IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 25, 2018, accepted September 26, 2018, date of publication October 16, 2018, date of current version November 8, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2876201

Accelerating BFS via Data Structure-Aware Prefetching on GPU

HUI GUO[1], LIBO HUANG[1], YASHUAI LÜ[2], JIANQIAO MA[1], CHENG QIAN[1], SHENG MA[1], AND ZHIYING WANG[1], (Member, IEEE)

[1]National University of Defense Technology, Changsha 410073, China
[2]Space Engineering University, Beijing 101416, China

Corresponding author: Libo Huang (libohuang@nudt.edu.cn)

This work was supported in part by the NSF of China under Grant 61433019, Grant 61472435, Grant 61572058, Grant 61672526, Grant 61202129, and Grant U14352217, in part by YESS under Grant 20150090, and in part by the Research Project of NUDT under Grant ZK17-03-06.

ABSTRACT Breadth First Search (BFS) is a key graph traversing algorithm for many graph analytics applications. In recent decades, as the scale of the graph analytics problem has become larger and larger, it has raised many interests to accelerate graph traversing on GPU. However, due to the irregular memory access pattern of BFS, a great number of the memory divergent accesses harm the efficiency of GPU dramatically. Data prefetching can fetch useful data into the on-chip memory in advance to reduce the latency of accessing the off-chip memory. However, traditional prefetching techniques on GPU cannot deal with irregular memory accesses efficiently. By analyzing BFS algorithms for GPU, we find an opportunity to design an efficient prefetching mechanism by using the explicit information of the graph data structure. In this paper, we propose DSAP, a data structure-aware prefetcher on GPU that generates prefetching requests based on the well-defined data structure access pattern of BFS. Also, we introduce an adaptive fine-grain prefetching management to adjust the status of the prefetching granularity dynamically to balance the cache resource contention and data prefetching based on the utilization of the prefetched data. We implement DSAP on a GPGPU-sim simulator and evaluate six data sets from three different kinds of applications. DSAP can achieve a geometrical mean IPC improvement of 28%, up to 48.4%, compared with that of GPU with no prefetching technique, while in contrast, a stride-based global history buffer prefetching mechanism makes no effects on improving BFS performance for these data sets. Also, we use the GPUWattch to estimate the power consumption, and the power increases 8.3% in average and up to 11.8%, but the total energy cost drops 15.1% in average.

INDEX TERMS Accelerator architectures, breadth first search, data structure aware, GPGPU computing, prefetching mechanism, irregular memory access.

I. INTRODUCTION

In the big data era, GPU has been successfully applied to solve big data problems for many applications. As the scales of graphs are increasingly larger, it has raised many interests to accelerate graph analytics applications on GPU, such as Single Source Shortest Path (SSSP) and Graph Coloring (GC). Because most of the graph analytics applications have uncomplicated arithmetic calculations, the biggest cost comes from memory accesses generated by graph traversing. Breadth First Search (BFS) is the most widely used graph traversing algorithm. However, GPU cannot accelerate BFS efficiently due to the irregular memory access pattern of BFS. GPU has to issue more than one memory requests for one irregular memory access, which dramatically impacts its efficiency. Moreover, GPU has a bad caching behaviour for the data structures of the graph, the miss rate of which is even greater than 80%. Fig. 1 shows that GPU has a relatively high miss rate for the data structures of the graph, due to the irregularity of access pattern of BFS. As a result, all the threads have to spend many cycles to wait for data and even worse, GPU cannot achieve the latency hiding through its massive parallelism due to the insufficient arithmetic calculations of BFS.

Data prefetching is one of the promising techniques to improve the efficiency of memory accesses and cache efficiency. Typical prefetchers on GPU, such as stream

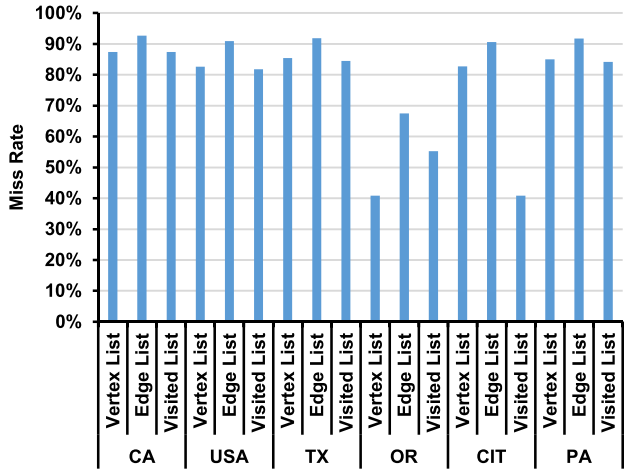

FIGURE 1. The GPU L1 cache miss rates of accessing the different graph data structures for different data sets respectively.

prefetchers [1], [2], stride prefetchers [3], [4] and GHB prefetchers [5], [6], can effectively reduce the memory latency for applications with regular access patterns. MT-prefetching [7] proposes an inter-thread prefetching mechanism with a hardware prefetcher training to reduce the negative effect of prefetching on GPU. APRES [8] combines warp scheduling and prefetching to improve cache efficiency. These prefetching techniques reduce the latency of accesses to off-chip memory efficiently and effectively. However, for irregular memory accesses, the error rate of prefetching prediction based on typical prefetching mechanisms is much higher than that for regular memory accesses. The high error rate of prediction causes severe cache pollution and memory bandwidth waste due to the useless data prefetching. Also, typical pattern-detected based prefetching mechanisms are too inefficient to identify complicated and various access patterns of irregular memory accesses. And, due to the inefficient pattern detection, these prefetching mechanisms almost make no contributions to reduce the latency of memory accesses and improve the efficiency of GPU. Therefore, the recent proposed GPU prefetching mechanisms based on typical prefetching mechanisms are not able to deal with data-dependent memory accesses of BFS.

