electronics

MDPI

Article

CGAcc: A Compressed Sparse Row Representation-Based BFS Graph Traversal Accelerator on Hybrid Memory Cube

Cheng Qian [1], Bruce Childers [2], Libo Huang [1,*], Hui Guo [1] and Zhiying Wang [1]

[1] College of Computer, National University of Defense Technology, Changsha 410073, China; mengnanpeter@gmail.com (C.Q.); huiguo@nudt.edu.cn (H.G.); zywang@nudt.edu.cn (Z.W.)
[2] College of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA; childers@cs.pitt.edu
* Correspondence: libohuang@nudt.edu.cn

Received: 1 October 2018; Accepted: 2 November 2018; Published: 7 November 2018

Abstract: Graph traversal is widely used in map routing, social network analysis, causal discovery and many more applications. Because it is a memory-bound process, graph traversal puts significant pressure on the memory subsystem. Due to poor spatial locality and the increasing size of today's datasets, graph traversal consumes an ever-larger part of application execution time. One way to mitigate this cost is memory prefetching, which issues requests from the processor to the memory in anticipation of needing certain data. However, traditional prefetching does not work well for graph traversal due to data dependencies, the parallel nature of graphs and the need to move vast amounts of data from memory to the caches. In this paper, we propose a compressed sparse row representation-based graph accelerator on the Hybrid Memory Cube (HMC), called CGAcc. CGAcc combines Compressed Sparse Row (CSR) graph representation with in-memory prefetching and processing to improve the performance of graph traversal. Our approach integrates the prefetching and processing in the logic layer of a 3D stacked Dynamic Random-Access Memory (DRAM) architecture, based on Micron's HMC. We selected HMC to implement CGAcc because it can provide quite high bandwidth and low access latency. Furthermore, this device has multiple DRAM layers connected to internal logic to control memory access and perform rudimentary computation. Using the CSR representation, CGAcc deploys prefetchers in the HMC to exploit the short transaction latency between the logic and DRAM layers. By doing this, it can also avoid large data movement costs. In the runtime, CGAcc pipelines the prefetching to fetch data from DRAM arrays to improve memory-level parallelism. To further reduce the access latency, several optimized internal caches are also introduced to hold the prefetched data to be Processed In-Memory (PIM). A comprehensive evaluation shows the effectiveness of CGAcc. Experimental results showed that, compared to a conventional HMC main memory equipped with a stream prefetcher, CGAcc achieved an average 3.51× speedup with moderate hardware cost.

Keywords: compressed sparse row; graph traversal; hybrid memory cube

1. Introduction

We now live in the Big Data era, and thus, today's memory-bound applications put even more pressure on the memory system than before, forcing memory techniques to advance and new approaches to be adopted. Many new memory systems have recently been developed, such as Wide-I/0, fully-buffered DIMM, buffer-on-board, High-Bandwidth Memory (HBM) and the Hybrid Memory Cube (HMC) [1]. These memory systems significantly improve on conventional DDR4 memory. For example, the latest generation of HMC can provide 480 GB/s of peak bandwidth, which

is almost 20× greater than DDR4. Applications with regular access patterns can gain substantial benefit from these new technologies, especially when combined with prefetching and deeply parallel memory architectures.

However, if an application shows an irregular access pattern, the prefetch technique and parallel architecture will not be able to substantially improve the performance of such applications. This also means that such applications may not enjoy the benefits of high bandwidth, which the prefetch technique and parallel architecture rely on. These applications do not work very well due to their unpredictable access patterns, poor spatial locality and sometimes data-dependent accesses. Graph traversal is such an application that has an irregular access pattern. However, it is widely used in many situations, such as road routing, analyzing social networks to find relationships, analyzing gene graphs, detecting causal relationships, and so on. Due to the unique property of graphs, graph traversal is quite time consuming, especially when a graph has a massive number of vertexes and edges. In fact, the size of a real-world graph can be very huge. For example, the California road network has nearly two million vertexes. In Section 2.4, we show that graph traversal suffers from a high memory stall ratio and a large cache miss rate.

Prefetching is a conventional way to accelerate memory-bound workloads. This technique can efficiently reduce the high memory transaction latency by learning an application's access pattern to maintain a relatively high accuracy to predict and fetch the next likely accessed data. Unfortunately, the access pattern for a graph is typically irregular, data-dependent and non-sequential. This conclusion has already been testified in many previous works [2,3]. The memory behavior of DFS-like traversal has something in common with BFS-like traversal, but great difference exists as well. The common point is that the access pattern in both BFS-like traversal and DFS-like traversal is indirect, which means the next access address can be known only if the current data have been fetched. The difference is that BFS-like traversal can confirm the following K vertexes when the current vertex and its expanded edges are confirmed, while DFS-like traversal can only confirm the next vertex when the current vertex and one of its expanded edges are confirmed. It has been said that it is very difficult to implement DFS at the hardware level, because a stack structure is needed to keep metadata during the DFS procedure [4]. In comparison, the acceleration of BFS-like traversal is more promising. This paper aims to provide support for the acceleration of this BFS-like indirect memory access pattern, which can also be generalized to other similar access patterns shown in Section 3.4. Thus, applications that rely on graphs are very challenging for conventional prefetchers to predict the access pattern accurately. Several classical prefetchers, such as stride and pointer prefetch, are reported to be inefficient for graph traversal [3,5]. It is also difficult to use special-purpose devices (e.g., GPU) to accelerate memory access because graph processing cannot be easily parallelized [6]. In recent years, many works have tried to customize optimized distributed frameworks (MapReduce, Spark) for parallel graph processing problems, such as Distributed GraphLab [7], GraphX [8], JMGPA [9], DPM [10], and so on. These works try to build a special-purpose distributed framework to accelerate graph processing at the software level and have been reported to achieve great speedup. However, these methods cannot take full advantage of hardware at the software level.

Fortunately, graph traversal itself has a simple and well-defined principle that offers hope for prefetching. This principle can be described as follows: the memory controller will use the current fetched data as a memory address and then fetch the data that are located at that address. Thus, based on knowing where to locate the data used in the near future, we can try to fetch these data as early as possible. By integrating a logic layer under several Dynamic Random-Access Memory (DRAM) layers, 3D stacked memory makes it possible for the memory system to undertake some computational tasks rather than just being treated as a storage device. In this paper, we use Micron's Hybrid Memory Cube (HMC) to design an accelerator for graph traversal. The current HMC specification [1] provides for a high-bandwidth main memory with integrated logic to handle simple atomic commands and control memory access. These simple atomic commands contain some arithmetic operations, logic operations, bitwise operations, and so on. Some operations have unique limitations. An atomic

operation command incurs read, update and write operations, and these operations happen atomically, which means that subsequent commands to the same bank are scheduled after the writing of the atomic command is complete. For example, for a two-operator add operation, one of the operators should be constant. Adding to this baseline design, we propose CGAcc, a compressed sparse row representation graph traversal accelerator on HMC. CGAcc deploys three prefetchers on HMC's logic layer, and these prefetchers are arranged as a pipeline to reduce the transaction latency cooperatively. We use the Compressed Sparse Row (CSR) because it is the de facto representation for sparse graphs, which applies for almost all realistic large-scale graph applications. Some prior work has shown the efficiency of CSR compared to other formats. In comparison to the standard HMC, our evaluations show that CGAcc achieved an average 3.51× speedup (up to 7.4× speedup) with modest hardware cost.

There are many previous works about the optimization of graph processing. Before the wide use of compressed sparse row representation, some works focused on the pointer prefetching because the linked structure is the base data structure in graphs. Some typical work (e.g., content-directed prefetching [5], jump-pointer prefetching [11], and so on [12,13]) is based on pointer prefetching. Limited by the graph representation, the speedup improvements of these works were moderate. Recent works mainly focused on the CPU side, targeted the conventional memory architecture, incorporated a newly-optimized prefetcher in the CPU cache and treated memory as a slave that receives memory accesses passively. Examples of such kinds of works include the Indirect Memory Prefetcher (IMP) [14], PrefEdge [15], the explicit graph prefetcher [2], Minnow [16], and so on. Generally, the key principle of all of these works is to use spare time for prefetching the data in advance, which is different from our design. While this solution is mainstream for conventional memory, it faces problems such as over-fetching and cache pollution, imbalance between prefetch access and common memory accesses, expensive data movement cost, and so on. Inspired by the novel three-dimensional memory architecture, HMC is also used as an accelerator to implement optimization for some specific scenarios. For instance, neurocube [17] is a programmable and scalable digital neuromorphic architecture included in the logic layer of HMC. It can effectively accelerate some neural network applications. Other similar work (e.g., GraphH [18], HMCSP [19]) treated the HMC as a co-processor that can undertake parts of computational tasks. CGAcc treats the HMC as a smart master that takes on active responsibility of the graph traversal. In addition, compared to these works, CGAcc concentrates on CSR-based graph representation and can obtain good speedup with moderate cost. Some hardware-level works aim at designing a specific architecture for graph processing using an FPGA (Field-Programmable Gate Array), NVM (Non-Volatile Memory), and so on. Examples of these kinds of works include OmniGraph [20], GraphR [21], and so on. These hardware-level works need extensional devices for acceleration; thus, the overhead is larger than CGAcc because CGAcc is deployed in HMC, and HMC can be treated as the main memory system in a computer system. Some software-level works optimized graph processing by enriching the instruction set architecture [22] or customizing the compiler [23]. Software-level works cannot make full use of hardware. Besides, complex management frameworks or strategies are always needed to make the design work well, which is also very costly.

This paper makes the following contributions:

1. We characterize the performance bottleneck of graph traversal, analyze the benefit of using 3D stacked memory for traversal and motivate the design of CGAcc. In our approach, the memory system is as an active partner rather than a passive or co-processing device as in most previous works.
2. We propose CGAcc, a CSR-based graph traversal accelerator for the HMC. This design is based on knowledge of the workflow and structure of graph traversal. CGAcc augments the HMC's logic layer with prefetching, which operates in a pipeline to reduce transaction latency and data movement cost.

3. We evaluate CGAcc under a variety of conditions to consider several design trade-offs. The experimental results demonstrate that CGAcc offers excellent performance improvement with modest hardware cost.

The rest of the paper is organized as follows. Section 2 presents background information about the HMC, CSR-based graph traversal and conventional prefetching. We also describe the problem of graph traversal for conventional memory and illustrate the opportunity for optimization. Section 3 presents the design of CGAcc. Sections 4 and 5 describe our experimental methodology and evaluation results. Sections 6 and 7 summarize the related work and the conclusions.

