Memory Fast-Forward: A Low Cost Special Function Unit to Enhance Energy Efficiency in GPU for Big Data Processing

Eunhyeok Park, Junwhan Ahn*, Sungpack Hong**, Sungjoo Yoo, and Sunggu Lee
Embedded System Architecture Lab, POSTECH
Design Automation Lab, Seoul National University*
Oracle Labs**

Abstract

Big data processing, e.g., graph computation and MapReduce, is characterized by massive parallelism in computation and a large amount of fine-grained random memory accesses often with* structural localities *due to graph-like data dependency. Recently, GPU is gaining more and more attention for servers due to its capability of parallel computation. However, the current GPU architecture is not well suited to big data workloads due to the limited capability of handling a large number of memory requests. In this paper, we present a special function unit, called* memory fast-forward (MFF) *unit, to address this problem. Our proposed MFF unit provides two key functions. First, it supports pointer chasing which enables computation threads to issue as many memory requests as possible to increase the potential of coalescing memory requests. Second, it coalesces memory requests bound for the same cache block, often due to structural locality, thereby reducing memory traffics. Both pointer chasing and memory request coalescing contribute to reducing memory stall time as well as improving the real utilization of memory bandwidth, by removing duplicate memory traffics, thereby improving performance and energy efficiency. Our experiments with graph computation algorithms and real graphs show that the proposed MFF unit can improve the energy efficiency of GPU in graph computation by average 54.6% at a negligible area cost.

1. Introduction

Big data processing is one of grand challenges in energy-efficient server design. Currently, a large number of power hungry out-of-order cores often waste significant amount of energy by waiting for data from memory or remote nodes. Big data processing, e.g., graph computation [1][2][3], MapReduce [4], etc., is often characterized by massive thread-level parallelism with a low ratio of computation to data, i.e., each thread has a small amount of computation. In addition, it often issues a large amount of fine-grained random memory accesses. For instance, MapReduce requires a significant amount of random traffics in its shuffle phase [4].

We focus on graph computation as a key area of big data processing. Graph computation is to perform database queries on the graph database. It is expected to become more and more popular in many applications including machine learning, data mining, big data analytics, etc. [3][5]. In graph databases [6][7], the relationship between data entities, e.g., websites, personalities, objects, etc., is represented in a graph structure. Thus, their structural connectivity, i.e., neighbor relationship, gives locality in memory accesses which we call *structural locality*. In graph computation, the characteristics of memory access is more complicated due to structural localities as well as massive fine-grained random accesses

Recently, GPU is gaining more and more attention for servers due to its capability of parallel computation [8]. However, as we will show in our study, the current GPU architecture is not well suited to emerging big data workloads. Even though GPU is originally designed to exploit memory bandwidth to overlap parallel computation with memory accesses, it is not prepared for new memory access behavior, massive fine-grained random accesses with structural locality, in big data processing. To be specific, the current GPU architecture is good at coalescing memory requests by utilizing 'instantaneous' memory access locality between threads in a thread group called *warp*. However, it lacks in exploiting memory access locality over the entire period of warp execution.

In our work, we present a low cost special function unit called *memory fast-forward (MFF)* unit which aims at maximizing memory request coalescing over the entire period of warp execution. In this paper, we show how the MFF unit can exploit structural locality in graph computation to significantly reduce the amount of memory requests thereby improving energy efficiency and performance. In order to expose and exploit structural locality which covers the entire period of warp execution, it is required to make threads to move fast forward in their memory accesses. To do that, our proposed MFF unit allows threads to specify pointer chasing functions. Then, on behalf of threads, the MFF unit performs pointer chasing to issue as many memory requests as possible which exposes structural locality by increasing the number of outstanding memory requests. The MFF unit, then, coalesces those requests to avoid duplicate accesses to the main memory.

This paper is organized as follows. Section 2 gives related work. Sections 3 and 4 explain preliminaries and our problem. Section 5 describes our proposed memory fast-forward unit. Section 6 reports experimental results. Section 7 concludes the paper.

2. Related Work

Several distributed frameworks have been presented for graph computation [1][2][3][5]. GBase is based on Hadoop Map-Reduce. It takes advantage of existing database on Hadoop by modifying graph data representations properly [1]. Oracle NoSQL database also supports graph representations in a similar way [7]. Pregel is based on a message passing-based vertex-centric model [2]. It iterates over multiple super-steps. In each super-step, each vertex is processed once and its vertex attributes are updated. Since the next super-step can be started only after all the vertices are processed in the current super-step, it is called bulk synchronous parallel (BSP) model. PowerGraph adopts asynchronous computation where the result of vertex processing can be immediately available for processing other (neighbor) vertices [3]. It also presents a vertex partitioning in order to handle the characteristics of real graphs, i.e., power law distribution of vertex degrees.

