.

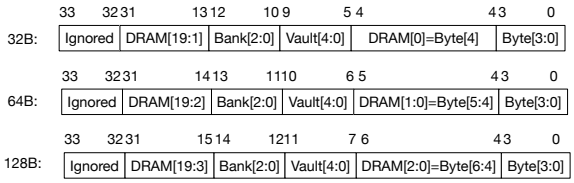

Figure 3: HMC address mapping scheme of 32B, 64B, 128B memory address granularity

Besides link granularity, HMC also has a configurable memory address granularity. As shown in Figure 3, the HMC uses an address field of 34 bits for internal memory addressing, which contains *vault* address, *bank* address, DRAM row and column address (within a bank) and byte address. For different memory access patterns, HMC provides four memory address modes with different granularities (16 bytes, 32 bytes, 64 bytes, 128 bytes). We can obtain different address mapping scheme by changing the size of byte address, which is the maximum payload size for a link packet, as each link packet can only access one vault. With the configurable memory address granularity, we can achieve different trade-offs between latency and throughput by distributing the memory access to different vaults or coalescing the memory access to one vault. In section 4, we will show how to obtain the optimal memory address granularity.

The base logic layer of HMC opens up opportunities for near-data computing. The HMC standard defines several locking and read-modify-write commands which are preferred to be executed by logic units near memory instead of host CPU. Although the idea of near data computing is not new, HMC is the first commercial device to practically implement the concept. In section 3, we will present software modifications to exploit near data operations to improve performance.

2.2 Breadth First Search

Breadth First Search (BFS) is a widely used graph traversal algorithm in broad applications ranging from data analysis in social networks [8] to routing optimization in Electronic Design Automation [9]. In this section, we formally define the BFS problem and its objective. These definitions will be referred to in later sections to analyze and optimize its implementation.

Assuming an unweighted graph *G* with vertex set *V* and edge set *E*, BFS finds a path from a source vertex $v_s \in V$ to all the other vertices in the graph *G*. In the output, for each vertex $v \in V$, BFS will produce a level value l, indicating its distance from v_s (v can be accessed from v_s by traveling through $l-1$ edges), and its father vertex id $f \in V$, indicating the vertex on the path to v which is the direct ancestor of v (naturally $(f, v) \in E$).

BFS traverses a graph by processing all vertices with the same distance from the source vertex iteratively. We define a *frontier* as the set of vertices which have the same distance from the source. We denote the latest known frontier as *current frontier*, and unknown frontier that will be generated based on current frontier as the next frontier in this iterative process.

A detailed description of level-synchronized BFS has been depicted in Algorithm 1. The *level* and *parent* are arrays that store the level and father information for all traversed vertices. Initially, all values in *level* and *father* arrays are set to 0 and -1 respectively. At the beginning of the algorithm (line 2-3), $level[v_s]$ and $father[v_s]$ are set to 1 and $NULL$ because v_s is added to the current frontier (line 4). Then, $current\_level$, which holds the level number currently being processed, is set to 1 (line 5). During the iterative process (i.e. the while loop), at each level, for every vertex v in the *current frontier*, all unvisited neighbors of v (n) are added to next frontier (*next frontier* $\leftarrow n$). Whether a neighbor has been visited or not is determined by checking if its level is non-zero (line 11). The *level* and *father* for these neighbor vertices are set to results calculated from $current\_level$ and the corresponding vertex in *current frontier*, v (line 12, 13). At the end of the iteration, the value of $current\_level$ is incremented, and *current frontier* and *next frontier* are swapped. The algorithm will not be terminated if the *current frontier* is not empty, which means there are still unvisited vertices in the graph. In a multi-thread context, threads that finish the traversal of their portions of the current frontier first should not further proceed until all threads finish the processing of the current frontier for synchronization purpose. Therefore, this algorithm is also called *level-synchronized BFS*.

Algorithm 1 Level-synchronized BFS

```
1:  procedure BFS
2:      level[v_s] = 1
3:      parent[v_s] = NULL
4:      current frontier ← v_s
5:      current_level = 1
6:      while current frontier not empty do
7:          for v ∈ current frontier do
8:              current frontier = current frontier − v
9:              E_v = {n ∈ V|(v, n) ∈ E}
10:             for n ∈ E_v do
11:                 if level[n] is 0 then
12:                     level[n] = current_level + 1
13:                     parent[n] = v
14:                     next frontier ← n
15:         current_level = current_level + 1
16:         Swap currentfrontier with nextfrontier
```

3. BFS IMPLEMENTATION ON HMC-FPGA PLATFORM

In this section, we present our BFS implementation tailored to a system consisting of a FPGA and a HMC. We first describe the software design of the level-synchronized BFS that leverages the advantages of HMC. Then, we present the design details of our FPGA implementation using a Map-Reduce-like framework. Note that as there is no prior effort to implement BFS on FPGA-HMC platform, we will use this implementation as a baseline and compare it with an optimized design in Section 5.

3.1 Software Implementation

To best implement BFS on FPGA-HMC platform, it is important to carefully choose a design that best matches algorithmic behaviors with the available hardware resources to maximize performance and energy efficiency. Furthermore, the design needs to be scalable to accommodate real-world graphs at extremely large scales. Considering all these factors, we implement the following data structures and software execution flow, based on the unique characteristics of the BFS algorithm and hardware resources provided by our FPGA-HMC platform.

3.1.1 Data Structures

BFS requires different data structures for representing the graph, maintaining the intermediate meta data for frontiers, and storing the final results. Here, we describe these data structures, the reasons for choosing them, and how our BFS implementation uses them.