By analyzing the BFS algorithm for GPU, although its memory accesses are data-dependent and highly irregular, its access pattern to the graph data structure is well-defined and predictable. The order of accessing the graph data structure is fixed, according to the definition of the breadth first search. For example, assuming a node is to be searched, the next step of BFS is to find all the edges the node connects. With this information, a prefetcher can fetch the data of all the edges in advance as long as the ID of the node to be searched is known. Based on this observation, we propose a Data Structure-Aware Prefetching (DSAP) mechanism, which is explicitly informed of the knowledge of the graph data structure access pattern of BFS. With the information of the node being searched, DSAP can prefetch the necessary graph data for the next node to be searched. Although, we only discuss the usage of DSAP for BFS in this paper, but it also benefits other graph analytics applications that have the same data structure access pattern with BFS.

Besides, where to store the prefetched data needs to be considered, due to the restricted on-chip storage resources of GPU. Because L1 cache has a relative sufficient memory space and has the advantage of data management, DSAP selects L1 cache as the storage for the prefetched data. However, cache pollution and early eviction are common problems caused by prefetching for caches. Many works tend to use metrics of the cache miss rate and early eviction rate to determine whether to do data prefetching or not. Although this can avoid the harm of data prefetching, but the performance degrades dramatically at the same time. According to the knowledge of the data structure access pattern of BFS, DSAP adopts an adaptive fine-grain prefetching management to balance the cache resource contention and data prefetching. DSAP sets up multiple prefetching statuses that represent different prefetching granularities, and DSAP can control the number of the generated prefetching requests by dynamically switching the status based on the cache early eviction rate.

In this paper, we propose a Data Structure-Aware Prefetching (DSAP) that generates prefetching requests based on the well-defined data structure access pattern of BFS and the knowledge of the graph data structure. DSAP can prefetch the graph data accurately with the explicit graph data structure information and improve the cache efficiency by the adaptive fine-grain prefetching management. For six datasets from three different applications, DSAP improves the IPC by 28% in geometric mean, and up to 48.4% with an average of 15.1% energy cost reduction, compared to the GPU without prefetchers, while the stride-based GHB prefetcher cannot improve the performance. The following summarizes our most important contributions:

- We first analyze the shortcomings of typical prefetching mechanisms on GPU for irregular memory accesses and the irregularity of memory accesses of BFS. We demonstrate that typical prefetching mechanisms have high error rates and cannot detect the irregular memory access patterns of BFS.
- We find that the graph data structure access pattern of BFS is well-defined and predictable. We propose the Data Structure-Aware Prefetching mechanism (DSAP) that uses the explicit knowledge of the graph data structure access pattern to help it improve the prefeching accuracy.
- We observe that simple metric-based prefetching managements result in performance degradation. We propose an adaptive find-grain prefetching management to balance cache efficiency and prefetching efficiency.
- We design and implement the proposed DSAP on GPGPU-sim simulator, and demonstrate that DSAP can improve the performance of BFS by prefetching data with the explicit information of the graph data structure

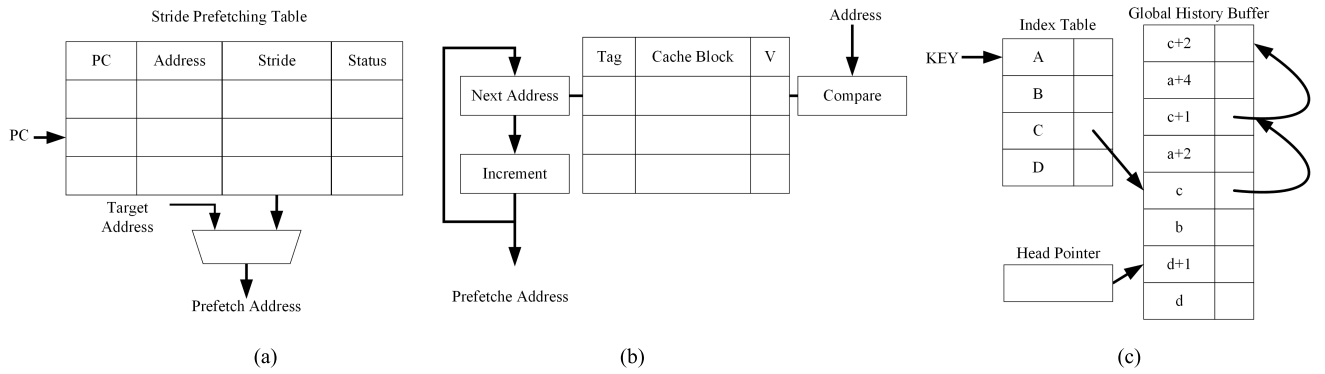

FIGURE 2. The designs of the three typical prefetching mechanisms. (a) Stride prefetching. (b) Stream prefetching. (c) GHB prefetching.

and memory access pattern of BFS, and balance the cache efficiency and the data prefetching by the adaptive fine-grain prefetching management.

II. BACKGROUND AND MOTIVATION

A. GPU PREFETCHING

Prefetching mechanisms on GPU have been widely researched. Generally, there are three kinds of typical prefetching mechanisms: stride prefetcher, stream prefetcher and GHB prefetcher. Fig. 2 shows the designs of the three typical prefetching mechanisms.

1) STRIDE PREFETCHER

The stride prefetcher (Fig.2 (a)) uses a table to record the local history information of memory accesses, including the program counter (PC, as the index of the table), the last address (for the computation of the next local stride), the most recent stride (the difference between two recent addresses) and the status of the recent stride [3], [4]. If the constant offset is found for the same PC, the stride prefetcher will generate prefetching requests by using the constant stride and the last accessed address.