Recently, single machine-based graph computation is gaining attention due to its cost effectiveness, i.e. moderate performance at a low cost. TurboGraph is a synchronous graph computation engine running on a PC [9]. It keeps vertex data in the main memory and performs vertex-centric processing by sequentially fetching edge data from the disk. X-stream adopts an edge-centric processing [10]. Thus, instead of grouping edges for each vertex on the disk as in [9], edge data can be stored in a random order in partitions. On each partition, all the edges are processed to produce updates on vertex

978-3-9815370-4-8/DATE15/©2015 EDAA

attributes which are stored as a log in the solid state disk (SSD) thereby exploiting good sequential write performance in SSD.

In [11], Harish and Narayanan present a vertex-centric graph processing on GPU. Each vertex is assigned a thread and a large number of vertices are processed in parallel exploiting the thread-level parallel architecture in GPU. In [12], Hong et al. propose a fast breadth-first-search (BFS) scanning method and a bit vector-based vertex scheduling targeted for GPU. In this paper, we use, as the baseline, an integrated solution of both works in [11] and [12], i.e., vertex-centric processing with the bit vector-based scheduling.

There have been several studies on improving memory-related performance in GPU. In [13], Power et al. report a problem of high TLB miss penalty when virtual memory is supported in GPU. They present, as a solution, per-GPU core TLBs, a shared multi-threaded page table walker, and a shared page walk cache. In [14], Power et al. propose applying region coherence to reduce coherent traffics from GPU cores in the heterogeneous CPU-GPU architecture. Recently, NVIDIA announced Pascal as a future GPU architecture where 3D stacked DRAM provides high memory bandwidth [15]. We expect our proposed MFF unit can still provide a cost-effective solution to improving effective memory bandwidth in the Pascal architecture supporting virtual memory (with TLB).

Recently, Kocberber et al. present Widx, an on-chip accelerator for database hash index lookups [16]. Compared with Widx, our proposed MFF unit allows memory request coalescing to reduce memory traffics in order to better utilize memory bandwidth, which is critical in big data processing.

3. Preliminaries

In order to understand the current problem of graph computation on GPU and explain our basic idea, we first introduce a graph database representation and an example of graph algorithm called PageRank [17].

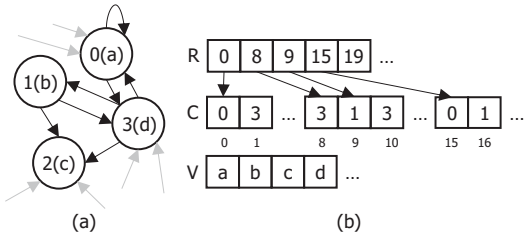

Figure 1 Graph example (a) and CSR representation (b).

Figure 1 (a) illustrates a part of a graph. A circle represents a vertex and an arrow an edge. Each vertex has an attribute (e.g., rank) in parentheses, e.g., vertex 2 has its attribute of *c*. Figure 1 (b) shows the compressed sparse row (CSR) representation of the graph in Figure 1 (a). CSR is one of widely used graph representations to reduce the graph size since vertices are typically sparsely connected with each other in the graph [18]. In Figure 1 (b), we assume an in-edge-based CSR representation.[1] As shown in the figure, the CSR representation uses three arrays: row-offset array (*R*), column-index array (*C*), and vertex attribute array (*V*). An entry of row-offset array, *R*[*i*] corresponds to a destination vertex *i* in the graph. *R*[*i*] represents a start index of the adjacency list (a list of indices of neighbor vertices) on the column-index array. For instance, in Figure 1 (b), *R*[0] has a value of zero which points to the first entry of the column-index array. Each entry of the column-index array has an index of a source vertex. Thus, in Figure 1 (b), *C*[*R*[0]]=0 represents that vertex 0 has an in-edge from itself as shown in Figure 1 (a). As another example, *C*[*R*[0]+1]=3 represents that vertex 0 (a destination vertex) has another in-edge originating from vertex 3 (a source vertex). Thus, while vertex 0 is being processed, if we need to access the vertex attribute of neighbor vertex 3, then we access an entry of the vertex attribute array in this way, *V*[*C*[*R*[0]+1]]=*V*[3]=*d*, which incurs a pointer chasing from *R*, *C* to *V* arrays. As shown in Figure 1 (a), vertex 0 has multiple in-edges. The number of in-edges of vertex 0, i.e., the size of the adjacency list for vertex 0 on array *C*, is calculated to be *R*[1]-*R*[0] (=8 in-edges) in Figure 1 (b) since *R*[1] represents the start index of the adjacency list on *C* for destination vertex 1.

```
1.  Input: {v|v ∈ V}, {e = (j → k) | j, k ∈ V, e ∈ E}
2.  Initialize: Rank_0(v) = p

3.  for v ∈ V
4.      T = 0
5.      for {w|(w → v) ∈ E}
            /* for i: R[v] to R[v+1] – 1                    */
            /* Steps 1~2 in Figure 3                        */
            /* allocate shared memory S                     */
            /* call PC_gather(S, R[v], R[v+1]-R[v])         */
            /* call PC_wait()                               */
6.          T += Rank_{l-1}(w) / N_out(w)
            /* T += V[C[i]] / N_out(C[i])                   */
            /* Steps 3~4 in Figure 3                        */
            /* for i:0 to R[v+1]-R[v]-1                     */
            /* T += S[i]                                    */
7.      Rank_l(v) = p + (1 − p)T

8.  p: random reset probability
9.  N_out(w): the number of out-degree of w
```

Figure 2 PageRank algorithm.