There are generally three types of data structures that are required for BFS algorithm: 1) an adjacency list, which is used to store the graph structure 2) two bitmaps that are used for bookkeeping the information of the current and the next frontiers 3) two arrays, which are required to store the level and parent information for all vertices. The adjacency list representation of a small graph has been depicted in Figure 4. In this data structure, a large adjacency array is allocated to store indices of neighboring vertices for each vertex in the graph back to back. In the vertex array, each slot stands for a vertex in the graph, and stores a pointer to the beginning of its neighbors in the adjacency array. Through this data structure, all neighbors of any random vertex can be easily accessed with one level of indirection. Overall, in case of a directed (or an undirected) graph with $|V|$ vertices and $|E|$edges, adjacency list requires storage of $|V|+1$ values in the pointers array and $|E|$ ($2|E|$ in case of an undirected graph) values in the adjacency array. This data structure provides a good balance between data compactness and random access speed.

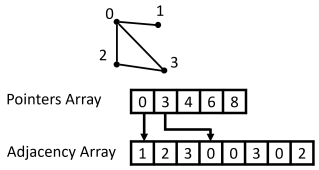

Figure 4: Adjacency list representation of a graph

Bitmaps are used to bookkeep the information for frontiers. A bitmap is an array containing $|V|$ bits, each of which indicates whether the corresponding vertex has been visited or not. To add a vertex to the frontier, we simply set its corresponding bit in the bitmap to 1. As shown in Algorithm 1, we only allocate two arrays to store the current frontier and the next frontier, respectively. At the end of each BFS iteration, we clear all bits in the bitmap of current frontier, and then use the bitmap of the next frontier as the new current frontier to start a new iteration (i.e. the role of the two bitmaps are swapped every time at the end of the iteration). We will theoretically analyze the performance of this process in Section 4.

3.1.2 BFS Execution Flow

The adjacency list, bitmaps, and two result arrays are all stored in the storage unit and are loaded into the processing unit for computation when needed. Based on the data structures defined, the data flow of Algorithm 1 can be implemented in hardware, as depicted in Figure 5. Within each iteration of the BFS algorithm (processing of one frontier), part of the bitmap for current frontier is loaded into the processing unit. For each marked vertex in the current frontier, its corresponding neighbors are marked in the bitmap for the next frontier.

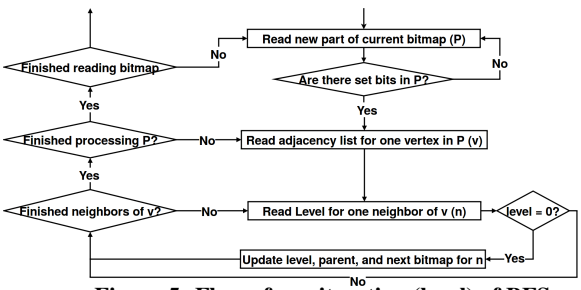

Figure 5: Flow of one iteration (level) of BFS

During the execution, BFS frequently accesses scattered locations in compact arrays stored in the HMC. This is an ideal case for us to fully exploit HMC's low random access latency and high parallelism. In addition to the significantly improved random access latency, HMC also provides native bit-level atomic updates, which is especially useful when updating the bitmap for the next frontier. Normally, updating the bitmap requires a reading of several bytes from main memory, an operation of bit updates in the processing element (set few bits to 1), and a writing back operation. Frequent bitmap modifications not only induce more traffic on the memory bus, but also are likely to result in frequent stalls when other processing elements are accessing the same address. With HMC, these unnecessary round-trip traffics can simply be avoided to save the memory bandwidth. Furthermore, by offloading the atomic operation to the HMC, the chance of stalls are considerably reduced.

3.2 FPGA-HMC based Graph Processor

In this subsection, we first introduce a Map-Reduce-like Framework to leverage the advantage of FPGA-HMC platform. Then, we present the implementation detail of proposed FPGA-HMC graph processor.

3.2.1 Map-Reduce-like Framework

Selecting a suitable execution framework is of great importance for an efficient hardware implementation. For BFS, this execution framework should be able to effectively manage irregular memory access patterns without much penalty. Additionally, as each part of the bitmap (frontier) can be processed independently in the level synchronized BFS, the framework should also be efficient in handling parallel tasks.

Based on the needs stated above, the Map-Reduce execution model is one ideal choice that well fits these descriptions. In this model, a task is divided into two phases – Mapping and Reduction. Mappers process a partition of the input data independently through a parallel streaming process. The output from mappers are then passed to reducers that produce the final results. This framework naturally offers a good degree of parallelism, making it possible to exploit random access capability of HMC by generating enough sporadic memory accesses.

Algorithm 2 depicts the Map-Reduce version of BFS. At the mapping stage, each mapper reads a partition of the bitmap, extracts the current frontier, and reads the adjacency list for these vertices. The reducers then read the level array for each neighbor, update parent and level arrays for previously unvisited vertices, and mark them by updating the bitmap for the next frontier. In the next subsection, we will convert these mappers and reducers into pipeline stages implemented on FPGA-HMC platform.

Algorithm 2 MapReduce BFS

```
1: procedure MAP(current_frontier[u:v])
2:     for i = u : v do
3:         if current_frontier[i] then
4:             for j = vertices[v] : vertices[v + 1] do
5:                 Emit (i, neighbors[j])
6: procedure REDUCE((i, [n_1, n_2, ...]))
7:     for j ∈ [n_1, n_2, ...] do
8:         if level[j] is 0 then
9:             level[j] = current_level + 1
10:            parent[j] = i
11:            next_frontier[j] = 1
```

3.2.2 Platform-aware BFS Implementation

As discussed in Section 3.2.1, we propose a Map-Reduce-like Framework to leverage the capabilities of parallel execution and