2. Background

2.1. HMC Overview

The Hybrid Memory Cube (HMC) is a 3D stacked memory architecture based on DRAM. Micron Corp. announced the HMC in 2011, targeting an improvement in performance by 15× compared with DDR3. The first specification for the HMC was published in 2013. It used 16-lane or eight-lane full duplex differential serial links, with each lane having 10, 12.5 or 15 Gb/s SerDes. The second version of HMC was released in 2014. In this version, the HMC has more choices for SerDes rates that range from 12.5 Gb/s–30 Gb/s, leading to an aggregate link peak bandwidth of 480 GB/s. Up to eight HMC packages can be chained together with cube-to-cube links to provide even higher bandwidth. The newest HMC supports four SerDes links connected to an HMC controller. In an HMC system, a packet-based protocol is used to transmit memory commands and data. Figure 1a illustrates the requester-responder pair to handle a packet transaction.

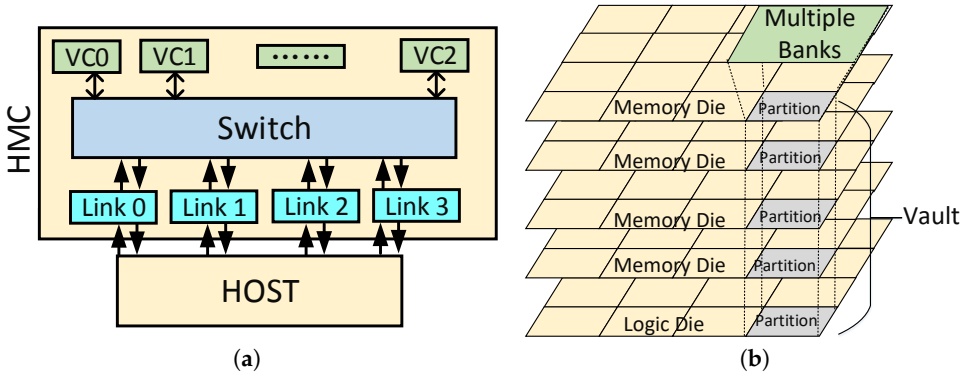


Figure 1. Overview and structure of the Hybrid Memory Cube (HMC). DRAM: Dynamic Random-Access Memory; TSV: Through-Silicon Via. (**a**) Overview of the HMC. (**b**) Structure of the HMC.

The latest HMC's structure is shown in Figure 1b. Generally, four or eight DRAM dies are stacked on top of a single logic die. Through-Silicon Vias (TSVs) are used to connect these dies, or layers. The logic layer can provide multiple functions, such as data refresh, error correction, data routing and DRAM sequencing. Some simple, but useful processing in memory is also included, which offers an opportunity to further embed processing in the logic layer. Within an HMC, portions of dies from the same physical locations are combined to form a vertical sub-structure, called a vault. Each vault has its own controller on the logic layer to maximize memory-level parallelism. A vault has several banks, and each bank has a fixed 16-MB capacity.

2.2. Graph Traversal with CSR

To reduce capacity cost and improve storage efficiency, CSR representation is widely used to represent graphs [24]. For a CSR-based graph, three arrays (i.e., `vertex`, `edge`, `visited`) are used.

These arrays hold indexes rather than pointers. Figure 2 and the code below show the workflow and pseudocode of a CSR-based graph traversal example, respectively. In Figure 2, the notation such as $a - b$ means an edge from vertex a to vertex b. In the parameter list of the pseudocode, WorkList, Vertex_List, Edge_List and Visitedrepresent the entry of each array. Vertex_cnt and Edge_cnt refer to the number of vertexes and edges, while *Root* means the start point of the graph traversal procedure. The index of a work vertex leads to the corresponding two locations ($Index$ and $Index + 1$) in the vertex array. These two values, which are fetched from vertex arrays, illustrate the range that the data should take from the edge array. Similarly, the edge data will be used as the index for the visited array. Finally, the visited array will be accessed to determine whether this vertex has been visited, and if not, the vertex will be pushed into the work list as a new vertex. Note that although we illustrate a directed graph example, CGAcc supports all kinds of graphs, regardless of whether they are acyclic, directed or undirected. This is because CGAcc is such a design that focuses on optimizing the current hardware system. There is no difference in handling different kinds of graphs from the viewpoint of the hardware. HMC just needs to handle read/write operations and simple arithmetic operations such as address calculation. Part of the timeline for traversing this sample graph is compared between using a conventional memory and CGAcc in the following sections.

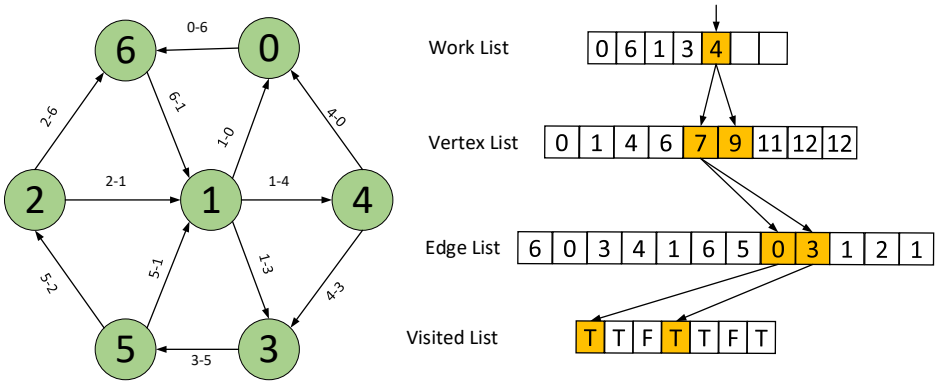


Figure 2. Compressed sparse row-based graph traversal overflow. The notation such as $a - b$ means an edge from vertex a to vertex b. Part of the timeline for traversing this sample graph is compared between using conventional memory and CGAcc in the following sections.

2.3. Conventional Prefetching Techniques

Stride prefetch: Stride prefetch is a classical technique that prefetches data according to a pre-defined distance relative to the current memory address. Stride prefetch works well for sequential access patterns such as arrays and matrices. However, graph traversal is irregular and data-dependent, which causes stride prefetch to be inefficient. Stride prefetch can even cause performance degradation. Previous work reported that using two distinct prefetchers in the L1 and L2 cache yielded only a 17% performance speedup [2].

CSR-BASED GRAPH TRAVERSAL
$(WorkList, Vertex\_List, Edge\_List, Visited, Vertex\_cnt, Edge\_cnt, Root)$

```
1  for Cur_vertex = Root to Vertex_cnt
2      if Visited[Cur_vertex] == True
3          Continue
4      Vertex.push(Cur_vertex)
5      Visited[Cur_vertex] = True
6      while Vertex.Empty() == False
7          Vertex.Index = Vertex.top()
8          Vertex.pop()
9          Edge_Start = Vertex_List[Vertex_Index]
10         Edge_End = Vertex_List[Vertex_Index + 1]
11         for Edge_Index = Edge_Start to Edge_End
12             Visited_Index = Edge_List[Edge_Index]
13             if Visited[Visted_Index] = False
14                 Vertex.push(Visited_Index)
```

Stream prefetch: Stream prefetching works based on access pattern history. The CPU profiles several access addresses (known as "streams") during a time window. Memory accesses with similar patterns can issue a prefetch instruction according to the saved stream. This scheme works well because memory accesses tend to have a re-occurring pattern, and thus, the technique primarily relies on spatial locality. Figure 3 shows a performance comparison between a CPU and cache without/with stream prefetch. We deployed a stream prefetcher on the L2 cache to evaluate the speedup. The graph shows that stream prefetch had a small speedup of 6.5% in the best case. Although the benefit of stream prefetching for graph traversal is limited, it outperforms stride prefetching by about 5%, according to a previous work [2]. According to these results, in our experiments, we used a two-level memory system armed with a stream prefetcher as a state-of-the-art baseline.

Software prefetch: Many programming frameworks provide support for prefetching through annotations (programmer or compiler) that mark critical code sections suitable for prefetch. For graph traversal, software prefetching has minimal improvement. According to Figure 3, any read (load) accesses to obtain data will cause memory stalls. Like any other prefetch scheme, inaccurate software prefetching can incur too many additional memory accesses, which cause contention for memory resources or pollute the cache.

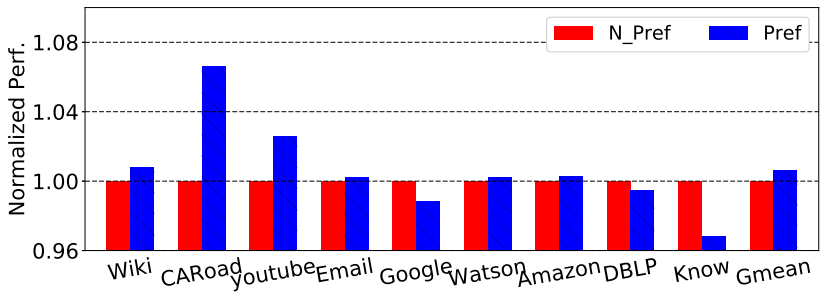


Figure 3. Speedup of graph traversal benchmarks without prefetch (`N_Pref`) and with stream prefetch (`Pref`).

2.4. Bottleneck in Graph Traversal

Graph traversal suffers from memory stalls and cache misses due to limited spatial locality. Spatial locality for traversal only exists in certain edge lists for a vertex. We evaluated graph traversal on nine graph benchmarks from the SNAP dataset [25]. Detailed descriptions of these workloads are shown in Table 1. Figure 4 shows the stall ratio traversal for these graphs. The figure shows that nearly all of the

benchmarks had a high stall rate, which approaches 90% on average. Here, we especially illustrate the comparison of Nstall rate between the situation without a prefetcher and with a prefetcher, as Figure 5 shows. From this figure, we can conclude that the prefetcher provided almost no benefit to the Nstall rate because the Nstall ratio difference was less than 0.1%. Figure 6 shows the high L1 miss rate of the benchmarks as well. On average, the L1 miss rate was 58.9%. With stream prefetching, there was only a 7.2% improvement. According to prior work, accessing the edge array accounts for the majority of the misses [2]. In most cases, the edge array is several times larger than the visited and vertex arrays, which contributes to most of the misses. The miss rate is still high for the visited and vertex arrays because accesses to these arrays happen in a data-dependent order. In turn, the high cache miss rate for graph traversal directly leads to poor performance.

Table 1. Benchmark description.