The graph algorithm, PageRank [17], calculates the popularity of an object, e.g., web page, personality, news, etc. An object corresponds to a vertex and their connectivity an edge. Figure 2 shows a pseudo code of PageRank on bulk synchronous parallel (BSP) model. Given an input graph having a vertex set *V* and an edge set *E* (line 1), the rank value, a vertex attribute, of each vertex is initialized (line 2). Lines 3-7 corresponds to a super-step of PageRank calculation. For a vertex *v*, we iterate over its in-edge list. For an in-edge *w*, we add, to a temporary variable *T*, the rank of the neighbor vertex (obtained in the previous super-step *l*-1) multiplied by a weight factor, $1/N_{out}(w)$ where $N_{out}(w)$ is the number of out-edges of the neighbor vertex (line 6). For simplicity, we assume $N_{out}(w)$ to be a constant in this example. After covering all the in-edges, we calculate an updated rank of vertex *v* at super-step *l* (line 7). We perform the super-step (lines 3-7) multiple times until all the ranks converge.

4. Motivation

PageRank in Figure 2 can be executed in parallel on the GPU. Typically, a vertex processing (lines 3-7 in Figure 2) can constitute a GPU thread [11][12]. As mentioned above, GPU execution is performed at the granularity of warp (e.g., 32 threads). Figure 3 illustrates memory accesses originated from the execution of a warp to process vertices in Figure 1.

In Figure 3, each thread first reads its corresponding entry of row-offset array, e.g., *R*[*j*] (Step 1 in Figures 2 and 3). Note that, thread *j* processes vertex *j* in Figure 3. Each thread, e.g., thread *j*, calculates the number of in-edges, i.e., *R*[*j*+1]-*R*[*j*] (Step 2). Each thread iterates over its in-edges (Steps 3 and 4, i.e., lines 5-6 in Figure 2). For the first in-edge, it accesses the corresponding entry of the

[1] In reality, we use two *C* (*R*) arrays, one for in-edges and the other for out-edges. Each vertex attribute has a separate array *V*.

column-index array, e.g., $C[R[j]]$ for vertex j (Step 3-a), in order to obtain the index of the source vertex. Then, the thread accesses the vertex attribute of the source vertex, e.g., $V[C[R[j]]]$ as shown in Figure 3 (Step 4-a). After obtaining the vertex attribute, its weighted value is added to the temporary value T (line 6 in Figure 2). As mentioned earlier in this paragraph, each vertex, i.e., each thread iterates Steps 3 and 4 over all of its in-edges.

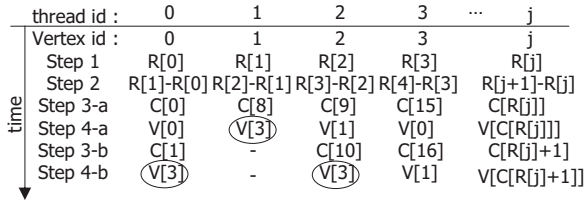

Figure 3 Memory access pattern in a warp run.

As Figure 3 shows, vertex processing generates frequent memory accesses, e.g., $R[j]$, $C[R[j]]$, and $V[C[R[j]]]$. Due to the structural locality, those memory accesses are often duplicated. For instance, in Figure 3, $V[3]$ is accessed three times from threads 0, 1 and 2 since vertex 3 is connected to vertices 0, 1 and 2 as shown in Figure 1 (a). In such a case of duplicate requests due to structural locality, we should be able to coalesce duplicate memory requests to avoid redundant memory traffics. However, the current GPU architecture lacks in the capability of coalescing requests in such a case. In Figure 3, for instance, two requests to $V[3]$ at Step 4-b are coalesced by the GPU core (e.g., symmetric multiprocessor in NVIDIA GPUs) while the other request to $V[3]$ at Step 4-a is not coalesced with the other two by the GPU core. Thus, two requests to the same data are issued incurring redundant consumption of memory bandwidth.

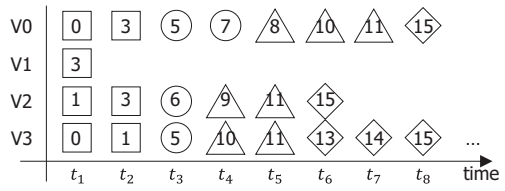

Figure 4 An illustrative example of duplicate requests.

Figure 4 illustrates the problem. Y-axis represents threads which process vertices. The symbols, ○, △, □ and ◇ represents memory read requests generated by the threads. The number in a symbol represents an address. Memory requests having the same symbol belong to the same cache block.