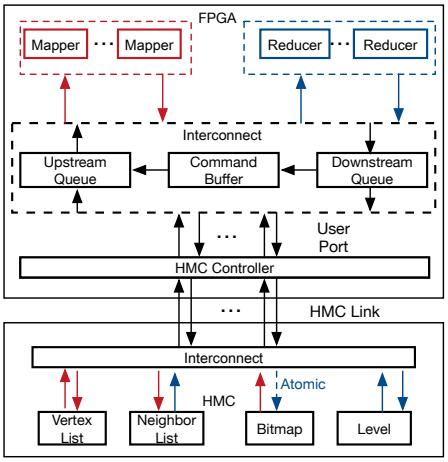

Figure 6: System diagram of FPGA-HMC based BFS implementation

the built-in atomic operation of HMC and the flexibility of FPGA. In this subsection, we will present the FPGA-HMC based BFS implementation that will be used as the basis of the analysis in Section 4. As shown in Figure 6, all mappers and reducers share the HMC Controller and HMC link via an interconnect which comprises a downstream queue, an upstream queue and a command buffer. The HMC controller is used to convert the high speed HMC link traffic into low speed user traffic. For example, a half-width HMC link (x8) will have 5 corresponding user ports, each of which could accept one downstream (to HMC) request packet and send one upstream (from HMC) response packet in one cycle. To fully utilize the bandwidth of HMC, we replicate the mapper and reducer by the same number of user ports to saturate the port resources. As there are more kernels than user ports inside the HMC controller, we add memory access queues between the BFS kernels and the HMC controller to buffer the requests that can not be served immediately. If the queue is full, the BFS kernels corresponding to the queue are stalled. In addition to memory queues, we further add a command buffer between the downstream queue and upstream buffer to log the destination of each HMC request, which decouples different kernel stages. For each HMC read request, the user needs to assign a tag to each request, which is used to keep track of the out-of-order HMC responses. The command buffer logs the tags and destination kernels of the memory accesses when sending HMC read requests, and forwards the HMC read responses to the corresponding kernels based on returned tags.

At the beginning of executing BFS, the bitmap will be reset except that the bit corresponding to the starting vertex will be set. At each cycle, the interconnect will first push the HMC access request from the mappers and reducers into the downstream queue and check 1) if HMC controller is ready to receive data, and 2) the availability of tag for HMC read request. When both conditions are satisfied, the TX interconnects will pass the memory requests to the HMC controller and store the destination in the command buffer using the tag as the effective address. Meanwhile, the interconnects will also check if there are incoming HMC responses generated at the HMC controller. If so, it will fetch the response using the tag and forward the returned data to the next destination. We will present more detailed design of mapper and reducer in section 5.3.

4. DEEPER INSIGHTS FOR PARALLEL BFS PERFORMANCE

Although we presented the data structures and execution flow of BFS that can maximize the utilization of HMC in Section 3, there are still numerous design parameters and detailed design choices that cannot be easily determined by merely examining the general characteristics of the hardware. To that end, we propose an analytical performance model and will use this model to analyze the performance of our BFS design and identify optimization opportunities.

4.1 Analytical Performance Model

As BFS is a memory-heavy algorithm (as opposed to computation -heavy), the performance of the system is generally determined by memory performance. We thus first focus our attention on modeling the memory system. In the process of developing the model, our desire is to most accurately predict the performance of HMC by capturing its unique characteristics while avoiding too much complexity. Therefore, We derive the model based on a set of observations from the HMC architecture:

1. Packets are serialized through the IO. That means at each time stamp, the IO link between the HMC and the processing unit is occupied by only one packet. The duration for which the link is occupied is proportional to the size of the packet including the data being transferred, the header, and the tail.

2. The latency of processing a packet after it was received (the internal delay) by the HMC comprises a constant delay of processing the packet header in addition to internal data transfer delay which is proportional to data size.

3. The internal delay when multiple packets are serially received by the HMC, changes depending on whether these packets access different vaults or the same vault. Naturally, parallel access to different vaults result in less latency compared to accesses with vault conflicts.

Based on these guidelines, we propose the following model for processing a single access and then extend this model to encompass more complicated situations. Equation 1 shows the latency for a read and a write operation of g bytes. In this equation, g is the packet data size, H is the packet overhead including header and tail, b and B are internal and IO bandwidths in $Bytes/s$ respectively, and t_C is the constant header processing delay. In case of a read operation, we need to account for both a request packet as well as a response packet. The request consists of only a header and a tail which takes an additional delay of $\frac{H}{B}$ to travel through the link, resulting in a difference between read and write latencies.

$$t_r = \frac{g}{b} + \frac{g+2H}{B} + t_C, \quad t_w = \frac{g}{b} + \frac{g+H}{B} + t_C \tag{1}$$

To model multiple accesses, we first consider two cases for n consecutive accesses with completely different access patterns. In the first case, all accesses are directed to the same vault. The resulting vault conflicts produce proportionally longer internal data transfer delay. On the other hand, the constant delay is hidden by overlapping processing requests. As a result, the read delay can be represented by:

$$t_r = n\frac{g}{b} + n\frac{g+2H}{B} + t_C \tag{2}$$

On the other end of the spectrum, all n accesses are directed to different vaults. In this case, packets are processed in parallel inside the HMC. Therefore, the read latency would be:

$$t_r = \frac{g}{b} + n\frac{g+2H}{B} + t_C \tag{3}$$

Based on these equations, we can now present the latency for a general case where vault conflict happens but, at the same time,

some accesses can be processed in parallel as well. The read and write latencies with such access pattern would be:

$$t_r = \alpha\frac{g}{b} + n\frac{g+2H}{B} + t_C, \quad t_w = \alpha\frac{g}{b} + n\frac{g+H}{B} + t_C \quad (4)$$

where α ($1 \leq \alpha \leq n$) represents the maximum number of vault conflicts and is inversely proportional to the number of parallel accesses.

4.2 Performance Analysis

In this section, we analyze performance of BFS by estimating its execution latency using the HMC model we developed in the previous section. Note that although we only apply it to BFS in this work, the model is generically applicable to other algorithms.