2) STREAM PREFETCHER

The stream prefetcher (Fig.2 (b)) usually tracks the accessing direction of a memory region [1], [2]. When the accesses have the same direction, the stream prefetcher will prefetch data in this direction cache line by cache line. The prefetched cache lines are stored in the stream buffers not in the cache to avoid polluting the cache. If an access misses in the cache, the prefetched cache line will be fetched into the cache. If the sequential access pattern changes, the stream buffers will be flushed.

3) GHB PREFETCHER

The global history buffer (GHB) prefetcher (Fig.2 (c)) uses the global history buffer to store all the global addresses of the missed accesses and the global history buffer is organized as a FIFO table [5], [6]. Each of the GHB entries stores a miss address and a pointer. The pointers link the GHB entries in the time-ordered sequence. Another table is the index table. It stores the keys as the index of each entry and the key may be the PC of an instruction or a miss address. The index table also stores the pointers into the global history buffer for the entries. The GHB prefetcher can work together with other prefetching mechanisms, like stride prefetching and stream prefetching, to identify various memory access patterns.

Many research works are based on these prefetching mechanisms. Jog *et al.* [9] propose a prefetching technique on memory side to improve L2 cache hit rates for general purpose GPU applications. Inter-thread L1 data prefetching [10] combines prefetching mechanism with warp scheduling policy to improve the efficiency of prefetching. Koo *et al.* [11] use a leading warp to compute the starting address early for prefetching the data accessed in stride pattern. These works demonstrate that prefetching mechanisms can benefit general purpose GPU applications with regular memory accesses. However, for applications with irregular memory accesses, like BFS, the typical prefetching mechanisms have a poor performance. We implement a Next Line prefetcher and a stride-based GHB prefetcher on GPGPU-Sim, which represent the implementations of the streaming prefetcher and the combination of the stride prefetcher and the GHB prefetcher respectively. For BFS, both of the two typical prefetchers are not able to improve the performance of GPU. The performance of the stride-based GHB prefetcher almost equals to that of GPU without prefetchers, while the performance even gets 8% worse on average when using the Next Line prefetcher. The aggressive Next Line prefetcher pollutes the caches with useless prefetched cache lines and wastes the memory bandwidth. The stride-based GHB prefetcher has an extremely low coverage (<1%) for the memory accesses of BFS and is hard to find the constant stride access pattern for BFS. Therefore, as the irregular applications, such as graph computing applications, becomes more and more popular, the need of the prefetching mechanisms for irregular applications becomes more urgent.

B. PROGRAMMING MODELS FOR BFS

In general, breadth first search includes two basic operations on graph data structure: (1) modifying the visited statuses of nodes and (2) generating the frontier for the next

iteration of search. Two common models are used to mapping BFS onto GPU: topology-driven and data-driven [12]. The topology-driven implementation (Algorithm 2) uses the thread index to decide which node to be searched. Every thread checks the indexed node to see whether the node is in the frontier. Corresponding threads apply compute operators (accessing all the edges the node connects, checking the visited statuses of the neighbor nodes and updating the frontier for the next iteration) to the nodes in the frontier, while threads corresponding to nodes not in the frontier do none of compute operators. The kernel is called iteratively until the frontier is empty. The data-driven implementation (Algorithm 1) introduces a new data structure, called work list, as the frontier of each iteration to store the IDs of the nodes to be searched and shares the work list across all the threads. Each warp is assigned a small chunk of the work list to search every iteration and threads in the same warp search a node ID from the work list each time. Threads from the same warp access all the edges the node connects, check the visited statuses of the neighbor nodes in parallel and push the unvisited neighbor nodes into the work list for the next iteration.

Algorithm 1 The Data-Driven Implementation of BFS

```
1:  function BFS_kernel(graph, worklist, curr_level)
2:      tid ← blockIdx.x * blockDim.x + threadId.x
3:      lane_id ← tid%WARP_SZ
4:      warp_id ← tid/WARP_SZ
5:      task_start ← warp_id * CHUNK_SZ
6:      task_end ← task_start + CHUNK_SZ
7:      for id = task_start → task_end do
8:          v ← worklist[id]
9:          edge_ptr ← graph.vertexlist[v]
10:         num_edge ← graph.vertexlist[v+1] − edge_ptr
11:         for i = lane_id → num_edge do
12:             vid ← graph.edgelist[i + edge_ptr]
13:             if graph.visitedlist[vid] == INFINITY then
14:                 graph.visitedlist[vid] ← curr_level + 1
15:             end if
16:         end for
17:     end for
18: end function
```

In general, the data-driven implementation is more efficient to process BFS than the topology-driven implementation on GPU. For the topology-driven implementation, if a large proportion of nodes are not in the frontier, this will lead to inefficient parallel processing and performance degradation, while for the data-driven implementation, threads working on different chunks of the work list exploit more parallelism and avoid useless work. For BFS, the proportion of nodes in the frontier is always small. The topology-driven implementation allocates a great number of threads equal to the number of nodes in the graph, but only a few threads do the computation of search actually for each iteration, which decreases the efficiency of GPU. However, the data-driven implementation only needs to spawn a number of the threads equal to the number of neighbors of the nodes in the frontier and all the threads release their computing power to improve the performance. Therefore, in this paper, we select the data-driven implementation of BFS to study the prefetching mechanism.