The conventional GPU architecture is good at instantaneous coalescing which coalesces memory requests in the same warp at the same clock cycle. In Figure 4, the four requests at time t_1 are coalesced by the GPU core into one coalesced request (CR) 1, since they all belong to the same cache block. Thus, after fetching the cache block only once from the memory, the four requests are served. Three requests generated at time t_2 are also coalesced into a CR 2. However, the GPU core is not able to coalesce CRs 1 and 2 as they are issued at different cycles. Thus, when a GPU core runs in the scenario of Figure 4, it will issue 11 requests to 4 cache blocks. In order to reduce memory traffics, such duplicate requests need to be coalesced by MSHR (miss state holding register) or served by L1 or L2 cache in the GPU.

In our investigation with representative graph algorithms and large real graphs, the existing MSHR and L1/L2 caches in the GPU are not effective in coalescing memory requests generated by graph computation thereby losing opportunities of further coalescing. There are two reasons with the failure of further coalescing. First, MSHR and caches on the GPU are small compared with high memory demand of graph computation. Above all, the large footprint of graphs makes caches less effective. In addition, due to a large number of warps (e.g., 120~240 warps in our experiments), the number of (coalesced) requests from GPU cores can be larger than what MSHR and caches can support (32 MSHR entry and 16 KB L1 cache per core cluster and 768KB L2 cache in GTX480 case).

Second, the memory stall-based warp execution model makes the problem even worse. If a memory operation is encountered during a warp execution, the execution stalls until the memory operation is finished. During the stall time, the GPU core executes other warps. However, from the viewpoint of the stalled warp, its memory requests come to spread over a long execution window. For instance, in Figure 3, a request to V[3] at Step 4-a and the other two requests to V[3] at Step 4-b come to be separated far apart on the time line since the warp can be stalled for a long period between Steps 4-a and 4-b. In such a case, requests can hardly find opportunities of further coalescing since their duplicate requests (and data) can be evicted from the MSHR (and caches) due to conflicts at the MSHR (caches) before their arrivals at the MSHR (caches).

Note that requests can be coalesced in two dimensions: the conventional spatial (instantaneous) coalescing across threads in a warp (e.g., four requests are coalesced into one at time t_1 in Figure 4) and a new *temporal coalescing* which covers the entire period of warp execution, e.g., from time t_1 to t_8 or later in Figure 4.

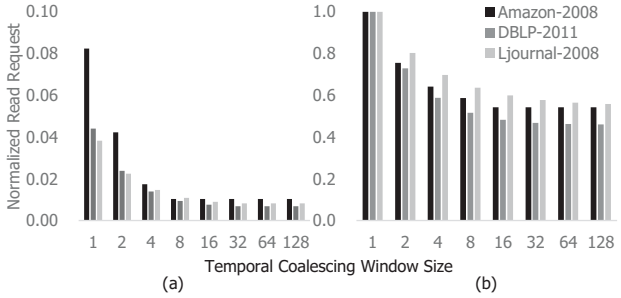

Figure 5 Potential of temporal coalescing of (a) step 3 and (b) step 4.

Figure 5 shows the potential of temporal coalescing for memory requests at Steps 3 (for array C) and 4 (array V) in Figures 2 and 3. We run PageRank with three real graphs on the GPU architecture used in our experiments (see Section 6). Y-axis represents the amount of memory requests normalized to the cases of unit window size in Figure 5 (b). X-axis represents the window size of temporal coalescing. The window of unit size corresponds to the case of baseline GPU architecture where only the instantaneous coalescing is applied and then the requests can also be coalesced or served by MSHR or caches. As the window size increases, more temporal coalescing can be performed thereby reducing memory traffics. For instance, in the case of graph *amazon-2008*, by applying temporal coalescing to 16 warp-level memory operations with minimum 16 coalesced requests and maximum 16*32 un-coalesced ones, we can reduce memory traffics by 45.8%. Since the performance of graph processing is sensitive to effective memory bandwidth, temporal coalescing has large potential of improvement in performance and energy efficiency (by shorter runtime and less memory traffics).

In order to realize the potential shown in Figure 5, we need as many outstanding requests as possible to expose structural locality as much as possible. However, due to the stall-based warp execution, each warp can issue only one set of coalesced requests at a time. Only after all the previous memory requests are served, the warp can resume its execution to issue next memory requests.

5. Memory Fast-Forward (MFF) Unit

Our proposed solution, memory fast-forward (MFF) unit enables a warp to issue much more memory requests than in the stall-based warp execution. To do that, the MFF unit performs pointer chasing (PC) on behalf of threads to make threads move fast forward in their memory accesses. In our current work, the MFF unit supports two main functions. First, it supports pointer chasing to issue as many memory requests as possible. Second, it performs temporal coalescing thereby reducing memory traffics. In order to utilize the MFF unit for graph computation, we use global variables, *active_vertex_list* (the bit vector used to identify vertices to process in the next super-step), and arrays *R*, *C*, and *V*. A GPU program can call one of the following functions.