Since BFS is memory bound, we can safely assume the total latency to be that of read and write operations of HMC. For this analysis, we only derive the results for bitmap operations (scanning current bitmap and updating next bitmap). A similar approach can be used to derive estimations for other portions of runtime, but results from bitmap operations accurately represent the scaling trends of the runtime and effectively help us identify performance bottlenecks and make design decisions.

Table 2 presents the terminology used in this analysis. As this table shows, these terms are closely related. More specifically, for a connected graph, the following equations hold.

$$\sum_{l=1}^{L} Q_l = V, \quad \sum_{i=1}^{V} q_{il} = Q_{l+1}, \quad \sum_{i=1}^{V} S_{il} = Q_l \quad (5)$$

Table 2: Analysis terminology

Term	Definition
V	Number of vertices
L	Maximum number of levels for which BFS operates
Q_l	Number of set bits in $currFront$ at the beginig of level l
q_{il}	Number of neighbors vertex i visits in level l
S_{il}	Whether vertex i was visited in level $l-1$. $S_{il} : \{0, 1\}$

For convenience, we assume $Q_{V+1} = 0$, which means $\sum_{l=1}^{L} Q_{l+1} = V - 1$ (since $Q_1 = 1$). Also, for level 1 we have:

$$S_{v,1} = \begin{cases} 1, & v = v_s \\ 0, & Otherwise \end{cases} \quad (6)$$

We present our analysis in two parts. First, we analyze latency of reading bitmap for current level (current bitmap) and estimate the total amount of time spent on reading this array through the execution of BFS. Then, we do the same for updating the next frontier (the next bitmap). Although it is likely that this method results in overestimation of runtime by ignoring some overlappings of operations, it simplifies our analysis and provides a better picture of performance bottlenecks and improvement opportunities.

Reading Current Bitmap: Since the current bitmap is stored in the HMC and, for large graphs, is too large to read all at once, it has to be read in multiple partitions. We assume the number of partitions to be m and reading each partition is done by issuing several memory requests. Since we decide the order in which current bitmap is scanned, we can guarantee that requests issued to read a partition have maximum parallelism. We model this operation by k sets of n completely parallel read requests (in total we issue mkn read requests and read $D = mkng$ bytes of data). With this model, we can estimate the runtime for scanning current bitmap in level l to be:

$$\begin{aligned} T_{scan_l} &= m(k\frac{g}{b} + kn\frac{g+2H}{B} + t_C) \\ &= D\Big[\frac{1}{nb} + \frac{1}{B} + \frac{1}{g}\big[\frac{2H}{B} + \frac{t_C}{kn}\big]\Big] \end{aligned} \quad (7)$$

Here, D is determined by the graph size, n is determined by the number of vaults, and k is determined by the available on-chip BRAM. For a large graph, D is going to be large while n and k are limited by available resources. Therefore, since the scan latency is proportional to D, reading the bitmap is going to be a bottleneck of the performance if the graph is large. We will later introduce the two level bitmap to address this issue in Section 5.

We can also see from this analysis that larger values of k and g (generating more read requests with larger granularity) can reduce latency. In the case of read granularity, this is due to the read overhead which is comparatively reduced for larger requests. Increasing k reduces the execution time as well by increasing the overlap between handling requests and hiding the constant request processing latency.

Writing to Next Bitmap: As shown in line 12 in Algorithm 1, when traversing edges from a visited vertex v, its newly visited neighboring vertex n has its corresponding bit in the next bitmap set. This operation is done for all neighbors using the native atomic operation of HMC. Similar to a write request, the atomic operation does not require a response. We can estimate the latency for updating bitmap for neighbors of v ($T_{wb_{vl}}$), time spent on writing to bitmap in level l (T_{wb_l}), and the total time spent on writing to bitmap as (T_{wb}).

$$\begin{aligned} &T_{wb_{vl}} = \alpha S_{vl}\frac{g}{b} + q_{vl}\frac{g+H}{B} + S_{vl}t_C \\ \Rightarrow &T_{wb_l} = \sum_{v=1}^{V} T_{wb_{vl}} = \alpha Q_l\frac{g}{b} + Q_{l+1}\frac{g+H}{B} + Q_l t_c \\ \Rightarrow &T_{wb} = \sum_{l=1}^{L} T_{wb_l} = \alpha V\frac{g}{b} + (V-1)\frac{g+H}{B} + Vt_c \end{aligned} \quad (8)$$

This latency is dependent on g (the granularity of atomic writes), α (the average amount of access parallelism when writing to the bitmap), and V (the number of vertices). This result has a complexity of $O(V)$, which matches our expectation that this operation dominates the overall performance, as BFS in general has a complexity of $O(V + E)$ and the complexity becomes $O(V)$ for sparse graphs. In addition, since for atomic writes, g is fixed by the HMC architecture and V is a constant, the latency of this step is determined only by the amount of available write parallelism. Due to the random and data-dependent nature of memory accesses in BFS, the implementation can not adaptively change the amount of parallelism based on memory access patterns. However, it can be optimized using preprocessing with an intelligent strategy for storing the graph. We plan to investigate this optimization method in future works.

Insights from the Analyses: Using this analytical performance model, we identified the performance bottlenecks and improvement opportunities for bit-level operations of BFS. We also applied the same method to other operations performed in BFS and conducted similar analysis which generally confirms the findings from analyzing the bitmap portion of BFS. Therefore, we only discuss the results here without presenting more details.

The analysis shows how read and write granularity of accesses affects the runtime. Best read performance for reading bitmap, vertex, and adjacency analysis is achieved when larger read granularity is used. Conversely, write operations favor smaller granularity of accesses. That is because their low locality results in low access efficiency. The only exception happens when reading the level array to check whether a vertex was previously visited. Since this read also has low locality, it should be accessed with small granularity.

Another major bottleneck we identified is the scanning the bitmap during the time when the whole bitmap is read from the HMC. Usually in this data transfer process, as the graph is sparse, only a small part of it contains useful information. This problem can be

addressed by using prior knowledge about the parts of the bitmap that will be used to look for the frontier information. In Section 5, we will present a scheme to implement this optimization.