Algorithm 2 The Topology-Driven Implementation of BFS

```
1:  function BFS_kernel(graph, curr_level)
2:      tid ← blockIdx.x * blockDim.x + threadId.x
3:      if graph.visitedlist[tid] == curr then
4:          start ← graph.vertexlist[tid]
5:          end ← graph.vertexlist[tid + 1]
6:          for i = start → end do
7:              vid ← graph.edgelist[i]
8:              if graph.visitedlist[vid] == INFINITY then
9:                  changed ← true
10:                 graph.visitedlist[vid] ← curr_level + 1
11:             end if
12:         end for
13:     end if
14: end function
```

C. COMPRESSED SPARSE ROW FORMAT GRAPH DATA STRUCTURE

Adjacent matrix is a popular data structure for graphs that GPU can efficiently operate on. Adjacent matrix stores the weight of the edge indexed by row index and column index. However, as the scale and sparsity of the graph increases, the memory space used to store the adjacent matrix grows explosively and most of the space stores useless information. For example, assuming a graph has 1 million nodes and 1 million edges and the data size is 4B, a adjacent matrix needs a 4TB storage space to store the whole graph and only 0.0001% of data are useful. Compressed sparse row format (CSR) graph data structure is one of the popular compressed graph data structures used to represent large and sparse graphs. CSR format represents a adjacent matrix with three arrays. The indices array stores column indices and the data array stores the nonzero values in the matrix. The third array stores the start of the row in column indices and data, called ptr. Therefore, the item in the matrix indexed by i row and j column can be accessed as data[ptr[i]+k] and k is the index of j in the indices[ptr[i]:ptr[i+1]]. Comparing with the adjacent matrix, CSR format only needs a 12MB space to store that graph.

The algorithm of BFS we use for evaluation is from the benchmark GraphBig [13] and it adopts CSR as the graph data structure, shown in Algorithm 1. According to the Algorithm 1, the graph data structure includes vertex list, edge list and visited list, and the vertex list and the edge list refer to the arrays of ptr and column indices in CSR respectively. The visited list stores the visiting information of each node and the work list is introduced by the data-driven implementation to store the vertex frontier of

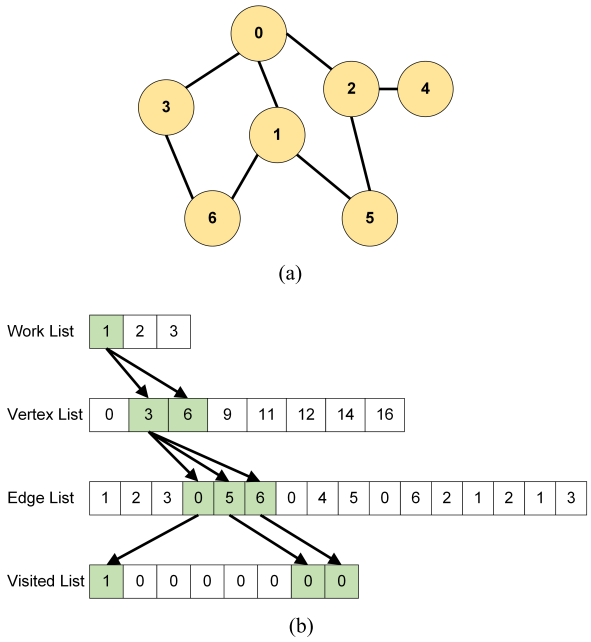

FIGURE 3. An example of graph data structure access pattern of the data-driven implementation of BFS. (a) An example abstract graph. (b) The order of accessing each data structure to complete a breadth first search for one node.

each iteration. Fig. 3 shows an example of graph data structure access pattern of the data-driven implementation of BFS. The order of accessing data structures to complete a breadth first search for one node is shown in Fig. 3(b). First, the warp fetches an item from the top of the work list to get the vertex ID of the node to be searched. Second, the warp accesses the vertex list to get the row length and the start index of the edge list. Third, threads in the warp process different edges in parallel to get the frontier of the next iteration. Lastly, threads check the value of each frontier node in the visited list and put the unvisited nodes in the work list. This process is executed iteration by iteration until the work list is empty.

According the well-defined graph data structure access pattern of BFS, what to access next is predictable upon the data structure currently accessed. Therefore, a prefetching mechanism with such an information can easily generate prefetching requests for BFS. Although the prefetching mechanism proposed in this paper aims at boosting BFS using CSR format, it still can be applied to other graph data structure easily. And for other graph analytics applications with the same access pattern of BFS, such as SSSP and GC, this prefetching mechanism also can improve their performance.

III. DATA-STRUCTURE AWARE PREFETCHING

BFS has a well-defined and predictable data structure access pattern. By using the knowledge of graph data structure and the access pattern of BFS, a prefetching mechanism could precisely fetch the necessary graph data in advance to reduce the long latency of irregular memory accesses to graph data structure. Also, according to the order of accessing graph data

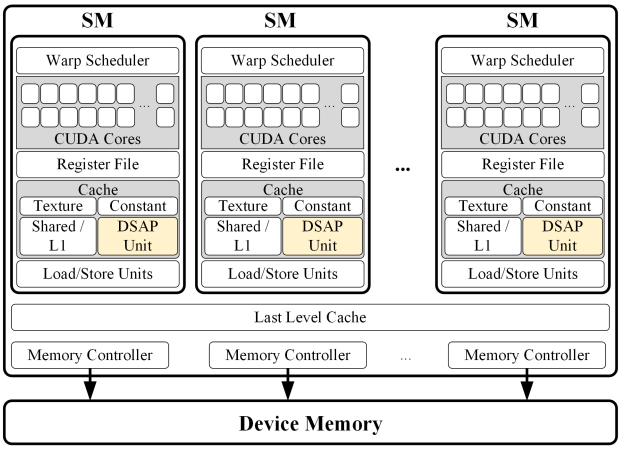

FIGURE 4. The integral GPU hierarchy detail with DSAP supports. Each SM contains a distributed DSAP unit.

structure of BFS, an adaptive fine-grain prefetching management can be applied to balance the efficiency of L1 cache and data prefetching. In this section, we propose a data-structure aware prefetching mechanism, called DSAP. First, we show the architecture overview of DSAP and how it works with other components in GPU. Second, we describe hardware supports for DSAP to generate prefetching requests to the graph data structure. Third, we introduce the adaptive fine-grain prefetching management to guarantee the efficiency of L1 cache and DSAP, and show how the hardware control logic works. Lastly, we introduce the software supports for DSAP with extra CUDA APIs.