PC_gather(*vertex_attr_array* target, int c_index, int size*): The function fetches the attributes of neighbor vertices to the array *target*. *c_index* is the start index of adjacency list in array *C* and *size* the number of vertex attributes to fetch, i.e., the number of neighbor vertices covered by the function *PC_gather()*. For each vertex (i.e., thread), it fetches *size* indices of neighbor vertices starting from index *c_index* of array *C*. Then, it fetches *size* vertex attributes from array *V*, and stores them in the array *target*. In our implementation, the maximum *size* is 32. Thus, when there are more than 32 in-edges for a vertex, *PC_gather()* is called multiple times for the vertex. Each thread in a warp calls *PC_gather()*. Thus, the array *target* of 32 threads in a warp constitutes a two-dimensional array having maximum 32 x 32 words which is allocated in the shared memory of the GPU core to avoid cache coherence issues. In summary, when *PC_gather()* is called by a thread in a warp, for the thread, the MFF unit first accesses the array C and then issues *size* read requests to access the array *V*. After receiving vertex attributes, it returns to the GPU core an array of *size* words, i.e., vertex attributes.

PC_scatter_set(*int c_index, int size*): It is a non-blocking function and typically used to set the bit vector for vertex scheduling [12]. For instance, the single source shortest path (SSSP) algorithm (to be explained in Section 6) uses this function to activate vertices for vertex processing in the next super-step. It first fetches *size* words from array *C* starting from index *c_index* in the out-edge-based CSR data. The fetched data corresponds to the indices of destination vertices. Then, the function sets the corresponding entries of *active_vertex_list* to '1'.

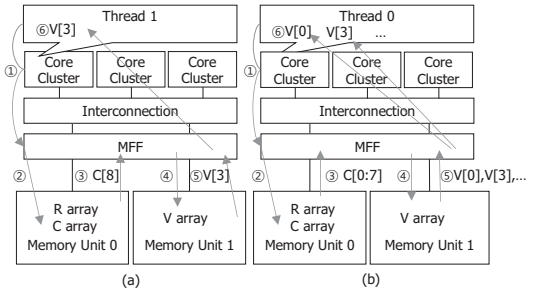

Figure 6 Memory fast-forward unit in the GPU structure.

The GPU program calls MFF functions as typical software-based prefetch functions. Thus, after calling an MFF function, e.g., *PC_gather()* (e.g., between lines 5 and 6 in Figure 2), the GPU program waits for the required data, e.g., vertex attribute, $Rank_{l-1}(w)$ by calling a simple synchronization function, *PC_wait()* as shown in Figure 2. On receiving the whole data in the array *target*, each GPU thread processes the associated data in *target*, e.g., adds the vertex attributes (after a weighting operation) to the temporary variable *T* in line 6 of Figure 2 until all the data of *target* are consumed.

Figure 6 shows where the MFF unit is located in the GPU architecture and how it coalesces memory requests. As shown in the figure, the MFF unit is located between on-chip interconnect and memory units. A memory unit consists of an L2 cache and memory controllers. There are typically 6~8 memory units (each with 2 memory channels) in a GPU [19]. Figure 6 illustrates how the MFF unit issues memory requests and communicates with the GPU core. In this figure, to give a concrete view, we use the operations in Figure 3 where memory requests to *V*[3] are coalesced by the MFF unit. We assume that the three arrays, *R*, *C*, and *V* are located in the memory connected to two memory units 0 and 1 as shown in the figure. For simplicity, we assume that data in array *R* already exist in the L1 cache of the GPU core.

In Figure 6 (a), assume that thread 1 calls *PC_gather()* with the arguments of the start index (=R[1] as shown in Figure 3) and the size (=R[2]-R[1]) of the adjacency list. The GPU core collects PC requests, i.e., *PC_gather*() calls, from threads of a warp and issues one warp-level PC command to the MFF unit. After calling *PC_gather()*, the warp is stalled at a synchronization point (*PC_wait*() in Figure 2) until a reply to the *PC_gather()* arrives from the MFF unit. When receiving, from the MFF unit, a reply to the PC command, i.e., vertex attributes in the array *target*, the GPU core resumes the execution of corresponding warp.

In Figure 6 (a), on receiving a warp-level PC command (Step 1 in the figure), for each thread, the MFF unit issues a read request to Memory Unit 0 to read a chunk of data from array *C* (Step 2). After reading a data chunk (e.g., *C*[8] in Step 3), the MFF unit sends a read request to Memory Unit 1 to fetch an entry from array *V* (Step 4). After reading the data, *V*[3] in this case (Step 5), the MFF unit sends the data *V*[3] (with the corresponding warp and thread IDs) to the GPU core (Step 6). The GPU core stores the data at the corresponding position in the array *target* on the local shared memory of GPU core. Note that the MFF unit performs the above operations for each thread associated with the PC command.