Workload	Vertex	Edge	Description
Wiki	2,394,385	5,021,410	Wikipedia talk (communication) network
CA_Road	1,965,206	2,766,607	Road network of California
YouTube	1,134,890	2,987,624	YouTube online social network
Email	265,214	420,045	Email network from an EU research institution
Google	875,713	5,105,039	Web graph from Google
Watson	2,041,302	12,203,772	Watson gene graph
Amazon	262,111	1,234,877	Amazon product co-purchasing network from 2 March 2003
DBLP	317,080	1,049,866	DBLP collaboration network
Knowledge	138,612	1,394,826	Knowledge graph

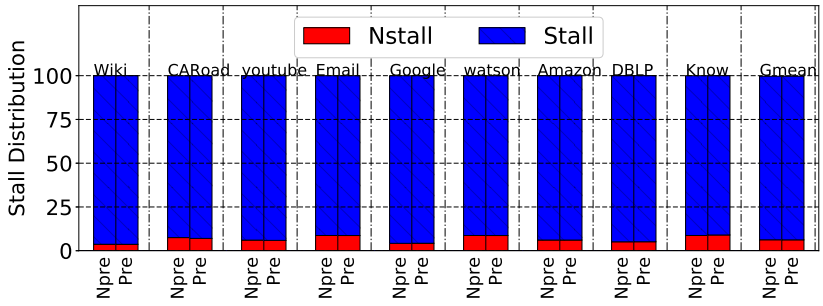


Figure 4. Stall ratio of graph traversal benchmarks without prefetch (`N_Pref`) and with stream prefetch (`Pref`).

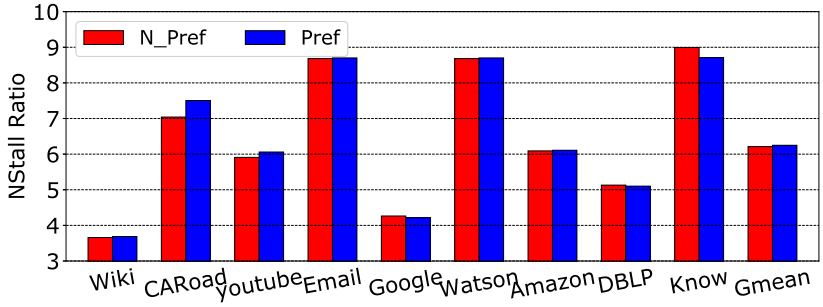


Figure 5. NStall ratio of graph traversal benchmarks without prefetch (`N_Pref`) and with stream prefetch (`Pref`).

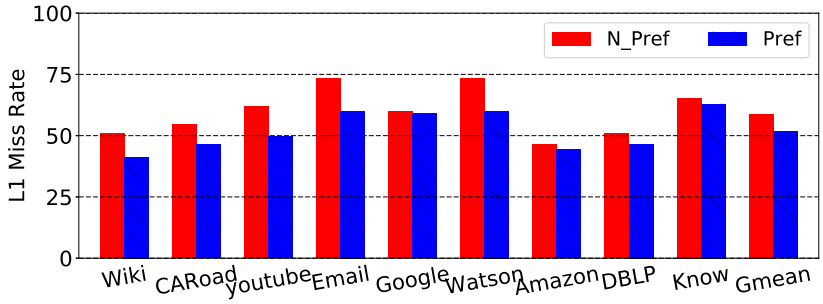


Figure 6. L1 miss rate graph traversal benchmarks without prefetch (`N_Pref`) and with stream prefetch (`Pref`).

3. Architecture

We present an architecture called CGAcc that accelerates CSR-based graph traversal using 3D stacked DRAM. Figure 7 gives an overview of CGAcc. As noted earlier, in this paper, we use Micron's HMC, and CGAcc is integrated into the HMC's logic layer. CGAcc incorporates several structures, including registers, buffers and control. Memory accesses for graph traversal can be divided into three parts, according to the target arrays (i.e., vertex, edge and visited). Accesses that are targeted to different graph vertexes are data-independent and inherently pipelineable. CGAcc processes accesses with separate prefetchers in a pipeline to hide transaction latency. In addition, it incorporates an internal cache to further reduce transaction latency.

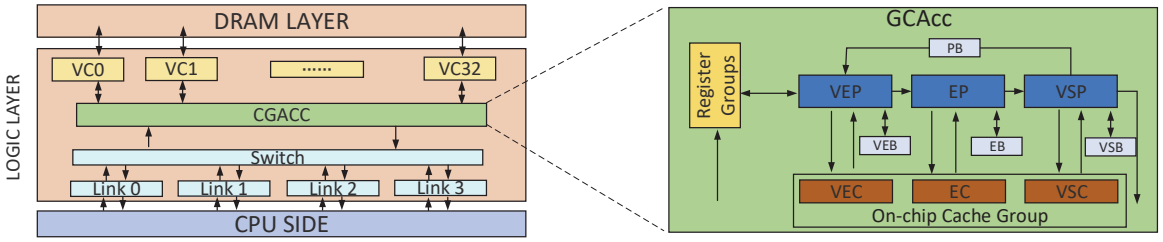


Figure 7. Overview of CGAcc. EB: Edge Buffer; EP: Edge Prefetcher; PB: Prefetch Buffer; VB: Vertex Buffer; VEP: Vertex Prefetcher; VSB: Visited Buffer; VSP: Visited Prefetcher; EC: Edge prefetch Cache; VSC: Visited prefetch Cache.

Figure 2 shows an example of graph traversal, and Figure 8 compares a part of the timeline for traversing this sample graph by using a conventional memory and CGAcc. The comparison shows that memory accesses can be overlapped with CGAcc. For instance, the accesses to different graph vertexes in time slots t3 and t4 can be prefetched independently with CGAcc, but must be done sequentially for a conventional memory.

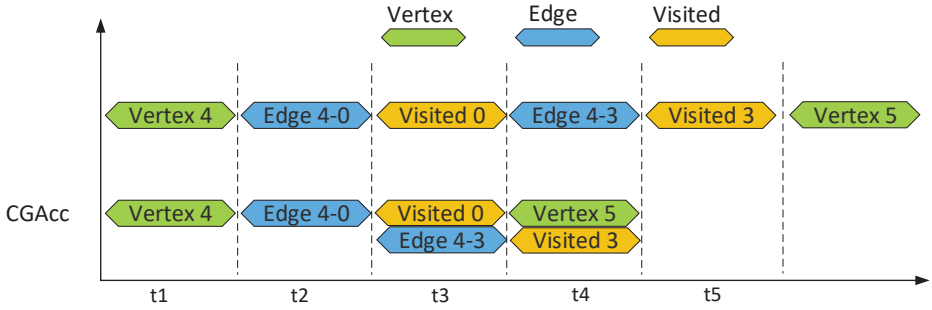


Figure 8. Example timeline of steps for graph traversal. This figure compares a part of the timeline for traversing the sample graph in Figure 2 by using a conventional memory and CGAcc.

To achieve the overlap, CGAcc does a graph traversal on behalf of the CPU. The CPU sets up and hands off a traversal to CGAcc. By using the internal computational components, CGAcc does all address computations and memory operations on the memory side to walk through the graph in a pipeline. This computation is a simple form of Processing In-Memory (PIM) tailored to graphs. The PIM capability avoids unnecessary memory transfers to/from the CPU and the memory and avoids polluting the CPU cache with the traversal. Most critically, CGAcc can exploit the internal parallelism and high bandwidth of the HMC to traverse a graph without involvement of the CPU. Thus, CGAcc overcomes the sequentiality and inefficient way of handling memory accesses by a conventional memory and improves the memory performance of a traversal. It is worth noting that CGAcc is a pure hardware architecture-level design, which means it is totally transparent to the operating system and the programmers. Neither the operating system nor software stack are involved. Figure 9 shows the workflow after a programmer calls a function to begin graph traversal.

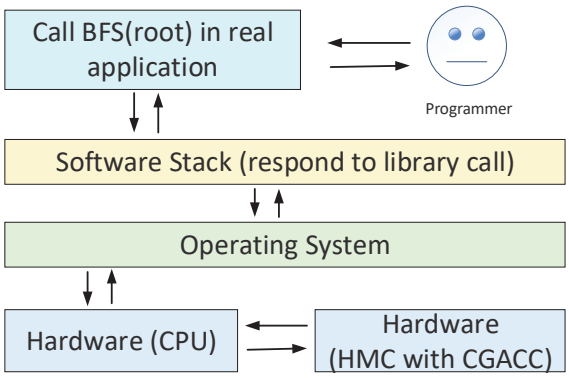


Figure 9. BFSworkflow from programmer to hardware.

3.1. CGAcc Structure

In this section, we introduce the components of CGAcc, which are illustrated in Figure 7.

(1) Register group: A collection of registers that maintains metadata and status information about a graph traversal. First, the Activation Register (AR) is used to enable CGAcc. When the CPU initiates a traversal, it sends an activation request to CGAcc, which includes a start vertex index. This request is recorded in the AR. Second, the Continual Register (CR) is used to store the current maximum start vertex index. This register is needed because a graph may contain several unconnected subgraphs. When the current subgraph is finished traversing, the address in CR will be used as the start vertex of the following traversal. Lastly, the End subgraph Register (ER) is used to record the end of the currently processed subgraph. For some on-line algorithms that only need partial traversal, there is no essential difference for CGAcc in traversing a complete graph or a partial graph. CGAcc just keeps fetching data from these three arrays on the memory side and sending the traversal order in the runtime. The CPU can stop the traversal procedure by setting the ER register if only partial traversal is needed.

(2) Prefetch group: The core of CGAcc. Because CSR represents graphs with three arrays, elements from these arrays can be prefetched by separate prefetchers. Thus, the prefetch group includes the Vertex Prefetcher (VEP), Edge Prefetcher (EP) and Visited Prefetcher (VSP). VEP receives and uses a new vertex index to access the visited array and to fetch vertex data, according to the visited status. The VEP reads the AR to start and then reads the CR to get the address of the next start vertex when notified that processing of the current subgraph is finished. In other cases, the VEP receives requests that contain the new vertex index from the VSP. When vertex data are fetched, the VEP will send some requests to the EP to fetch edge data. After edge data (extended by the current processing vertex) are fetched, the EP will send a request to the VSP to fetch visited data. It receives a request from the EP and then determines whether this vertex is

new by simply snooping to see if there exists a write access. The only situation where write access is issued to a visited array is when a new vertex (i.e., never visited before) is visited, and the value in its corresponding location in the visited array will be written as true. In this case, this vertex should be sent to the VEP as an expanded vertex for the following traversal.

(3) Internal cache: Used to reduce transaction latency. The cache is arranged as three small buffers: Vertex prefetch Cache (VEC), Edge prefetch Cache (EC) and Visited prefetch Cache (VSC). These buffers cache a portion of the vertex, edge and visited arrays. For a memory access by a particular prefetcher, the corresponding cache is accessed first. The data are directly fetched on a cache hit. Otherwise, the prefetcher associated with the array performs a memory access to the DRAM. The EC and VSC use Least Recently Used (LRU) replacement. The VEC uses an optimized replacement policy, which is described in Section 3.3. Although these prefetchers are independent, they share cache resources as part of CGAcc. These internal caches store data from different arrays (i.e., vertex, edge and visited arrays). At runtime, every prefetcher can access an arbitrary cache if necessary. For example, the VEP will not only access the data in the VEC, but also data in the VSC because the VEP will handle both the vertex and visited array.