A. OVERVIEW

A GPU contains many Streaming Multiprocessors (SMs) and every SM consists of many simple single-thread cores. In GPU, every 32 threads are grouped into a small unit, called warp, for scheduling. All the threads in the same warp execute the same instruction at the same time. For the data-driven implementation of BFS, because each warp needs to process a unique small chunk of the work list, the information of graph data structure for each warp and SM is different. Therefore, DSAP uses distributed DSAP units to generate prefetching requests for each SM. Fig. 4 shows the integral GPU hierarchy detail with the supports of DSAP. The on-chip memory of each SM consists of register file, texture cache, constant cache, L1 cache and shared memory. Texture cache and constant cache are read-only, which cannot be used to store prefetched data. Moreover, storing prefetched data into register file will exacerbate the resource contention. Shared memory has a sufficient space to store data, but it needs programming supports to manage the prefetched data, which may increase the complexity. Therefore, DSAP chooses L1 cache as the storage for the prefetched data and also L1 cache is responsible for processing prefetching requests.

Fig. 5 shows a local overview of how to generate prefetching requests and how to process these requests among

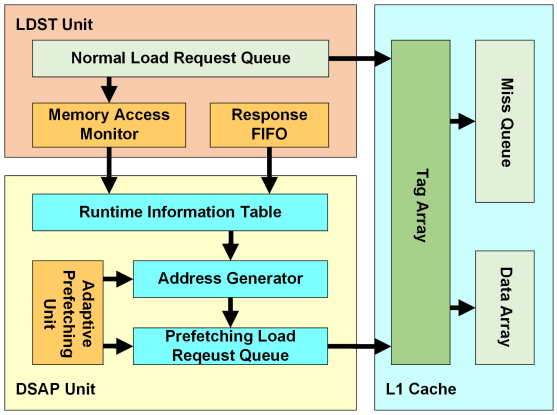

FIGURE 5. A local overview of how to generate prefetching requests and how to process these requests among Load/Store unit (LDST unit), L1 cache and DSAP unit.

Load/Store unit (LDST unit), L1 cache and DSAP unit. DSAP unit receives the information of memory accesses to the graph data structure from two components of the LDST unit, memory access monitor and response FIFO. The memory access monitor is responsible for monitoring normal loads to the work list. By monitoring loads to the work list, DSAP unit knows the new start of a search iteration and prepares to prefetch the graph data used in the next search iteration. The response FIFO stores the requested data and the information of the processed memory requests from L1 cache. Since the prefetching requests are also processed by L1 cache, the response FIFO can monitor the prefetching requests and send the requested data and information of the requests to DSAP unit. By using the information from LDST unit, DSAP unit can generate prefetching requests for graph data based on the graph data structure access pattern of BFS. After receiving the information from LDST unit, DSAP unit updates the entry of the runtime information table according to warp ID and chooses a corresponding request generator to generate prefetching requests based on the source of the received information (from the memory access monitor or the response FIFO) and the data structure the monitored load accesses. At last, DSAP unit puts the new generated prefetching requests into the prefetching request queue. The adaptive prefetching unit in DSAP unit is responsible for controlling the number of the prefetching requests generated. L1 cache processes both normal loads and prefetching loads, and treats prefetching loads as normal loads when processing.

B. HARDWARE SUPPORTS FOR DATA-STRUCTURE AWARE PREFETCHING

DSAP unit receives graph data information from two sources, the memory access monitor and the response FIFO, and chooses the request generator based on the source type of the information (from the memory access monitor or the response FIFO) and the data structure the monitored load accesses. The received information includes the requested address and data of the monitored memory request, the ID of the warp that issued this memory requests and the source type.

Fig. 6 shows the main components of the DSAP unit and how the prefetching mechanism works. There are four components: address space classifier, runtime information table, request generator unit which includes generators for each data structure and prefetching request queue. The address space classifier has two parts, address range table and eight comparators. The address range table stores the start and end addresses for each data structure. The comparators compare the requested address with the address range of each data structure in parallel and tell which data space the requested address is from. The runtime information table updates the entry of the graph data structure information with the requested data, according to the warp ID. The request generator unit has a generator selector and four types of request generators, and is responsible for generating prefetching requests to each graph data structure. The prefetching request queue stores all the generated prefetching requests and L1 cache fetches the prefetching requests when it is free.

1) GENERATOR SELECTION

Because the access pattern to each graph data structure is different, DSAP unit adopts four request generators to generate prefetching requests for each data structure. Generally, these generators can be grouped into two types, generator for the work list and generator for the other data structures (the vertex list, the edge list and the visited list). This is because the information they need to generate requests is from different components (the memory access monitor or the response FIFO). Generating the prefetching request to the work list is triggered by the normal load to the work list, which is monitored by the memory access monitor, while generating prefetching requests to other data structures is based on the prefetching requests to the graph data structure, which is monitored by the response FIFO.

Besides, DSAP unit uses the requested address of the monitored prefetching request to choose the generator for the vertex list, the edge list and the visited list. According to the graph data access pattern of BFS, the order of accessing each data structure is predictable. Therefore, when monitoring a prefetching request to the work list, DSAP unit can generate a prefetching request to the vertex list, while if the monitored prefetching request accesses the vertex list, prefetching requests to the edge list will be generated. Also, when monitoring the prefetching request to the edge list, DSAP unit will generate requests to the visited list. Therefore, both the source of the received information and the requested address determine which request generator to use.