Figure 6 (b) shows how the MFF unit handles the PC command for thread 0 whose vertex 0 (in Figure 1 (a)) has multiple in-edges. In Step 1, the MFF unit receives the PC command from thread 0. Note that both Step 1's in Figure 6 (a) and (b) represent the same single warp-level PC command. After the MFF unit fetches a data chunk, *C*[0:7] (Steps 2 and 3), it tries to issue multiple (8 in this case, since the number of in-edges is 8) read requests to fetch corresponding entries in array *V*, i.e., *V*[0], *V*[3], etc. (Steps 4 and 5). In this case, the MFF unit can coalesce a request to fetch *V*[3] with an existing one issued by thread 1 (in Figure 6 (a)). The request coalescing allows the MFF unit to prepare the required data, i.e., *V*[3] early as well as issue less requests to the memory unit.

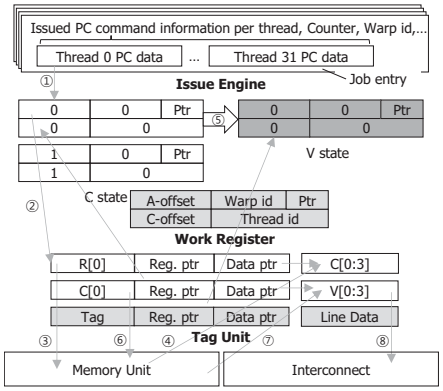

Figure 7 Operation detail of MFF unit.

Figure 7 shows the structure of MFF unit. It consists of three parts: Issue Engine, Work Register, and Tag Unit. Issue Engine receives

warp-level PC commands from GPU cores. Work Register manages the per-thread execution state of warp-level PC command. Tag Unit keeps the information (address, associated PC register and a pointer to data array) of outstanding memory requests and associated data array called Line Data. Assume that the warp in Figure 3 issues a PC command (corresponding to *PC_gather()*) to the MFF unit. On receiving it, a job entry is allocated in Issue Engine. As shown at the top of Figure 7, the job entry contains the PC information for each thread in the warp, a counter indicating the remaining total number of per-thread PC commands, and warp ID.

When there are free registers in Work Register, a job is issued and allocates a required number of registers (*size* registers for a thread) in Work Register (Step 1 in Figure 7). Figure 7 illustrates two allocated registers (two white rectangles). We call the state of initially allocated register *C state* since they contain information to access array *C* in the CSR format data. In this example, the two registers are used to fetch the attributes of two neighbor vertices for thread 0 in warp 0 (see the format of work register in the gray rectangle). They differ in A_offset (offset in the adjacency list) and C_offset (offset in cache block-size data in the Line Data of Tag Unit). According to A_offset values, the two registers are used for two entries, $C[R[0]]$ and $C[R[0]+1]$. Note also that the two registers are connected in a linked list using a field *Ptr* as shown in the figure.

After allocating a register, an associated entry of Tag Unit is allocated (Step 2 in Figure 7) and an associated read request is issued to the memory to fetch a chunk of data from array *C* (Step 3). The allocated entry of Tag Unit points to the associated register. In this example, a memory read request is issued to fetch a cache block $C[0:3]$ containing $C[0]$. The fetched block is stored in Line Data (Step 4). After obtaining the corresponding entry of array *C*, $C[0]$, the register changes its state from *C* state to *V* state (Step 5). As soon as the register becomes *V* state, the corresponding memory read request is issued, e.g., for $V[0]$ in this case (Step 6). A new entry of Tag Unit is allocated to keep the address information, $C[0]$. The new entry points to the associated register and a block in Line Data. When the read data, $V[0:3]$, arrive at the MFF unit, they are stored in Line Data (Step 7). Then, the required vertex attribute, $V[0]$ is sent to the GPU core (Step 8). When all the required vertex attributes (corresponding to a warp-level PC command) are sent to the array *target* on the shared memory of the GPU core, then the MFF unit sends a message to the GPU core to resume the execution of stalled warp. Then, the warp processes the data in *target* and the associated resources of Work Register and Tag Unit become free to be reallocated for a new job.

Figure 7 does not illustrates request coalescing in details. Every new memory read request looks up Tag Unit for a match. If there is a match, it can be coalesced with an outstanding one. In this case, a linked list is formed using a field *Ptr* between the *V* state register of the new request and that of the existing outstanding one. Thus, when the associated data arrive from the memory unit, all the coalesced requests in the linked list can be served.

6. Experiments

6.1. Experimental Setup

In order to evaluate the performance and power consumption of GPU, we use a cycle accurate GPU simulator, GPGPU-Sim (version 3.2.1) [19] and GPUWattch [20]. Table 1 shows the GTX480 configuration used in the experiments. The MFF unit has a configuration of 32 Issue Engines, 8 job entries per Issue Engine, 8192-entry Work Register, and 256-entry Tag Unit per memory unit and 20-entry Line Data (each cache line has 128B data). Their parameters are obtained through an extensive sensitivity analysis. The power consumption of MFF unit is estimated with CACTI6.5 [22] because the majority of area and power in the MFF unit is consumed by data storage and tag comparison.