5. OPTIMIZATION SCHEME

As discussed in Section 4, the bitmap scanning becomes the bottleneck of FPGA-HMC based BFS implementation. In this section, we propose a two-level bitmap design to eliminate unnecessary HMC accesses by leveraging the sparsity of the graph. We first present the idea of the two-level bitmap, and then find the optimal bitmap granularity using the analytical model. Finally, we will present the implementation of the two-level bitmap on the hardware.

5.1 Two-level Bitmap

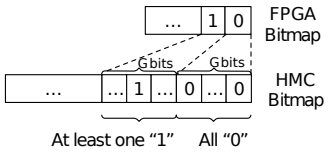

Figure 7: Illustration of two-level bitmap

In Section 4.2, it was shown that the scanning of the bitmap for current frontier creates a bottleneck for our BFS implementation. One important observation is that the bitmap is typically sparse (with regards to the placement of 1's in the whole array), resulting in a considerable amount of unnecessary data movement. Based on this observation, it is possible to take advantage of this sparsity with one level of indirection.

As shown in Figure 7, we propose a two-level bitmap scheme comprising a coarse-grained first level bitmap stored in the block ram (BRAM) on FPGA in conjunction with the fine-grained second level bitmap that we used in the baseline design. In this scheme, a block of G adjacent bits in the second level bitmap, called a range, are represented by one bit in the first level bitmap. A bit in the first level bitmap is set as long as one of the bits in the corresponding range of the second level bitmap is non-zero (each bit in the first level bitmap is the logic OR of its corresponding range in the second level bitmap). In this way, the first level bitmap can filter out reads to the second level bitmap when the bit in the first level bitmap is not set.

5.2 Bitmap Mapping Granularity

The performance of the two-level bitmap design depends on the granularity of the first level bitmap as well as the structure of the specific graph being analyzed. A more fine-grained first level bitmap provides more information about the second level bitmap, but with the trade-off of increased size. Therefore, it is important to find the lower bound for the size of on-chip bitmap that is necessary to deliver good performance. This lower bound depends on the actual structures of different graphs. A graph with fewer number of levels (higher average edges per vertex) will have less sparsity in its bitmap, resulting in a smaller lower bound. On the other hand, for a graph with a larger number of vertices and a larger off-chip bitmap, this bound should be larger. We will analytically determine this bound in a way that the performance of scanning bitmap in the two-level design is, on average, sufficiently higher than the single-level design.

To evaluate the performance improvement of the proposed two-level bitmap, we can apply the same analytical method we used in the previous section to this design. We assume the length of on-chip and off-chip bitmaps to be L_1 and L_2 bits respectively. This means that each bit in first level bitmap corresponds to $G = \frac{L_2}{L_1}$ adjacent bits in the second level bitmap. Each range, therefore, requires $k' = \frac{L_2}{8g'L_1}$ read operations, where g' is the granularity of reads in the two-level bitmap scheme.

To estimate the time spent on reading all required bitmap ranges from the HMC in one iteration of BFS, we make two assumptions. First, we have enough on-chip BRAM to store a complete off-chip bitmap range. Second, in iteration l, there are M_l set bits in the first level bitmap. In other words, M_l bitmap ranges need to be read from the HMC. Based on these assumptions, T_{scan_l} can be estimated as follows.

$$T_{scan_l} = M_l(k'\frac{g'}{b} + k'\frac{g'+2H}{B} + t_C) \tag{9}$$

This result is similar to the latency we estimated previously for scanning the bitmap in the single-level scheme. The key difference is that the number of required steps to completely scan the off-chip bitmap is reduced from m to M_l and during each step, k' requests with g' granularity are generated. Note that, when calculating T_{scan_l}, we assume that all k' accesses are serially processed by the HMC (no vault-level parallelism). However, if G was large enough, we could break ranges corresponding to each bit in the on-chip bitmap and distribute them among multiple vaults. In this way, the k' accesses required to read one range could be parallelized to reduce T_{scan_l}.

Using this result, we can guarantee that performance is, on average, sufficiently high, with a judicious choice of L_1. Next, we present an optimization method of choosing L_1 so that scanning the bitmap in the two-level bitmap design would be on average β times faster than in the single-level bitmap design. This ratio β should not only be large enough to alleviate the bitmap update bottleneck, but also result in a reasonable size of on-chip bitmap that can fit within available on-chip resources. The following equations present a condition in which the two-level design is β times faster. Intuitively, we expect a large β to require a large first-level bitmap that represents the second level bitmap in a fine-grained granularity.

$$\begin{aligned}
\beta T_{scan_l} &= \beta M_l(k'\frac{g'}{b} + k'\frac{g'+2H}{B} + t_C) \\
&< \beta M_l k'(\frac{g'}{b} + \frac{g'+2H}{B} + t_C) \\
&= \beta M_l k' T' < m(k\frac{g}{b} + kn\frac{g+2H}{B} + t_C) = T \\
&\Rightarrow M_l k' < \frac{T}{\beta T'}
\end{aligned} \tag{10}$$

Here, T and T' are known values that can be calculated based on architectural parameters of the HMC and available storage resources of the chip (k, n, b, B, etc.). As described before, we assume the number of set bits in the second level bitmap at the beginning of the l-th iteration to be Q_l. Therefore, we can find the expected number of set bits in the first level bitmap M_l based on Q_l, using statistical analysis. To find this value, equivalently, we can make an analogy to the problem where we have L_1 boxes and we randomly throw Q_l balls into these boxes with multiple occupancy allowed. To find the expected number of filled boxes at the end of this experiment, we can use the probability theory to calculate the value: $M_l = E(Q_l) = L_1(1 - (\frac{L_1-1}{L_1})^{Q_l})$ [10].