For example, in Fig. 6, DSAP unit receives an information from L1 cache. The source type is 1, meaning this information is sent by the response FIFO. According to the requested address and the warp ID, this monitored request is a prefetching request to the vertex list from warp 0. Based on the value of the vertex index in the runtime information table for warp 0, which is 1, the address of data for the vertex 1

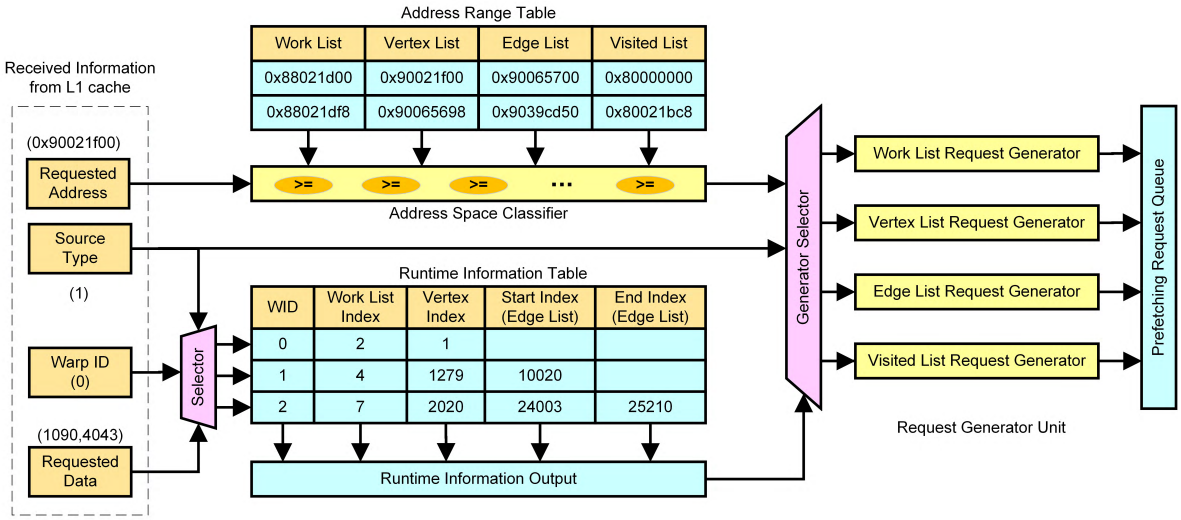

FIGURE 6. The main components of the DSAP unit and how the prefetching mechanism works.

is 0x90021f04. Since DSAP unit needs to get the values of the two adjacent nodes of the vertex list for the start index and end index of the edge list, DSAP unit reads out the two values (1090,4043) from the 128B requested data, the addresses of which are 0x90021f04 and 0x90021f08. The two values are then written in the runtime information table as the start index and end index for warp 0 and sent to the request generator unit to generate prefetching requests to the edge list.

2) REQUEST GENERATORS

Due to the differences of the access patterns between each data structure (the work list, the vertex list, the edge list and the visited list), DSAP unit adopts four prefetching request generators for each data structure.

The work list request generator is responsible for generating a prefetching request to the work list, when DSAP receives the information of a normal load to the work list. The requested address of the prefetching request is the address of the next item of this normal load requested in the work list. Therefore, the requested address is calculated with the address of the normal load requested and the size of the data in the work list. For example, if the normal load requests the data at the address of 0x88021d00, then the generator will generate a prefetching request to the address of 0x88021d04.

Two adjacent nodes in the vertex list point out the start and end edge index of the edges in the edge list, according to the CSR graph data structure. The previous prefetching request to the work list fetches the vertex ID. Therefore, the vertex list request generator can calculate the addresses of the two nodes in the vertex list, when the response FIFO sends back the information of the previous prefetching request to the work list. Usually, one memory request can fetch the data of the two nodes, when they are in the same cache line, while if the values are not in the same cache line, DSAP has to issue two memory requests. For example, in Fig. 6, according to the runtime information table, warp 1 only gets the start index from the monitored request. That means the values of the start index and the end index are not in the same cache line and DSAP unit is waiting for the other request to the vertex list.

The edge list request generator generates prefetching requests to the edge list based on the corresponding start and end edge index in the runtime information table. Since all the edges for one node is stored continuously in the edge list, the number of generated prefetching requests depends on how many cache lines could hold all the data and the case of unaligned addresses should be considered.

The visited list uses the data in the edge list as the vertex index. Therefore, the visited list request generator depends on the requested data of the previous prefetching requests to the edge list to generate request addresses of the visited list. Since the node IDs in the edge list are not adjacent, the visited list request generator needs to generate one memory request for each value of the requested data. That means if the cache line size is 128B and the data size is 4B, the visited list request generator needs to generate 32 memory requests to the visited list for each requested data of the edge list.

C. ADAPTIVE FINE-GRAIN PREFETCHING MANAGEMENT

Cache pollution and early eviction are common problems caused by prefetching. Many works tend to use metrics of the cache miss rate and early eviction rate to determine whether to do data prefetching or not. These strategies can quickly resume the efficiency of cache, but loss the performance improvement from data prefetching. Reducing the number of generated prefetching requests is an effective way to avoid cache pollution and early eviction. The graph data structure access pattern of BFS provides an opportunity to design an adaptive fine-grain prefetching management to control the number of generated prefetching requests. For BFS, the order of accessing the graph data structure is well-defined, which