(4) FIFO (First-In, First-Out) buffer: In our design, FIFO buffers (i.e., Vertex Buffer (VEB), Edge Buffer (EB) and Visited Buffer (VSB)) are needed for each prefetcher. Each buffer has entries to hold address information. The value in each entry is evicted after it has been accessed. These buffers store data in a specific way. Each entry in the VEB stores one address. The VEP uses this address to issue two accesses. Each entry in the EB is used to store an address pair (Addrs, Addre). The EP uses this address pair to issue multiple accesses. Finally, each entry in the VSB stores one address, and the VSP uses this address to issue one access.

3.2. CGAcc Operation

Graph traversal has a unique property that can be exploited: the currently-fetched data can be used as the next memory access address. Consequently, the next visited vertex can be known in advance. It is worth noting that the main optimization target of CGAcc is BFS graph traversal, as described in Section 2.2. Three loads will be issued if the next vertex needs to be determined. In a conventional memory, after the access to the vertex array, the accesses to edge and visited arrays are issued sequentially and continuously until all extended vertexes are traversed, which is time consuming. In fact, memory accesses in a traversal can be divided into three categories: to vertex array, to edge array and to visited array. These accesses can be pipelined. CGAcc leverages this property by arranging the VEP, EP and VSP (prefetch group) into a pipeline. Address calculation is also done in the prefetch group. For some more complicated cases (e.g., depending on calculations or random operation), CGAcc requires further support. For example, CGAcc needs to deploy more computational components to support random or more complicated operations so that all the tasks can be done on the memory side. This will be the subject of our future work.

CGAcc acts like a master rather than a slave, which means the CPU only needs to send a start request. After the request is accepted, CGAcc continues to fetch data until all graph vertexes are accessed. When a new vertex is found, which means the traversal order is confirmed, CGAcc will send the index of the new-found vertex back to the CPU as the traversal result.

A new traversal starts when the VEP detects that the AR has been set by the CPU. After that, the VEP will use the value in the CR and the CR + offset (the value of offset alliesto the data structure of the vertex array) to determine the addresses for the next two read accesses to the vertex array. These two reads fetch and insert data into the EB. As described in Section 3.1, the EB stores the address pairs. Every time, the EP takes the first element of the queue's head as an address and issues a load access to the edge array. At the same time, the first element of this queue entry is incremented by the offset. Once the first element is equal to the second element, the entry is evicted from the EB. The VSP issues load accesses using the head value of the VSB, which is then excited. Meanwhile, a Boolean judgment will be made, and if the result is false (which also means the new extended vertex is never

visited), the vertex address for this vertex will be calculated with the PIM part on the logic layer and inserted into the VEB. Each prefetcher fetches data until its corresponding buffer is empty. A traversal stops when all vertexes have been visited.

Table 2 describes CGAcc's operation: the events (left column) indicate actions that cause a particular operation (right column). CGAcc can outperform conventional prefetching due to shorter memory access latency (CPU side to DRAM layer vs. logic layer to DRAM layer) and the overlap of the CSR traversal operations among the prefetchers.

Table 2. Events that cause prefetcher actions. AR: Activation Register; CR: Continual Register; ER: End subgraph Register.

Operations	Event	Action
1. Operations in VEP	$AR = True$	$ActiveCGAcc$ $VEB.insert[CR]$ $AR = False$
	$VEB[top] = Addr$	$PrefetchAddr(Vertex[n])$ $PrefetchAddr + Offset(Vertex[n+1])$ $EvictVEB[top]$ $EB.insert(Vertex[n])$ $CR = Maximum(CR, Addr + Offset)$
	$ER = True$	$VEB.insert(CR)$ $ER = False$
2. Operations in EP	$EB[top] = \{Addrs, Addre\}$	$While \quad Addrs! = Addre$ $\quad PrefetchAddrs[Edge[n]]$ $\quad Addres+ = Offset$ $EvictEB[top]$ $VSB.insert(Edge[n])$
3. Operations in VSP	$VSB[top] = Addr$	$PrefetchAddr[Visited[n]]$ $EvictVSB[top]$ $If \quad Visited[n] == False:$ $\quad CalculateNewAddr$ $\quad VEB.insert(NewAddr)$

3.3. Optimization of On-Chip Cache

The CSR graph structure offers room for further optimization. In particular, the vertex information can be fetched in advance. As described above, when a read accesses the visited array, it means that the same vertex in the vertex array will be accessed in the near future. According to graph traversal, a write access to the visited array will follow a read access. The read is to the same vertex of the visited array if and only if the vertex has not been visited. When a read is made to the visited array (which can be detected by CGAcc), the corresponding vertex address can be calculated based on the visited array address, and two prefetches to the vertex array can be done simultaneously. Through this optimization, it is highly possible that memory accesses, which target the vertex array, will hit the cache. This is because the data were prefetched and stored in the cache, and if not, the data would be ready soon. Thus, if a data miss happens, the access just waits for the prefetched data to be inserted into the cache.

To support this optimization and ensure that the right data will be placed into the VEC, CGAcc deploys two registers: the Access Order Register (AOR) and the Vertex Order Register (VOR). The AOR is used to mark the order of every prefetch access. The initial value of the AOR is set to one. The value of AOR represents the order of prefetch access. This value accumulates when a write access to the visited array is detected. This data prefetching is out-of-order, which means that sometimes the prefetched data may not be used immediately. In this situation, the data are temporarily stored in a buffer. The VOR is used to mark the vertex access order. The initial value of VOR is set to zero, and the value of VOR accumulates while every pair of accesses to the vertex array is issued. The initial

values of AOR and VOR are set to one and zero because every prefetch access is issued before the next processing vertex access is issued.

When prefetched data (usually in a response access, which has the same order information as the request access) come back to the buffer (Prefetch Buffer (PB), used to cache the prefetched data that the VEP will access in the near future), the memory controller on the logic layer checks the order value to see whether it matches with the VOR. If the values match, the data will be evicted from the buffer and inserted into the vertex cache. Note that sometimes, the read access to the vertex array may directly hit the vertex cache, where the value of the VOR becomes larger than the value of the AOR. In this case, the prefetched data are dropped rather than inserted into the cache.

3.4. Generalized CGAcc

CGAcc is designed to accelerate breadth-first search traversal for CSR representation, it can also be applied to other applications with similar access patterns on CSR graphs.

BFS-like applications: CGAcc is most accurately called a "domain-specific accelerator", which can assist many applications, especially graph-related applications. "BFS-like" means the access pattern of the application is similar to the access pattern of BFS. CGAcc actually works well for a series of applications that show a similar access pattern to BFS. Single-Source-Shortest-Path (SSSP) is one application that has a similar access pattern to BFS, although their purposes differ. The Bellman–Ford algorithm [26] (or its derivative version, SPFA [27]) is widely used in a typical SSSP procedure in case some negative edges appear in a graph. Essentially, this algorithm is a relaxation that goes though all vertexes to relax the shortest path to other vertexes. The biggest difference in data structure between BFS and SSSP is that every entry in the visited array is only accessed once in BFS, while the visited array (i.e., distance) can be accessed multiple times in SSSP. Other similar applications include Connected Components (CCs), Kcore, Graph Coloring (GC), and others. CGAcc works well without modification or with slight adjustment of these applications.

Sequential Iteration Prefetching (SIP): SIP is a common access pattern in graph applications. The most obvious characteristic of this pattern is sequential iteration through vertex and edge data. Applications like Degree Centrality (DC), Triangle Counting (TC) and PageRankare representative benchmarks that have this kind of access pattern. Stride prefetch would do well if vertex and edge data were stored sequentially. Unfortunately, with the CSR format, the edge value is indexed by the vertex value, which makes the data access data-dependent and irregular and leads to inefficiency for a typical prefetcher. Nevertheless, with CSR, there is a similarity to the BFS-like applications. The only modification we need to make to CGAcc is to adjust the VSP to a fixed offset rather than the visited array, which has no relationship with the new vertex insertion. This is a simple generalization, and no extra hardware is needed.

Other access patterns: CGAcc can be applied to applications other than graph traversal. CGAcc takes advantage of the short latency between the logic and DRAM layers to shorten data transfer time, and thus, the data movement cost will be greatly reduced if many tasks are offloaded to the HMC. Furthermore, the prefetchers are arranged in a pipeline to improve the efficiency cooperatively. Thus, some memory-bound applications can be accelerated if they have obvious data dependence, which means the pipeline work style can effectively hide latency. For instance, CSR-based Sparse Matrix multiply Vector (CSR-SPMV) can be accelerated with CGAcc. By offloading most of the data load and computing tasks to the logic layer, CGAcc can fetch data in a pipelined way to do multiplication, improving performance. For other applications with different access patterns, the number of prefetchers may need to be adjusted for the best performance.

4. Experimental Setting

4.1. System Configuration

To evaluate CGAcc, we implemented it in the CasHMC simulator [28]. CasHMC is a cycle-accurate and trace-driven simulator that models the latest HMC. We used Intel PIN [29] to collect memory traces of graph traversal. The memory traces were collected across all parts of the algorithm (i.e., the traces included graph construction and initialization). We modeled an 8-core 2-GHz chip multi-processor with in-order cores. The buffers and cache in CGAcc were measured using CACTI 7.0 [30]. A conventional memory system with two-level cache (32 KB, 2-way L1 cache, 4-way L2 cache) and stream prefetching was used for the baseline.

The memory system was configured as a 4 GB HMC with 32 vaults and four high-speed serial links (30 Gb/s) connected to the HMC controller at the CPU side. Thirty-two TSV lanes shared the 10 GB/s vault data bandwidth, which means each TSV lane had a 2.5-Gb/s bandwidth. Timing parameters were set according to the HMC Specification 2.1 [1]. The detailed configuration is listed in Table 3.

Table 3. System configuration. EC: Edge prefetch Cache; VEC: Vertex prefetch Cache; VSC: Visited prefetch Cache.