Assuming Q_l to be $\frac{V}{L}$ on average, we can use Equation 10 to find a bound for L_1.

$$M_l k' = E(\frac{V}{L})\frac{L_2}{8g'L_1} = (1 - (\frac{L_1-1}{L_1})^{\frac{V}{L}})\frac{L_2}{8g'} < \frac{T}{\beta T'} \tag{11}$$

Simplifying Equation 11 gives us a lower bound for L_1 as shown below.

$$\frac{1}{1 - (1 - \frac{8g'T}{L_2\beta T'})^{\frac{L}{V}}} < L_1 \tag{12}$$

As shown in the equation above, in addition to β, the lower bound of L_1 depends on graph characteristics and a number of hardware dependent constants. We can not know the value of L in advance, but methods of estimating this value for large graphs have been proposed in previous works [11] and can be applied here. In Equation 13 both T and L_2 increase linearly with V. Thus, $\frac{8g'T}{L_2\beta T'}$ is typically independent of the graph size and is only proportional to $\frac{1}{\beta}$. Consequently, higher speedups require larger on-chip bitmaps. This relationship confirms our intuition. The lower bound also has a direct relationship with V and an inverse relationship with L which follows our initial expectations of the relationship between L_1 and the graph structure.

Finally, recalling that $G = \frac{L_2}{L_1}$, Equation 13 can be equivalently expressed as an upper bound for G.

$$G < L_2(1 - (1 - \frac{8g'T}{L_2\beta T'})^{\frac{L}{V}}) \quad (13)$$

This equation indicates the trade-off between performance and storage space. A higher speedup, requires the bound to be smaller, resulting in a smaller G. This means that, to increase performance, the on-chip first level bitmap should represent the off-chip second level bitmap in a more fine-grained manner. Alternatively we can say, as long as G is smaller than this bound, choosing a larger value for G can achieve better space utilization without reducing performance.

5.3 Implementation of Two-level Bitmap

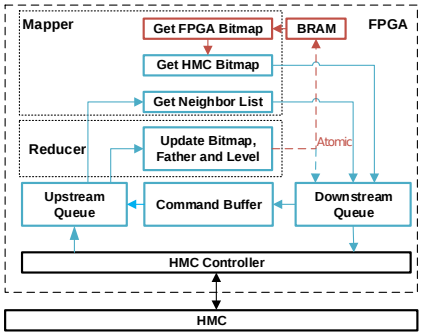

Figure 8: Detailed implementation of proposed HMC-FPGA based BFS processing system with two-level bitmap

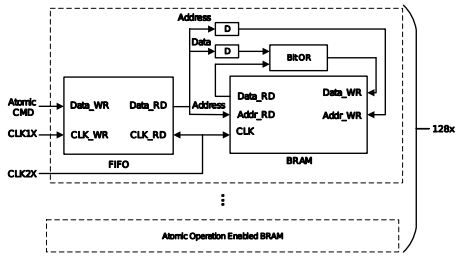

Figure 9: The implementation of atomic bitwise update operation using BRAM

To support the proposed two-level bitmap, we need to extend the FPGA implementation based on single level bitmap presented in section 3.2, which has a mapper design comprising two pipeline stages: getting the bitmap from HMC and getting the neighbor list from HMC. As shown in Figure 8, we add the following three components to the baseline design described in section 3.2.2: 1) BRAM for FPGA bitmap storage; 2) a third pipeline stage of mapper for scanning the FPGA bitmap; 3) supporting atomic updates for FPGA bitmap. In each BFS level, mappers first scan the FPGA bitmap to find the asserted bit in the FPGA bitmap and read the corresponding HMC bitmap. Reducers need to update both FPGA bitmap and HMC bitmap atomically.

As shown in Figure 9, to support the atomic bitmap updates, we need to first read the memory content, conduct a bit-wise "OR" operation with the input, and then write it back to the BRAM. As the atomic read-modify-write procedure requires two cycles, and our kernel runs at a relatively low frequency, we use a double pump BRAM to reduce the latency of atomic operations to one kernel clock cycle. Since now the BRAMs and Map-Reduce BFS kernels are in different clock domains, we further add a FIFO between the kernel and the BRAM. The FIFO also buffers the atomic bitmap update commands when BRAM conflicts happen. To provide enough parallelism of the bitmap scan as well as to reduce the BRAM conflict of atomic bitmap update, we use 128 BRAM blocks to store the the on-chip bitmap. We use lower bits of the bitmap address as the byte address, and higher bits of the bitmap address as the BRAM address.

6. EVALUATION

In this section, we first introduce the experimental setup. Then, we present the simulation and experiment results to validate the effectiveness of design choices using proposed techniques. Finally, we show performance comparison between our results and prior works.

6.1 Experimental Setup

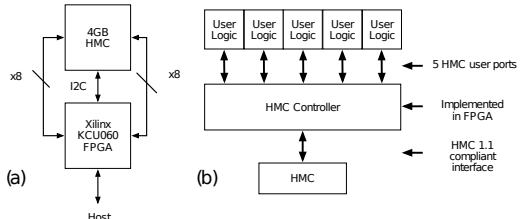

Figure 10: (a) Micron AC-510 board with two half-width HMC links [12] (b) HMC controller diagram [13]

We implement the proposed graph processor on an AC-510 FPGA module from Micron. As shown in Figure 10, AC-510 consists of a Xilinx KCU060 FPGA and a 4GB HMC chip. The AC510 board uses two half-width (8 lanes) 15G HMC links to connect HMC and FPGA, and provides an overall two-way bandwidth of 60GB/s. We implement our graph processing architecture under the PicoFramework, which provides communication between the host and the FPGA kernel. We use the HMC controller IP core from Micron as the interfaces between the FPGA kernel and the HMC. The host machine equips an Intel Xeon E5-1630V3 CPU and one DDR4 memory channel with 16GB capacity. We use Ubuntu 16.04.1 as the host operating system and compile our CPU implementation using gcc with flags "-Ofast" and "-march=native".