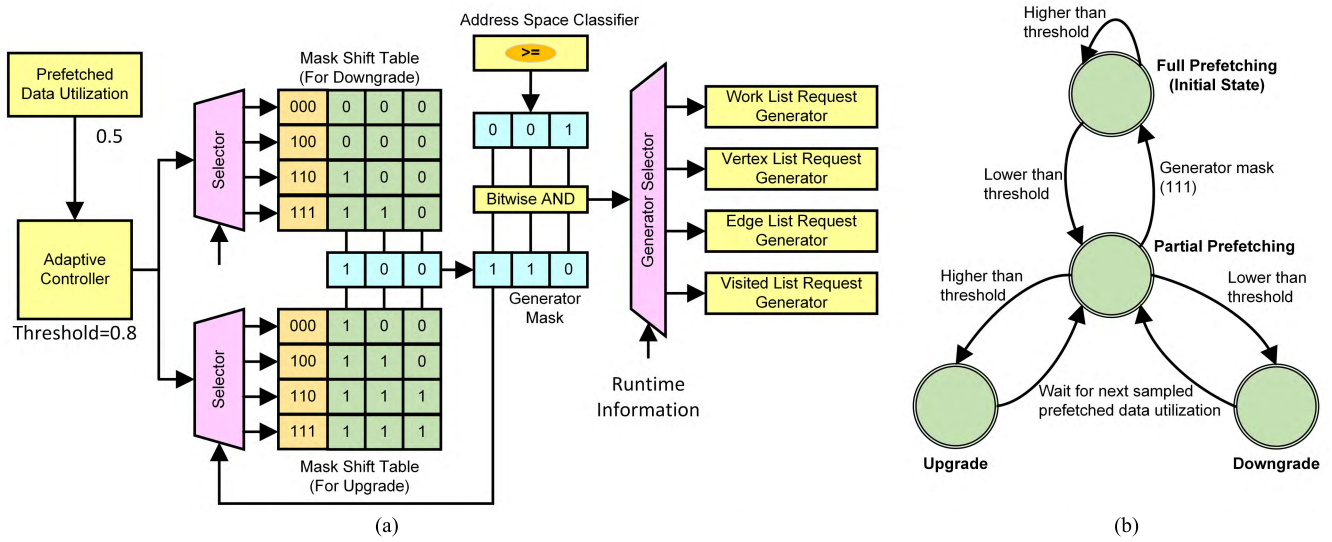

FIGURE 7. **Hardware design for the adaptive fine-grain prefetching management. (a) Hardware design of the adaptive fine-grain prefetching management. (b) Status transition of the adaptive conttroller.**

is the work list, the vertex list, the edge list and the visited list. Therefore, DSAP unit can have five statues for its prefetching granularity: no prefetching, prefetching to the work list, prefetching to the work list and the vertex list, prefetching to the work list, the vertex list and the edge list, and prefetching to the work list, the vertex list, the edge list and the visited list (full prefetching). Initially, each DSAP unit is at the full prefetching status. However, as long as the early eviction rate is higher than the threshold, DSAP unit can downgrade its prefetching status from the full prefetching to no prefetching gradually. And when the cache early eviction rate is lower than the threshold, DSAP unit also can upgrade its prefetching status dynamically.

Fig. 7(a) shows the hardware design of the adaptive fine-grain prefetching management. The prefetched data utilization is the ratio of the used prefetched data to all the prefetched data in L1 cache, which reflects the early eviction rate, and DSAP unit uses it as the metric of switching the status of the prefetching granularity. The adaptive controller periodically receives the prefetched data utilization from L1 cache and adjusts the prefetching status of DSAP unit. Fig. 7(b) represents the status transition of the adaptive controller. The partial prefetching represents the other three statuses of the prefetching granularity, except the prefetching to the work list, the vertex list, the edge list and the visited list, which is called the full prefetching, and no prefetching status. At the partial prefetching status, if the new received prefetched data utilization is higher than the threshold, the adaptive controller will upgrade the prefetching status to allow DSAP unit to generate more prefetching requests. Otherwise, the adaptive controller will downgrade the prefetching status.

The generator mask stores the status of the prefetching granularity and points out which request generators can generate prefetching requests at present status. What needs illustration is that the work list request generator is always allowed. Therefore, the three bits of the generator mask respectively refer to the vertex list, the edge list and the visited list. For example, in Fig. 7(a), the value of the generator mask, 110, means the vertex list request generator and the edge list request generator are allowed to generate prefetching requests, while the visited list request generator is not. The value of the generator mask and the result of the address space classifier do a bitwise AND operation to get the final mask for the generator selector. In Fig. 7(a), the result of the address space classifier is 001, which means the received information can be used to generate prefetching requests to the visited list. However, the generator mask blocks out the visited list request generator and the visited list request generator will not generate any prefetching requests at this status.

The value of the generator mask changes based on the current value of the generator mask, the control signal of the adaptive controller and two mask shift table. The adaptive controller issues an upgrade or downgrade signal to the corresponding mask shift table based on the status transition in Fig. 7(b). The mask shift table matches the entry with the current generator mask and reads out the corresponding value to update the value of the generator mask. For example, in Fig. 7(a), the current value of the generator mask is 110, but the sampled data utilization is 0.5 which is lower than the threshold (0.8). In this case, the adaptive controller issues a downgrade signal to the mask shift table (for downgrade) and reads out the corresponding value of the next generator mask (100) based on the value of the current generator mask (110). Then the value of the generator mask changes to 100, and that means only the prefetching to the work list and the vertex list is allowed at this status.

TABLE 1. Basic simulator configurations.

Number of SMs	15
Threads per SM	1536
Threads per warp	32
Warp scheduling policy	GTO
L1 cache size	48KB

D. EXTRA CUDA API SUPPORTS

DSAP provides two extra CUDA APIs, cudaMallocMark and cudaUpdateWl, to help the hardware get the static information of the graph data structure before launching kernels on SM cores. DSAP uses a modified cudaMalloc, cudaMallocMark, to mark the memory allocations for the graph data structure and send the allocated address spaces of the data arrays to the address range table. DSAP calls the cudaUpdateWl before entering the kernel to update the address range of the work list each time. Algorithm 3 shows an example of how to use the extra CUDA APIs to make DSAP work. The cudaMallocMark uses a parameter to specify the entry id of the address range table to store the range of the allocated addresses for each data array of the graph data structure. The cudaUpdateWl updates the address range of the work list based on the value of the variable wl_size every iteration.