Processor	**8-Core, 2 GHz, In-Order**
Cache (for baseline)	L1 Cache: 32 KB, 2-way L2 Cache: 2 MB, 4-way
Vault controller	close-page, 32 buffer size 16 command queue size
Link	4 SerDeslink, 30-Gb/s lane speed 480-GB/s max link bandwidth
On-chip cache	**VEC**: 16 KB, direct-mapping, latency: 0.15 ns power: 5.9 mW, area: 0.03 mm^2 **EC,VSC**: 64 KB, direct-mapping, latency: 0.3 ns power: 21.1 mW, area: 0.07 mm^2 each
HMC	32 TSVs, 2.5 Gb/s timing: tCK = 0.8 ns, tRP = 10 tRCD = 13, tCL = 13, tRAS = 27 tWR = 10, tCCD = 4
On-chip buffer	VEB, VSB: 1 KB, EB, PB: 32 KB

4.2. Workloads

The CSR-based graph traversal program was obtained from Graph-BIG [31]. Our benchmarks included nine real-world graphs from the SNAPdataset [25], which covers a variety of sizes and disciplines. The detailed information for each graph is listed in Table 1.

5. Evaluation

The speedup of CGAcc was evaluated relative to the baseline. In addition, we carried out several sensitivity studies to analyze different configurations of CGAcc.

5.1. Performance

Figure 10 shows a performance comparison of the baseline and CGAcc. Compared to the baseline, CGAcc achieved an average speedup of 3.51×. The performance gain can be attributed to several factors: (1) The prefetchers can directly access data from the DRAM layer in less time. Generally, the transaction latency between the logic and DRAM layers is half of the transaction latency between

the CPU and the DRAM layer. That is, CGAcc can process graph traversal "near memory" with much shorter latency. (2) Memory accesses are split into three parts according to their target array, and each one is assigned to a corresponding prefetcher. This pipeline helps hide transaction latency and improves memory-level parallelism. (3) The deployment of the cache in CGAcc further reduces transaction latency. The cache is direct-mapped, and the size is set according to the evaluation in Section 5.2. Accessing such a small cache costs less than 1 cycle in the HMC. Section 3.3 describes the optimization of the internal cache. To illustrate the impact of this optimization work, we compared the performance between the situations with and without this optimization. As Figure 11 shows, for almost all of the graph cases, the performance was better when cache optimization was applied. On average, the speedup reached 4.0%. This performance gain comes from the prefetching of data that are likely to be accessed in the near future.

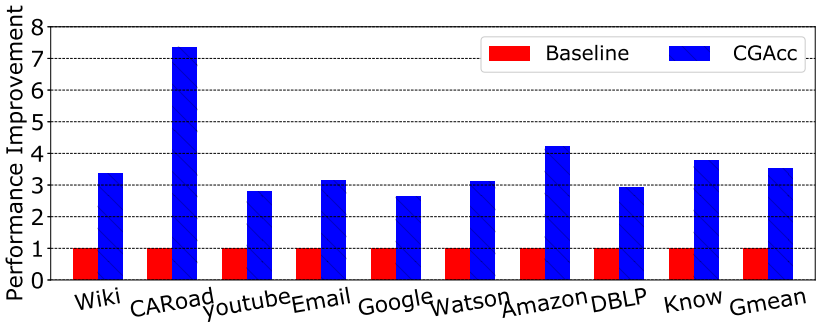


Figure 10. Comparison of the performance between baseline and CGAcc.

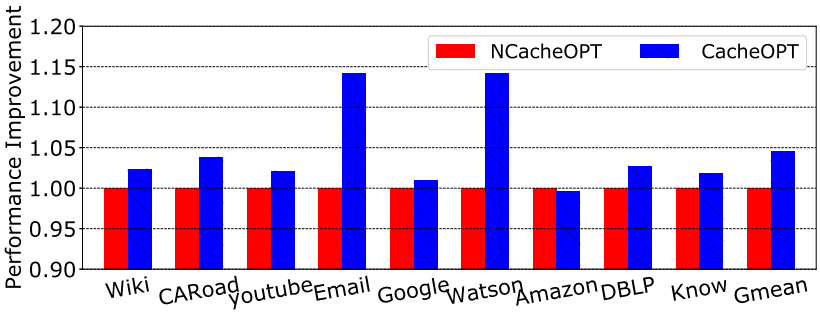


Figure 11. Comparison of the performance with/without cache optimization.

Previous work [2] described the latest state-of-the-art prefetcher for graph traversal. This prior work was based on conventional DRAM, and the prefetcher was placed in the CPU's cache system. We compared the performance of the graph prefetcher and CGAcc on common benchmarks. Figure 12 shows the results. On average, CGAcc had a 1.59× speedup compared with the graph prefetcher. We noticed that the speedup varied in different benchmarks. For example, the Web-Google benchmark had nearly the same speedup of (2.63× vs. 2.70×). However, California Road-Net had a relatively big difference of (1.9× vs. 7.4×). The reason for the speedup divergence is that we split memory accesses according to the array, and the imbalance between processing each category causes the prefetcher that has the heaviest task to be the bottleneck. Thus, CGAcc has better performance for sparse graphs (a sparse matrix generally contains flat numbers of vertexes and edges; the majority of real-world graphs are sparse matrices). As for the graph prefetcher in previous work, it could achieve flat speedup for arbitrary cases, but CGAcc outperformed it.

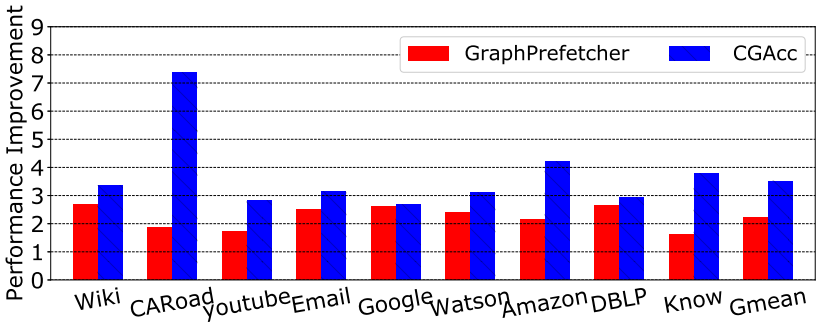


Figure 12. Comparison of the performance on a few benchmarks between graph prefetcher and CGAcc.

5.2. Effect on On-Chip Cache

We studied the effects of the on-chip cache on the capacity and associativity to determine the most suitable configuration. Table 4 shows the different latencies of the on-chip cache under different configurations. Essentially, the latency to access the on-chip cache made almost no difference because the frequency of the HMC was 1.25 GHZ, which means that any latency less than 0.8 ns would cost one cycle. Thus, the hit rate and the access latency to the CGAcc's cache directly affect performance. In this evaluation, we set the size value of each cache to be the same for simplicity.

Figure 13 shows a performance comparison between different cache capacity configurations. Considering the hardware cost of the logic layer, the cache capacity was limited to a maximum of 128 KB. From these results, as expected, as cache capacity increased, the speedup also increased. When normalized to 16 KB, the speedup was 1.17×, 1.51×and 1.52×, respectively for 32 KB, 64 KB and 128 KB. As performance is tightly related to the CGAcc cache hit rate, Figure 14 shows this parameter. We noticed that the hit rate of EC and VSC increased as cache size increased, but the hit rate of VEC was chaotic. This is because the EC and VSC use a classical LRU replacement policy, and thus, the properties of the hit rate and performance obey the typical LRU rules, as summarized above. However, as described in Section 3.3, the access from the VEP keeps accessing the VEC until it hits, which makes the hit rate of the VEC unpredictable. Therefore, it is not as necessary to make the VEC large, and the hit rate of the VEC does not affect the performance much. We also observed that the speedup seemed to saturate after capacity exceeded 64 KB. This is because the access latency increased when the memory capacity increased. In conclusion, we set the capacity of the EC and VSC to 64 KB, and we set the capacity of the VEC to 16 KB.

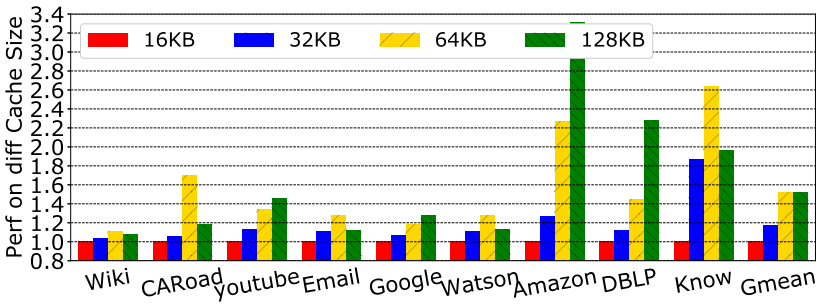
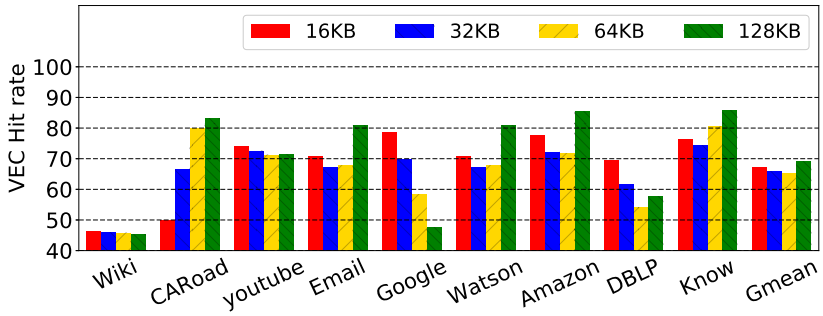
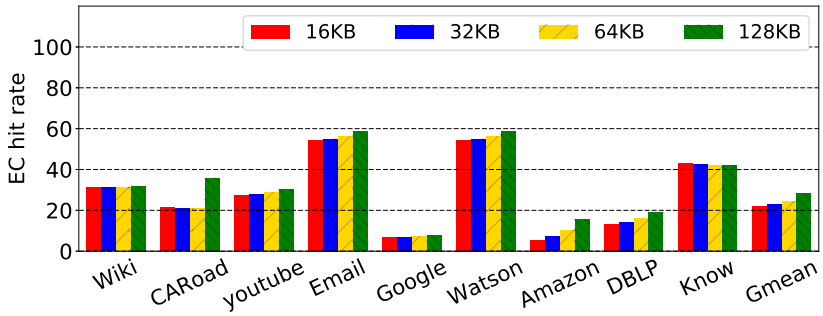


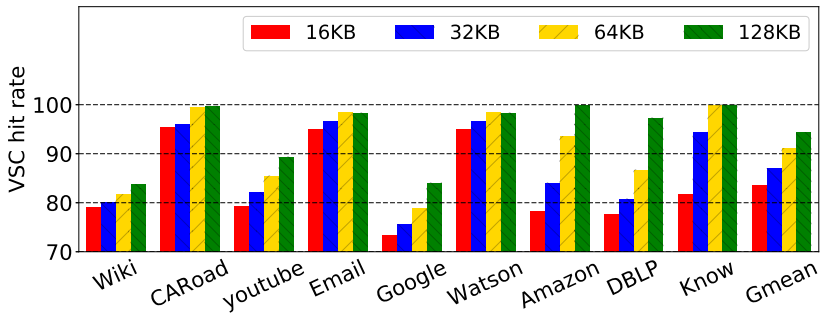
Figure 13. Comparison of the performance for different on-chip cache capacities.