To accelerate the development process and facilitate evaluation of optimization methods, we develop an event-based HMC simulator. Using this simulator, we can gain better insights into the internal mechanisms of the HMC and avoid tedious trial-and-error cycles. Here, we discuss the details of our simulator and present simulation results for our experiments to show the improvements we can achieve using our optimization methods.

The HMC simulator is developed based on the analytical model presented in previous sections. Compared to those cycle-accurate simulators, this simulator sacrifices accuracy for better simulation speed and thus is more suitable for large workloads. Using this simulator, we can produce performance and event statistics for graphs with millions of nodes.

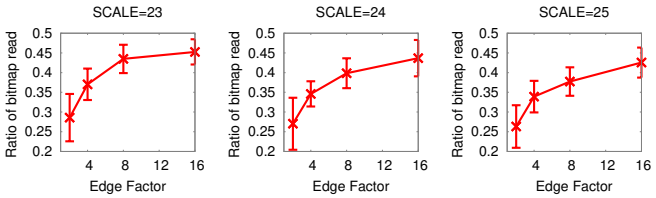

Figure 11: The ratio of HMC read request of bitmap scanning between two-level bitmap and single-level bitmap designs with different graph scale and edge factors

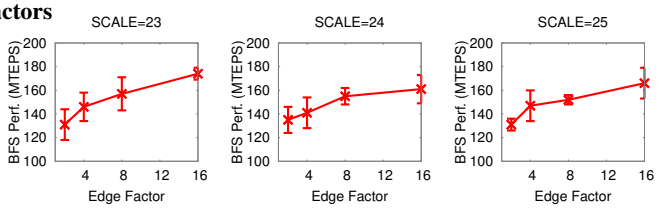

Figure 12: BFS performance of two-level bitmap design for different graph scales and edge factors

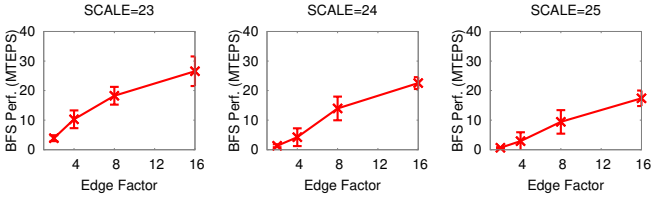

Figure 13: BFS performance of single-level bitmap design with different graph scales and edge factors

We use a similar method as stated in Graph 500 [1] to generate random graphs for testing our design. These graphs are generated with two tunable parameters, a scale (the number of vertices) and an edge factor (the ratio between total number of edges and total number of vertices). In other words, edge factor determines the average number of neighbors each vertex possesses. A larger edge factor results in a more connected graph. In the case of BFS, this means the algorithm would have to run for fewer numbers of iterations (L is smaller).

6.2 Results

We first use our event-driven simulator to verify the effectiveness of our two-level bitmap. A series of large sparse graphs with a scale of 23, 24, and 25 and different edge factors are generated. We plot the ratio of the number of bitmap reads between the two-level bitmap and the single-level bitmap scheme. As shown in Figure 11, the two-level bitmap scheme has consistently better performance (less reads). The sparser the graph is, the more effective this scheme can filter out unnecessary reads and the better the performance becomes. This trend also holds as the graph becomes larger. In Figure 13 and Figure 12, we further plot the BFS performance comparison between the single-level and the two-level bitmap schemes. It can be observed that the two-level bitmap leads to greater BFS performance gain on a sparser and larger graph. In contrast, due to the long off-chip latency and excessive reads generated in the single level bitmap scheme, the reference setup cannot saturate FPGA kernel resources by wasting a large portion of runtime waiting for the bitmap reads to be served.

We run a series of random access benchmarks from the Picoframework to evaluate the HMC access performance. As shown in Figure 14, we plot the traffic for four different cases: 100% READ, 100 % Write, Read-Modify-Write and Atomic Write with different payload sizes. If the size of the data payload is halved, the performance of random access does not double due to the overhead of packet head and tail. The results here confirmed our previous assumption that using larger payload size is suitable if the larger payload contains all useful data, which will be used by the kernels . Then, we plot memory access performance of our BFS implementation in Figure 15. We can see that our BFS implementation

Table 3: Performance comparison with existing works

System	Proposed	FPGP [4]	GRAPH Gen [3]	Torous-GRAPH [14]
Graph Type	Random	Twitter [15]	Twitter	Random
Max. Scale	26	25	26	22
Edge Factor	16	35	16	16
Runtime (s)	3.851	121.992	148,577	76.134
MTEPS	166.2	12.0	9.9	19.2

Table 4: Runtime comparison between single level bitmap, two-level bitmap and CPU (Scale=25, Edge Factor=16)

	Two-Level Bitmap	Single-level Bitmap	CPU
Runtime (s)	3.851	30.976	13.84
MTEPS	166.2	20.6	46.2

Table 5: Resource utilization

	FF	BRAM	DSP
Total	663360	2160	2760
Used	221894	580	64
Utilization	33%	27%	2%

could achieve the same memory access performance as the random access performance, which indicates that we have nearly saturated the HMC I/O.

In Figure 16, we further show that the BFS performance in the unit of million traversed edges per second (MTEPS) with different payload sizes. It can be seen that it has a similar trend as the ones shown in Figure 14, which indicates that the performance of our BFS implementation is largely determined by the random access speed. As long as the memory bandwidth can be further increased (i.e. a board with more high-bandwidth HMC links/lanes), we can achieve better BFS performance.