Table 1 Hardware configuration

Model	Frequency	Core Cluster	Bandwidth	TDP
GTX 480(Fermi)	1.4 GHz	15	177.4 GB/s	250 W

Table 2 Graphs used in the experiments

Graph name	Vertices	Edges	Type
DBLP-2010	326186	1615400	Bibliography network
DBLP-2011	986324	6707236	
CNR-2000	325557	3216152	Italian CNR domain
Amazon-2008	735323	5158388	Similarity lookup
DeWiki-2013	1532354	36722696	Wikipedia snapshot

Table 2 shows 5 real graphs used in the experiments. These graphs are known to have similar characteristics, such as power-law degree distribution and small diameter, to large real-world graphs [21]. We use the CSR representation in our experiments. However, our method can be applied to other graph representations, e.g., the one in [6] since pointer chasing is inevitable in graph representations. We run 4 different graph algorithms with different characteristics of data access as follows.

- PageRank: We adopt BSP model. Thus, every vertex is updated in each super-step.
- BFS (breadth first search): It sweeps the graph structure. From a start vertex, it traverses all the paths following out-edges. Iterations continue until a specified depth is reached or all the vertices are visited.
- SSSP (single source shortest path): It is similar to BFS in that the traversal takes tree-like paths. However, as path costs are updated, vertices need to be revisited with updated path costs.
- WCC (weakly connected component): It determines largest connected components in the graph. Two vertices are considered in the same component if there is an edge between them. In the beginning, all vertices are updated and, as iterations continue, the number of updated vertices gets smaller.

Graph algorithms are typically composed of two stages, vertex schedule and algorithm computation. In our experiments, we focus on algorithm computation stage which takes most of graph computation runtime. We assume all the graph data reside in the main memory.

6.2 Experimental Results

Figures 8 shows the energy consumption of MFF-based GPU normalized to the baseline GPU. The MFF unit gives average 54.6% reduction in energy consumption. Reduction in energy consumption is related with both power consumption and speedup. We first give power analysis and then speedup.

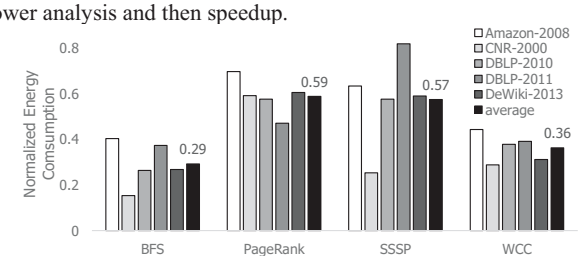

Figure 8 Energy consumption of MFF-based GPU.

Figure 9 shows the breakdown of power consumption. We show the results of two graphs (A: Amazon-2008 and B: DBLP-2011) due to space limit. As the figure shows, the MFF-based GPU gives higher power consumption, especially in main memory power. It is because the MFF unit enables more outstanding memory requests than the baseline and, thus, the memory needs to handle more requests per unit time. However, other components, e.g., core and interconnect (NOC in the figure) consume less power than in the baseline which will be explained in the performance analysis. The

overhead of MFF unit is not shown in the figure since it is negligible (0.3% of total energy).

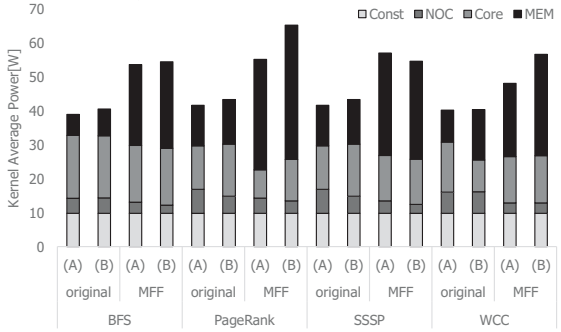

Figure 9 Power consumption breakdown.

Figure 10 shows the MFF unit gives average 1.75x~5.22x speedup. In each algorithm, speedup varies across graphs. In the case of BFS, speedup is the most significant because it uses only *PC_scatter_set()* and the scatter operation can coalesce up to 1024 requests (one for setting one bit in 128B data) into one for a cache block stored in Line Data of the MFF unit. In contrast, the average speedup of PageRank is smaller than the other algorithms because PageRank uses only *PC_gather()* and the gather operation can coalesce up to only 32 requests (each for 4B data) per cache block in Line Data. The other two algorithms use both functions thereby showing an intermediate level of speedup between PageRank and BFS. The MFF unit gives average 27.5% reduction in memory traffics. The MFF unit performs pointer chasing on behalf of threads near memory units (avoiding interconnect traffics), which reduces workload in on-chip interconnect as well as GPU cores thereby reducing their power consumption as shown in Figure 9. Our analysis on the interconnect traffics and stall cycles reveals that the reduced interconnect traffics enabled by the MFF unit significantly (by average 88.3%) reduces interconnect stall cycles, which also contributes to speedup. Note that the energy consumption in Figure 8 comes from the combined effects of reduced runtime (in Figure 10) and increased power consumption (Figure 9).