(**a**) VEC hit rate comparison



(**b**) EC hit rate comparison



(**c**) VSC hit rate comparison

Figure 14. Comparison of the on-chip cache hit rate for different on-chip cache capacities.

We also evaluated four cache associativities: direct-mapped, four-way-set, eight-way-set and fully-associative. Figures 15 and 16 show the performance and cache hit rate for these configurations when cache capacity was set to 64 KB. In Figure 16, we used the normalized results because the hit rates for various associativities had tiny differences. Fully-associative had the highest hit rate. Considering the actual hardware cost and transaction latency, we set the cache associativity to be directly mapped.

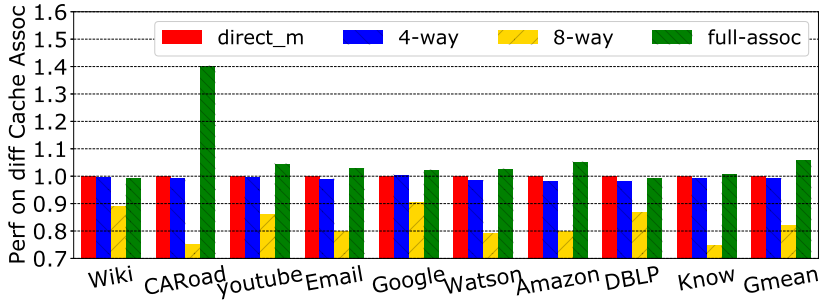
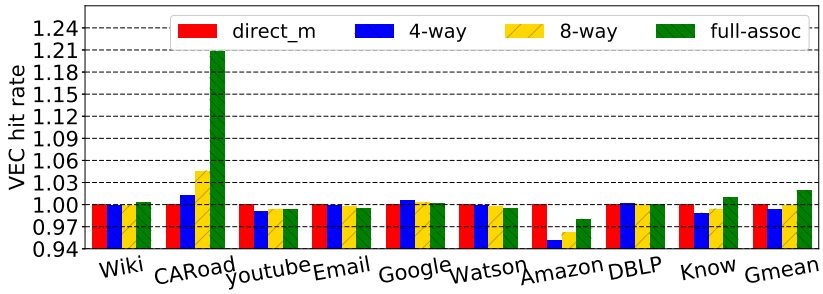
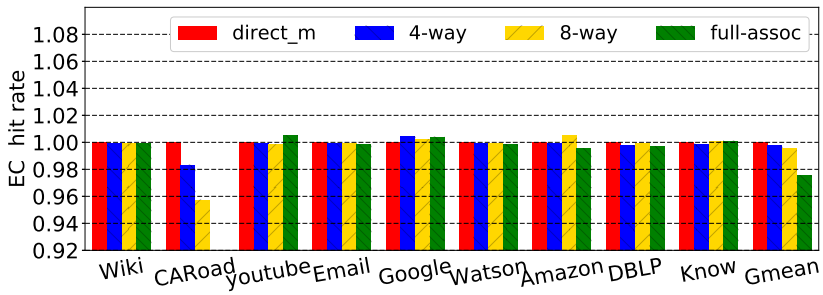


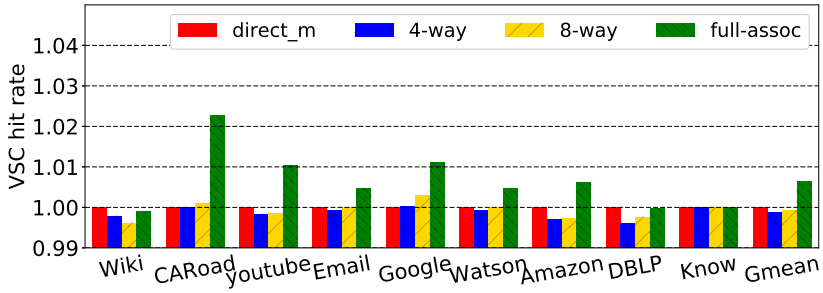
Figure 15. Performance comparison for different on-chip cache associativities.



(**a**) VEC hit rate comparison



(**b**) EC hit rate comparison



(**c**) VSC hit rate comparison

Figure 16. Comparison of the on-chip cache hit rate for different on-chip cache associativities.

Table 4. Access latency of the on-chip cache configuration (ns).

	16 KB	**32 KB**	**64 KB**	**128 KB**
Direct-Mapped	0.167	0.227	0.316	0.431
Four-Way-Set Assoc	0.420	0.454	0.464	0.523
Eight-Way-Set Assoc	0.753	0.779	0.812	0.868
Full Assoc	0.304	0.573	0.709	1.191

5.3. Effect on Graph Density

We also explored the effect of graph density. We considered two cases: (1) fix the vertexes, increase the edges; (2) fix the edges, increase the vertexes. To adjust the number of vertexes and edges flexibly, we used the Kronecker graph generator from Graph 500 [32]. For Case 1, we set the vertex factor to 16, which means the number was 65,536, and the edge factor varied from 5–25, which means the number varied from 655,360–3,276,800. For Case 2, we set the edge number to be 1,048,576, and the vertex number varied from 32,768–524,288.

Figures 17 and 18 show a performance comparison for Cases 1 and 2. Figure 17 shows that although the difference was not so obvious, CGAcc had the highest speedup in s16e15, which outperformed the average speedup by 1.7%. The speedup gaps between different graphs came from the imbalance of tasks assigned to each prefetcher. To make the data clear, we add Table 5 to show the speedup of CGAcc in this sensitivity test. For Case 2, the situation was similar to Case 1, but we noticed that when the vertex factor increased from 18–19 (i.e., the number of vertexes doubled),

the speedup had significant degradation. The VEC was saturated when the factor was 18, and when there were more vertexes, the VEP became a bottleneck, which slowed down the whole system performance.

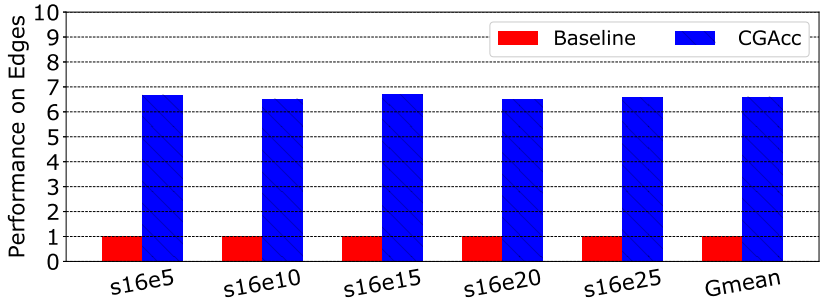


Figure 17. Comparison of performance between the baseline and CGAcc on edges.

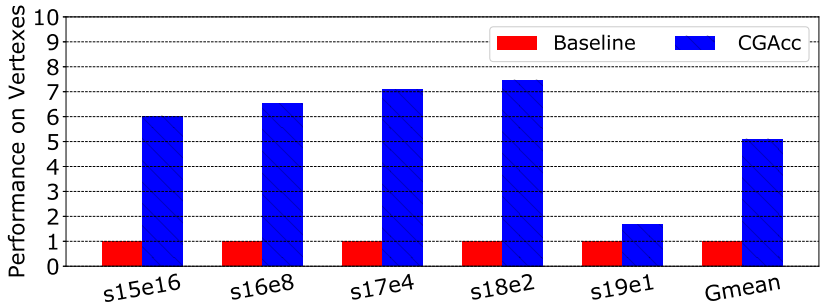


Figure 18. Comparison of the performance between baseline and CGAcc on vertexes.

Table 5. Speedup of CGAcc in the sensitivity test on edges.

Graph Case	s16e5	s16e10	s16e15	s16e20	s16e25	Gmean
Speedup	6.68	6.53	6.71	6.50	6.58	6.60

5.4. Generalized CGAcc

We explored the effects of CGAcc on other applications. For the BFS-like applications, we evaluated SSSP, Kcore and Graph Coloring (GC). For the sequential-iteration benchmarks, we evaluated Degree Centrality (DC), Triangle Count (TC) and PageRank. These benchmarks came from the GraphBig benchmark [31]. For illustration, we evaluated these benchmarks on three graphs: Amazon, Google and CARoad. Figure 19 shows the speedup of CGAcc on BFS-like benchmarks compared to the baseline and graph prefetcher. This result showed that CGAcc could work well for other applications: it obtained more than a 4× speedup on average for these three graphs and outperformed the graph prefetcher by 2.1× on average. The speedups for BFS-like applications were slightly smaller than BFS's speedup because the extra memory accesses (e.g., more accesses to the visited array in SSSP) caused one or more prefetchers to become a bottleneck. For sequence-iteration applications, Figure 20 shows that although the speedup was not as high as BFS-like applications, CGAcc could still achieve a 2.9× speedup on average and outperformed the graph prefetcher by 1.6×. In comparison to BFS-like applications, the lower speedup happened because sequential-iteration applications cannot always benefit from pipeline processing and use sequential access more often. Thus, the benefit mainly comes from the short latency between the logic and DRAM layers, rather than the pipeline processing provided by CGAcc.

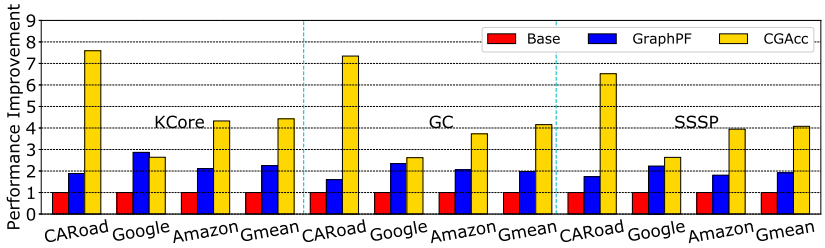


Figure 19. Comparison of the performance between the baseline, graph prefetcher and CGAcc on BFS-like applications. GC, Graph Coloring; SSSP, Single-Source-Shortest-Path.

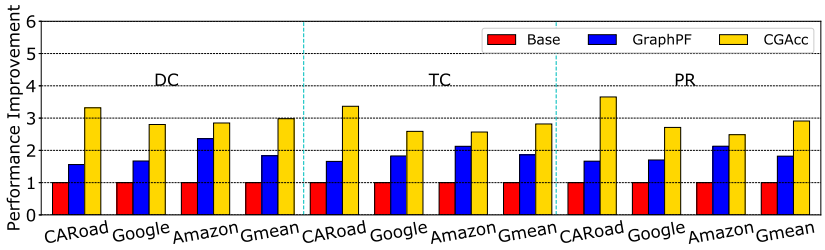


Figure 20. Comparison of performance between baseline, graph prefetcherand CGAcc on sequential-iteration applications.