We compare the runtime of a CPU (Xeon E5-1630v3) and the proposed HMC-FPGA platform on a GRAPH500 graph with a scale of 26 and an edge factor of 16. Table 4 shows that our implementation achieves 3× performance compared with CPU. The two-level

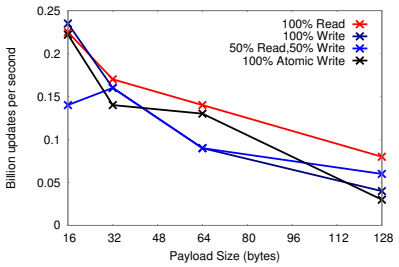

Figure 14: Benchmark of random access performance of HMC

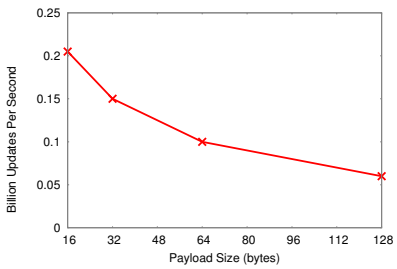

Figure 15: Memory access performance of BFS

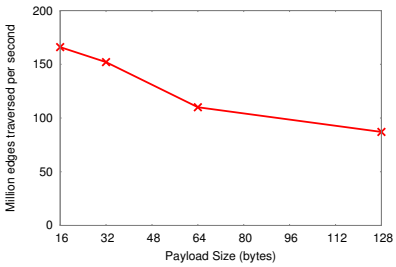

Figure 16: BFS performance with different payload size

bitmap scheme considerably boosts the performance of the proposed system by filtering out unnecessary reads and saving memory bandwidth.

In Table 3, we further compare the results with existing works. Due to the limited capacity of our HMC chip, we can only process a sparser graph but with the similar scale as the Twitter Graph [15] that is used in three prior works. Our implementation outperforms the prior works by nearly *one order of magnitude* and proves the effectiveness of proposed FPGA-HMC based graph processing system. Furthermore, as our previous analysis shows that our implementation tends to have better BFS performance with denser graphs, we expect to have much more performance gain (more than one order of magnitude) if using exactly the same Twitter Graph as the data input to the benchmark.

Finally, we show the resource utilization of our implementation in Table 5. As we store the full bitmap on HMC instead of on-chip BRAM, we only use 27% of the total 18Kb BRAM. This provides enough room for expansion if we have a faster and wider HMC link.

7. RELATED WORKS

There are several existing works on implementing graph accelerator using FPGA. GRAPHGEN [16] proposed an FPGA-based graph processing system using vertex centric model. However, it stores the whole graph in the on-board DDR DRAM, which severely limits the performance due to the bandwidth bottleneck of the memory. Also, the design does not provide any platform-aware software and/or hardware optimization for implementing BFS. TorusBFS [6] proposed a 2-D message passing structure to reduce the latency between parallel BFS kernels, but its performance is also limited by the poor random access performance of DRAM and the available on-chip resources. FPGP [4] employed interval-shared structure to maximize the the off-chip memory bandwidth and to fully exploit the parallelism of graph processing. However, its performance is in turn bounded by the capacity and bandwidth of FPGA's on-chip memory.

8. CONCLUSION

In this work, we present a graph processor design to fully exploit the capability of FPGA and HMC through collaborative software and hardware techniques. In particular, we first present the data structure and algorithm modifications, followed by Map-Reduce implementation of level synchronized BFS on FPGA-HMC platform. To gain deeper insights into the performance bottlenecks, we develop an analytical model for BFS runtime with respect to the HMC parameters and the graph properties. We found that the number of bitmap reads contributes a significant portion of the memory accesses and thus becomes the key performance limiting factor. To address the problem, we further introduce a two-level bitmap scheme, which leverages the sparsity of the bitmap and reduces the number of HMC accesses significantly. Finally, we use both simulation and experiment to verify the effectiveness of proposed techniques. Our implementation on Micron AC-510 development board achieves 166 MTEPS and outperforms CPU and other FPGA-based large graph processors.

ACKNOWLEDGEMENTS

We appreciate the insightful comments and feedback from the anonymous reviewers. We thank Micron for the donation of the development tool and hardware. We especially thank John Watson and Mark Hur for their support.

9. REFERENCES

[1] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray UserâĂŹs Group*, 2010.

[2] J. T. Pawlowski, "Hybrid memory cube (hmc)," in *IEEE Hot Chips*, 2011.

[3] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," in *USENIX OSDI*, 2012.

[4] G. Dai, Y. Chi, Y. Wang, and H. Yang, "Fpgp: Graph processing framework on fpga a case study of breadth-first search," in *ACM/SIGDA FPGA*, FPGA '16, 2016.

[5] Y. Umuroglu, D. Morrison, and M. Jahre, "Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform," in *IEEE FPL*, 2015.

[6] G. LEI, R. LI, S. GUO, and F. XIA, "Torusbfs: A novel message-passing parallel breadth-first search architecture on fpgas," 10 2016.

[7] P. Rosenfeld, "Performance exploration of the hybrid memory cube," 2014.

[8] S. Kaur, S. Singh, and S. Kaushal, "Performance comparison of sampling techniques for web-based networks," in *International Conference on Recent Advances in Engineering Computational Sciences*, 2015.

[9] Y. S. Deng, B. D. Wang, and S. Mu, "Taming irregular eda applications on gpus," in *ACM ICCAD*, 2009.

[10] M. Mitzenmacher and E. Upfal, *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[11] L. Gulyás, G. Horváth, T. Cséri, and G. Kampis, "An estimation of the shortest and largest average path length in graphs of given density," *arXiv preprint*, 2011.

[12] Picocomputing, "Ultrascale-based superprocessor with hybrid memory cube." http://picocomputing.com/ac-510-superprocessor-module.

[13] Picocomputing, "Hybrid memory cube (hmc) and controller ip." http://picocomputing.com/hybrid-memory-cube-hmc-controller-ip/.

[14] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *ACM SIGKDD*, ACM, 2013.

[15] J. Yang and J. Leskovec, "Patterns of toral variation in online media," in *Proc. of the fourth ACM international conference on Web search and data mining*, 2011.

[16] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *IEEE FCCM*, 2014.