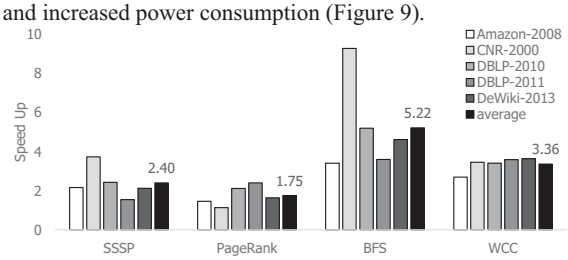

Figure 10 Speedup

In the baseline GPU architecture, a similar effect of temporal coalescing can be obtained by increasing cache sizes. We compared speedup between the baseline architecture with larger caches and the MFF-based architecture. We explored the sizes of L1 and L2 caches up to 16 times assuming an ideal condition that there is no increase in latency with larger caches. The larger L1 caches give the larger speedup. However, even in the case of 16 times larger, 256KB L1 cache, the large cache-based design with the ideal latency assumption still gives lower speedup (71% of MFF speedup) than the MFF-based design. In reality, the large (16x) L1 cache solution is prohibitively expensive in terms of power consumption (5.3x and 15.7x higher active and static cache power, respectively), cache access latency (2.3x longer latency than the conventional 16KB L1 cache), and area (additional area cost of 3.52MB in 15 core clusters). The L2 cache is not as effective as the L1 cache partly due to interconnect bottlenecks.

The area estimation with CACTI6.5 shows that the MFF unit consumes 0.87 mm^2 in 32nm technology. Compared to GTX480's die size, 521 mm^2 at 40nm and GTX780's die size, 551 mm^2 at 28 nm technology, the area overhead is negligibly small.

7. Conclusion

In our work, we investigated the conventional GPU architecture for emerging server workload, namely, graph computation. In order to resolve the problem of losing opportunity for further coalescing of memory requests, we present a low cost special function unit called memory fast-forward (MFF) unit and show how the MFF unit can exploit structural locality in graph computation to significantly reduce memory requests. Our experiments with four graph algorithms and real graphs show that the MFF unit gives average 54.6% reduction in energy consumption mostly due to the reduction in memory traffics. In future work, we will work on applying the concept of temporal coalescing and the MFF unit to larger systems, e.g., multi-GPU and other application areas.

8. Acknowledgement

This work was in part supported by the MSIP (Ministry of Science, ICT, and Planning), Korea under the "IT Consilience Creative Program" (NIPA-2014-H0201-14-1001) supervised by NIPA (National IT Industry Promotion Agency).

9. Reference

[1] U. Kang, et al., "Gbase: a scalable and general graph management system," Proc. KDD, 2011.
[2] G. Malewicz, et al., "Pregel: a system for large-scale graph processing," Proc. SIGMOD, 2010.
[3] J. E. Gonzalez, et al., "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," Proc. OSDI, 2012.
[4] B. He, et al., "Mars: A MapReduce framework on graphics processors," Proc. PACT, 2008.
[5] Z. Cai, et al., "A Comparison of Platforms for Implementing and Running Very Large Scale Machine Learning Algorithms," Proc. SIGMOD, 2014.
[6] Neo4j, available at http://www.neo4j.org/.
[7] Oracle NoSQL Database, http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html.
[8] NVIDIA, Co., "IBM, NVIDIA to Supercharge Corporate Data Center Applications and Next-Generation Supercomputers," press release, Nov. 2013.
[9] W. Han, et al., "TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC," Proc. KDD, 2013.
[10] A. Roy, et al., "X-stream: edge-centric graph processing using streaming partitions," Proc. SOSP, 2013.
[11] P. Harish, and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," Proc. HiPC, 2007.
[12] S. Hong, et al., "Efficient parallel graph exploration on multi-core CPU and GPU," Proc. PACT, 2011.
[13] J. Power, et al., "Supporting x86-64 Address Translation for 100s of GPU Lanes," Proc. HPCA, 2014.
[14] J. Power, et al., "Heterogeneous System Coherence for Integrated CPU-GPU Systems," Proc. MICRO, 2013.
[15] NVIDIA Co., GPU Roadmap, keynote, GPU Technology Conference, March, 2014, http://www.gputechconf.com/highlights/2014-replays.
[16] O. Kocberber, et al., "Meet the walkers: accelerating index traversals for in-memory databases," Proc. MICRO, 2013.
[17] L. Page, et al., "The PageRank citation ranking: Bringing order to the web," Technical Report, Stanford InfoLab, 1999.
[18] T. Armstrong, et al., "LinkBench: a Database Benchmark Based on the Facebook Social Graph", Proc. SIGMOD, 2013.
[19] A. Bakhoda, et al., "Analyzing CUDA workloads using a detailed GPU simulator," Proc. ISPASS, 2009.
[20] J. Leng, et al., "Gpuwattch: Enabling energy optimizations in gpgpus," Proc. ISCA, 2013.
[21] A. Mislove, et al., "Measurement and analysis of online social networks," Proc. IMC, 2007.
[22] CACTI 6.5, http://www.hpl.hp.com/research/cacti/.