5.5. CGAcc Prefetch Buffer

We explored reasonable sizes for the FIFO buffer described in Section 3.1. These buffers are used to store addresses for the following prefetching. Figure 21 records the entry amounts used, in which the blue, red, yellow and green lines refer to VEB, EB, VSB and PB, respectively. The VEB and VSB cost only a few entries because VEP and VSP can rapidly issue the read access as soon as the top entry of the corresponding buffer is not empty. EB and PB cost more entries. For the EB, this cost is incurred because every entry in the EB contains a pair of addresses, and each entry will not be evicted until the two addresses in a pair are the same (i.e., when the EP finishes traversing all edges). For the PB, the reason is that the prefetched data have to be stored until the VSB confirms whether the current vertex is visited or not, and during this time, the buffer will be cumulatively consumed. According to these results, we set the VEB and VSB to be 1 KB, while the EB and PB were set to be 32 KB.

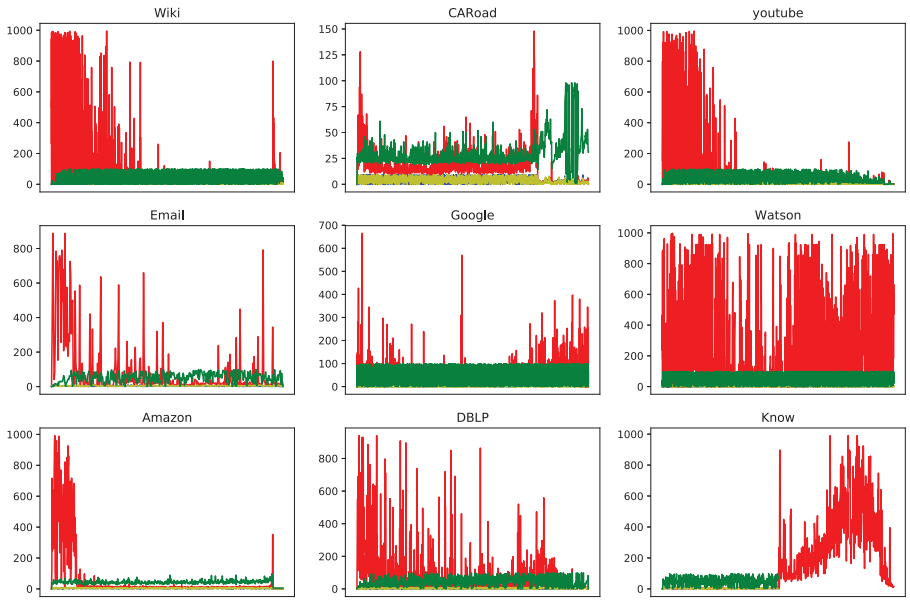


Figure 21. Entries' consumption for each buffer. Blue, red, yellow and green lines refer to VEB, EB, VSB and PB, respectively.

5.6. Hardware Overhead

As described above, the main hardware overhead of CGAcc comes from five parts: (1) five registers for metadata and status; (2) a small buffer to cache prefetched data; (3) input buffers for the prefetchers; (4) a prefetch buffer for optimization; and (5) prefetch logic. In terms of overhead, the caches and buffers are the biggest factor. Based on the sensitivity analysis in Sections 5 and 5.5, CGAcc has two 64-KB and one 16-KB directly-mapped caches to buffer the vertex, edge and visited arrays, respectively. The queue buffers are relatively small, and the prefetch buffer is 66 KB. In summary, the total size requirement is 210 KB for the logic layer, which is modest.

6. Graph Processing- and HMC-Related Work

6.1. Graph Processing-Related Prefetching

Several works have tried to accelerate graph traversal using the prefetching technique. Indirect patterns like A[B[i]] will lead to irregular memory access pattern, which will worsen the memory system performance. Yu et al. [14] proposed an efficient hardware Indirect Memory Prefetcher (IMP) to capture access patterns and hide latency. Essentially, it is a kind of stride-indirect prefetcher, and thus, the speedup is limited because graph traversal is quite irregular and not a stride-indirect type. On a GPU, irregular access pattern also causes performance degradation. Lakshminarayana et al. [33] proposed spare register-aware prefetching for load pairs that have one load dependent on the other, which is common in graph traversal algorithms. When the target load is detected, prefetch instructions will be issued, and the data are prefetched into spare registers, which are not used by any active threads. Based on the predictable access pattern property of graph traversal, Nilakant et al. [15] proposed PrefEdgeby using a look-ahead function to determine the memory-mapped address that will be used in the future, then to move the data from the SSD to the DRAM in advance. Through a parallelizing request to make full use of the maximum throughput from SSDs, PrefEdge can effectively hide the I/O latency. Ainsworth et al. [2] presented an explicitly-configured graph prefetcher. By snooping the data flow through the L1 cache and reasonably balancing the prefetching and common memory access time, it schedules timely loads of data before they are needed. Zhang et al. [16] proposed the Minnow engine, which augments CMP. Minnow consists of two parts: work list offload and work list-directed prefetching. Through offloading work list operations, it removes scheduling from the critical path to improve performance. Work list-directed prefetching is used to launch prefetch threads in respond to work list scheduling decisions. As we can see, generally, the key principle of all of these works is to use spare time for prefetching the data in advance, which is different from our design. These works are also mainly focused on the CPU side, which means the potential computational ability of the memory system is still not used. To the best of our knowledge, CGAcc is the first work that was specially designed for CSR-based graphs, using the HMC as a positive master for accelerating graph traversal.

6.2. Pointer-Related Fetchers

Some works concentrate on accelerating graph traversal in which graphs are not present in CSR format. Cooksey et al. [5] proposed a content-directed data prefetching architecture. This technique prefetches "likely" virtual addresses, which are observed in memory references. The speedup is moderate because the technique cannot avoid over-fetching, and it has some cache pollution. Al-Sukhni et al. [12] proposed the Compiler-Directed Content-Aware Prefetching (CDCAP) technique. CDCAP uses compiled-inserted prefetch instructions to convey the information of a dynamic data structure to a prefetching engine. With the help of such information provided by the compiler [34,35], CDCAP saves excessive prefetch operations. Ebrahimi et al. [36] proposed a low-cost hardware/software co-operative technique with two parts: compiler-guided prefetch filtering to tell the hardware the address that needs to be fetched and a prefetch throttling mechanism to manage multiple prefetchers in a hybrid prefetcher system based on runtime feedback. Roth et al. [11] proposed

a general Jump-Point Prefetching (JPP) to relieve the long access latency of handling linked structures. JPP overcomes the pointer-chasing problem by storing explicit jump-pointers. Lai et al. [13] proposed hardware-based pointer data prefetching. By identifying and storing pointer loads, this prefetcher fetches speculative virtual address data to a buffer. Limited by the graph representation, the speedups of these works are moderate. Because of its high efficiency and outstanding storage-saving ability when handling graph processing, CSR graph representation is quite widely used for large sparse graphs, which is definitely the feature of real-world graphs. CGAcc focuses on CSR-based graphs and obtains an impressive speedup.

6.3. HMC as an Accelerator

As a novel three-dimensional memory architecture, the HMC can be used as an accelerator in addition to the main memory. Some previous works used PIM to offload tasks from the CPU to memory. Kim et al. [17] proposed neurocube, a programmable and scalable digital neuromorphic architecture included in the logic layer of 3D memory. Based on the principle of memory-centric computing, neurocube consists of a cluster of PEs (Processing Engines) connected by a 2D mesh network for efficient neural computing. Dai et al. [18] proposed GraphH, which is a PIM architecture for graph processing on the HMC. Integrated with massive on-chip external hardware and algorithm optimization support, GraphH can obtain great speedup. Nai et al. [37] presented GraphPIM, which is similar to GraphH, but with less modification and cost. Some other works used the same concept, but focused on different optimization domains [38,39]. Based on 3D stacked memory, Hong et al. [40] proposed a novel Near Data Processing (NDP) architecture for Linked-List Traversal (LLT). This design includes NDP-aware data localization and LLT batching to reduce transaction latency and improve performance. Qian et at. [19] presented HMCSP, a simple optimization to extend the HMC's PIM capability to reduce the memory transaction latency of sparse matrix multiplication. Generally, this prior work treated the HMC as a co-processor that can undertake parts of computational tasks. In comparison, CGAcc treats the HMC as a smart master, which takes on active responsibility of the graph traversal. In addition, compared to these works, CGAcc concentrates on CSR-based graph representation and can obtain good speedup with moderate cost.

6.4. Graph Acceleration Architecture

Several prior papers aimed at designing a specific architecture for graph processing. Xu et al. [20] proposed OmniGraph on an FPGA. It is based on interval-shard and several computational engines. By combining these engines, OmniGraph can obtain good speedup when accelerating graph processing. There are also some similar FPGA-based works that show optimization for SSSP, all-pairs-shortest-path, parallel breadth-first search and strongly-connected components algorithms [41–44]. Ham et al. [23] proposed domain-specific Graphicionado, which exploits the data structure-centric datapath specialization and memory subsystem specialization. Dogan et al. [22] proposed a shared memory multi-core architecture. By introducing hardware-level messaging instructions into ISA, this design can accelerate synchronization primitives and move computation towards data more efficiently. These hardware-level works need extensional devices for acceleration, and thus, the overhead is larger than CGAcc because it is deployed on HMC, which can be treated as the main memory in the computer system. Software-level works cannot make full use of hardware. Besides, complex management frameworks or strategies are always needed to keep the design working well, which is also very costly. Song et al. [21] proposed GraphR. It is a ReRAM-based graph processing accelerator that consists of two components: memory ReRAM and Graph Engine (GE). The GE is responsible for processing graph computations, which are performed in a sparse matrix. GraphR can realize massive parallelism because of the unique feature of ReRAM. However, research about ReRAM remains in the theoretical stage. CGAcc is based on the HMC, which already has a hardware production. Thus, it is easier and more promising for CGAcc to be applied in real-world scenarios.

7. Conclusion

In this paper, we propose a novel compressed sparse row representation-based graph traversal accelerator on HMC, called CGAcc. Conventional prefetching techniques and parallel frameworks do not work well in handling irregular access patterns, which is precisely the memory access feature of graph traversal. Novel 3D stacked memory structures such as the HMC offer low transaction latency, very high bandwidth and PIM features, thus making them an ideal environment for optimization of graph traversal on the memory side. Armed with the knowledge of the CSR-based graph traversal's work flow and structure, the HMC acts like a master and uses the prefetchers in a pipelined way to reduce transaction latency and improve the overall performance. Comprehensive evaluations are described for several real-world graphs, which have numerous vertexes. Compared to a conventional memory system with stream prefetching, CGAcc could achieve a 3.51× speedup on average with small hardware cost. There are many research perspectives to improve CGAcc, such as the interaction with a streaming prefetcher, power gating-based energy-aware optimization, support for some more complicated graph processing algorithms, and so on. These works will be further explored in the future.

Author Contributions: Conceptualization, C.Q. and B.C.; Methodology, C.Q.; Software, C.Q.; Validation, C.Q.; Formal Analysis, C.Q.; Investigation, C.Q.; Resources, C.Q.; Data Curation, C.Q.; Writing—Original Draft Preparation, C.Q.; Writing—Review and Editing, B.C. and L.H.; Visualization, C.Q. and H.G.; Supervision, B.C. and L.H.; Project Administration, B.C. and Z.W.; Funding Acquisition, Z.W.

Funding: This work was supported by the NSFC under Grants 61472435 and 61572058 and YESSunder Grant 20150090.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Consortium, H. *Hybrid Memory Cube Specification 2.1*; Hybrid Memory Cube: Claremont, CA, USA, 2014.
2. Ainsworth, S.; Jones, T.M. Graph prefetching using data structure knowledge. In Proceedings of the 2016 International Conference on Supercomputing, Istanbul, Turkey, 1–3 June 2016; p. 39.
3. Falsafi, B.; Wenisch, T.F. A primer on hardware prefetching. *Synth. Lect. Comput. Archit.* **2014**, *9*, 1–67. [CrossRef]
4. Tran, H.N.; Cambria, E. A survey of graph processing on graphics processing units. *J. Supercomput.* **2018**, *74*, 2086–2115. [CrossRef]
5. Cooksey, R.; Jourdan, S.; Grunwald, D. A stateless, content-directed data prefetching mechanism. In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA, 5–9 October 2002; Volume 37, pp. 279–290.
6. Malewicz, G.; Austern, M.H.; Bik, A.J.; Dehnert, J.C.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: A system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, Indianapolis, IN, USA, 6–10 June 2010; pp. 135–146.
7. Low, Y.; Bickson, D.; Gonzalez, J.; Guestrin, C.; Kyrola, A.; Hellerstein, J.M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.* **2012**, *5*, 716–727. [CrossRef]
8. Xin, R.S.; Gonzalez, J.E.; Franklin, M.J.; Stoica, I. Graphx: A resilient distributed graph system on spark. In Proceedings of the First International Workshop on Graph Data Management Experiences and Systems, New York, NY, USA, 22–27 June 2013; p. 2.
9. Corbellini, A.; Mateos, C.; Godoy, D.; Zunino, A.; Schiaffino, S. An architecture and platform for developing distributed recommendation algorithms on large-scale social networks. *J. Inf. Sci.* **2015**, *41*, 686–704. [CrossRef]
10. Corbellini, A.; Godoy, D.; Mateos, C.; Schiaffino, S.; Zunino, A. DPM: A novel distributed large-scale social graph processing framework for link prediction algorithms. *Future Gener. Comput. Syst.* **2018**, *78*, 474–480. [CrossRef]
11. Roth, A.; Sohi, G.S. Effective jump-pointer prefetching for linked data structures. In Proceedings of the EEE Computer Society ACM SIGARCH Computer Architecture News, Atlanta, GA, USA, 1–4 May 1999; Volume 27, pp. 111–121.

12. Al-Sukhni, H.; Bratt, I.; Connors, D.A. Compiler-directed content-aware prefetching for dynamic data structures. In Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques, New Orleans, LA, USA, 27 September–1 October 2003; pp. 91–100.
13. Lai, S.C. Hardware-based pointer data prefetcher. In Proceedings of the 21st International Conference on Computer Design, San Jose, CA, USA, 13–15 October 2003; pp. 290–298.
14. Yu, X.; Hughes, C.J.; Satish, N.; Devadas, S. IMP: Indirect memory prefetcher. In Proceedings of the 48th International Symposium on Microarchitecture, Waikiki, HI, USA, 5–9 December 2015; pp. 178–190.
15. Nilakant, K.; Dalibard, V.; Roy, A.; Yoneki, E. PrefEdge: SSD prefetcher for large-scale graph traversal. In Proceedings of the International Conference on Systems and Storage, Santa Clara, CA, USA, 2–6 May 2014; pp. 1–12.
16. Zhang, D.; Ma, X.; Thomson, M.; Chiou, D. Minnow: Lightweight Offload Engines for Worklist Management and Worklist-Directed Prefetching. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, Williamsburg, VA, USA, 24–28 March 2018; pp. 593–607.
17. Kim, D.; Kung, J.; Chai, S.; Yalamanchili, S.; Mukhopadhyay, S. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, Korea, 18–22 June 2016; pp. 380–392.
18. Dai, G.; Huang, T.; Chi, Y.; Zhao, J.; Sun, G.; Liu, Y.; Wang, Y.; Xie, Y.; Yang, H. GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2018**, *20*. [CrossRef]
19. Qian, C.; Childers, B.; Huang, L.; Yu, Q.; Wang, Z. HMCSP: Reducing Transaction Latency of CSR-based SPMV in Hybrid Memory Cube. In Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Belfast, UK, 2–4 April 2018; pp. 114–116.
20. Xu, C.; Wang, C.; Gong, L.; Lu, Y.; Sun, F.; Zhang, Y.; Li, X.; Zhou, X. OmniGraph: A Scalable Hardware Accelerator for Graph Processing. In Proceedings of the 2017 IEEE International Conference on Cluster Computing (CLUSTER), Honolulu, HI, USA, 5–8 September 2017; pp. 623–624.
21. Song, L.; Zhuo, Y.; Qian, X.; Li, H.; Chen, Y. GraphR: Accelerating graph processing using ReRAM. In Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, Austria, 24–28 February 2018; pp. 531–543.
22. Dogan, H.; Hijaz, F.; Ahmad, M.; Kahne, B.; Wilson, P.; Khan, O. Accelerating graph and machine learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging. In Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, USA, 29 May–2 June 2017; pp. 254–264.
23. Ham, T.J.; Wu, L.; Sundaram, N.; Satish, N.; Martonosi, M. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–13.
24. D'Azevedo, E.F.; Fahey, M.R.; Mills, R.T. Vectorized sparse matrix multiply for compressed row storage format. In Proceedings of the International Conference on Computational Science, Atlanta, GA, USA, 22–25 May 2005; pp. 99–106.
25. Leskovec, J.; Krevl, A. SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.* **2016**, *8*, 1. [CrossRef] [PubMed]
26. Cheng, C.; Riley, R.; Kumar, S.P.; Garcia-Luna-Aceves, J.J. A loop-free extended Bellman-Ford routing protocol without bouncing effect. *ACM SIGCOMM Comput. Commun. Rev.* **1989**, *19*, 224–236. [CrossRef]
27. Fanding, D. A Faster Algorithm for Shortest-Path-SPFA. *J. Southw. Jiaotong Univ.* **1994**, *2*, 207–212.
28. Jeon, D.I.; Chung, K.S. Cashmc: A cycle-accurate simulator for hybrid memory cube. *IEEE Comput. Archit. Lett.* **2017**, *16*, 10–13. [CrossRef]
29. Luk, C.K.; Cohn, R.; Muth, R.; Patil, H.; Klauser, A.; Lowney, G.; Wallace, S.; Reddi, V.J.; Hazelwood, K. Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, 12–15 June 2005; Volume 40, pp. 190–200.
30. Wilton, S.J.; Jouppi, N.P. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid-State Circuits* **1996**, *31*, 677–688. [CrossRef]

31. Nai, L.; Xia, Y.; Tanase, I.G.; Kim, H.; Lin, C.Y. GraphBIG: Understanding graph computing in the context of industrial solutions. In Proceedings of the 2015 SC-International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, WA, USA, 25–30 June 2015; pp. 1–12.
32. Murphy, R.C.; Wheeler, K.B.; Barrett, B.W.; Ang, J.A. Introducing the graph 500. *Cray Users Group* **2010**, *19*, 45–74.
33. Lakshminarayana, N.B.; Kim, H. Spare register aware prefetching for graph algorithms on GPUs. In Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, 15–19 February 2014; pp. 614–625.
34. Gries, D. *Compiler Construction for Digital Computers*; Wiley: New York, NY, USA, 1971; Volume 24.
35. Muchnick, S. *Advanced Compiler Design Implementation*; Morgan Kaufmann: Burlington, MA, USA, 1997.
36. Ebrahimi, E.; Mutlu, O.; Patt, Y.N. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture, Raleigh, NC USA, 14–18 February 2009; pp. 7–17.
37. Nai, L.; Hadidi, R.; Sim, J.; Kim, H.; Kumar, P.; Kim, H. GraphPIM: Enabling instruction-level PIM offloading in graph computing frameworks. In Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, USA, 4–8 February 2017; pp. 457–468.
38. Aguilera, P.; Zhang, D.P.; Kim, N.S.; Jayasena, N. Fine-Grained Task Migration for Graph Algorithms Using Processing in Memory. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops, Chicago, IL, USA, 23–27 May 2016; pp. 489–498.
39. Ahn, J.; Hong, S.; Yoo, S.; Mutlu, O.; Choi, K. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Comput. Archit. News* **2016**, *43*, 105–117. [CrossRef]
40. Hong, B.; Kim, G.; Ahn, J.H.; Kwon, Y.; Kim, H.; Kim, J. Accelerating linked-list traversal through near-data processing. In Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, Haifa, Israel, 11–15 September 2016; pp. 113–124.
41. Zhou, S.; Chelmis, C.; Prasanna, V.K. Accelerating large-scale single-source shortest path on FPGA. In Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW), Hyderabad, India, 25–29 May 2015; pp. 129–136.
42. Attia, O.G.; Grieve, A.; Townsend, K.R.; Jones, P.; Zambreno, J. Accelerating all-pairs shortest path using a message-passing reconfigurable architecture. In Proceedings of the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Mayan Riviera, Mexico, 7–9 December 2015; pp. 1–6.
43. Attia, O.G.; Townsend, K.R.; Jones, P.H.; Zambreno, J. A Reconfigurable Architecture for the Detection of Strongly Connected Components. *ACM Trans. Reconfig. Technol. Syst.* **2016**, *9*, 16. [CrossRef]
44. Attia, O.G.; Johnson, T.; Townsend, K.; Jones, P.; Zambreno, J. Cygraph: A reconfigurable architecture for parallel breadth-first search. In Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW), Phoenix, AZ, USA, 19–23 May 2014; pp. 228–235.