IV. METHODOLOGY

We implement DSAP on GPGPU-sim simulator [14]. And the parameters of the simulator are based on GTX-480, shown in Table 1. For the L1 cache size, we choose the largest configurable size (48KB) for GTX-480. Also, we compare the performance of DSAP with a smaller L1 cache size (16KB) to show the impact of the cache size on DSAP. The BFS algorithm is selected from the GraphBig benchmark [13]. The datasets we test are chosen from the SNAP datasets [15] and their features are listed in Table 2. Moreover, we use the GPUWattch [16] and McPAT [17] integrated in GPGPU-sim simulator to estimate the power consumption and chip area of DSAP respectively.

Algorithm 3 CUDA API Supports for DSAP

```
1:  function main
2:      cudaMallocMark(worklist, sizeof(worklist), 0);
3:      cudaMallocMark(vertexlist, sizeof(vertexlist), 1);
4:      cudaMallocMark(edgelist, sizeof(edgelist), 2);
5:      cudaMallocMark(visitedlist, sizeof(visitedlist), 3);
6:      wl_size ← 1
7:      while wl_size! = 0 do
8:          cudaUpdateWl(wl_size);
9:          BFS Kernel function
10:     end while
11: end function
```

V. EVALUATION

A. PERFORMANCE

To demonstrate the effectiveness of DSAP, we compare the performance of DSAP with two prefetching mechanisms,

TABLE 2. The features of the tested datasets.

Name of Dataset	Abbrev.	Nodes	Edges
roadNet-CA	CA	1,965,206	27,766,607
USA-Road	USA	1,070,376	2,712,798
roadNet-TX	TX	1,379,917	1,921,660
Oregon-2	OR	11,461	32,731
Cit-HepPh	CIT	34,546	421,578
roadNet-PA	PA	188,092	1,541,898

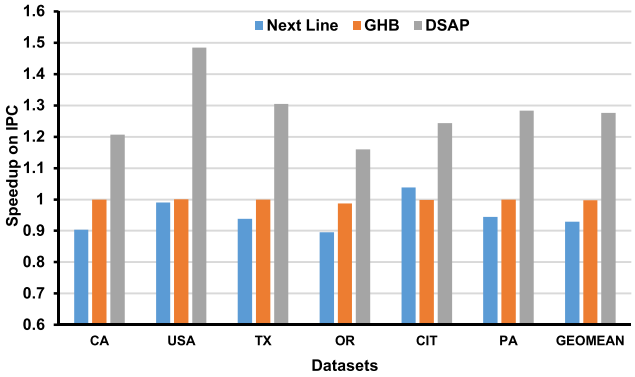

FIGURE 8. Speedups of the Next Line Prefetcher, the stride-based GHB prefetcher and the DSAP. The baseline is the performance of GPU without prefetchers.

a Next Line prefetcher and a stride-based GHB prefetcher standing for the stream prefetching and the combination of the stride prefetching and the GHB prefetching respectively. The Next Line prefetcher fetches data from the next cache line relative to the address of the last load. The stride-based GHB prefetcher uses GHB to maintain the global miss addresses and tries to find the constant stride accesses for the same PC. Both of the prefetchers benefit the memory accesses of BFS theoretically, because memory accesses to the work list and the edge list have some localities.

Fig. 8 compares the speedups on IPC of the Next Line prefetcher and the stride-based GHB prefetcher with those of DSAP and the baseline is the performance of GPU without prefetchers. In general, we test six datasets from three different applications, which are road networks, autonomous systems graphs and citation networks. DSAP improves the performance of IPC by 28% in geometric mean, while the Next Line prefetcher gets the worst performance for all the datasets and the stride-based GHB prefetcher is slightly worse than the GPU without prefetchers. The Next Line prefetcher aggressively prefetches the next cache line according to the last address of load, but these prefetched cache lines are useless for BFS and pollute the L1 cache. As a result, the Next Line prefetcher degrades the performance for nearly 8%. On the other hand, the stride-based GHB prefetcher cannot detect sufficient constant-strided access patterns from the memory accesses of BFS. Therefore, the stride-based GHB prefetcher does not issue any prefetching requests without a constant stride detected and its performance is nearly the same with GPU without prefetchers. On the contrary, DSAP can get a relative high coverage for the memory

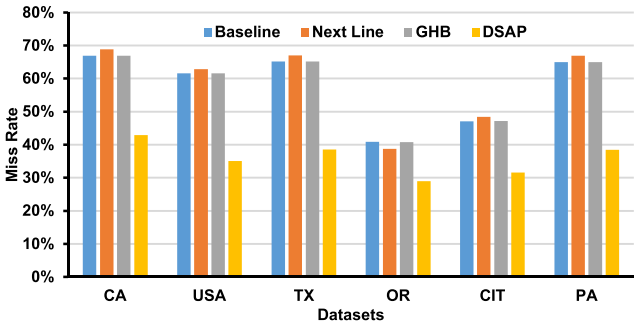

FIGURE 9. Miss rates of each prefeching mechanisms for each dataset.

accesses of BFS, which is nearly 60%. With the effective prefetching, DSAP gets the best improvement for 48.4% for the USA dataset. This demonstrates that DSAP is more effective than the typical prefetching mechanisms for BFS.

Fig. 9 shows the miss rates of L1 cache for the three prefetching mechanism. Obviously, DSAP greatly reduces the miss rates for all the datasets and the biggest decrease is 26.7%, while the smallest is 11.9%. This demonstrates that the big performance improvement of DSAP comes from reducing the miss rate of data accesses and the prefetching mechanism of DSAP can detect the memory access pattern of BFS and fetch useful graph data in advance. Fig. 10 compares the miss rates of accesses to each graph data structure of DSAP with those of the baseline GPU. In general, DSAP reduces the miss rate of the accesses to all the graph data structures compared with the baseline GPU. Specifically, due to the good locality, the miss rates reduction of the work list for all the datasets are the smallest compared with the other three data structures. Moreover, for the vertex list, the edge list and the visited list, the miss rate reduction of the vertex is the biggest, while the smallest is from the visited list. This is because in some cases, the threads have issued the normal loads to the visited list, but the corresponding prefetching requests to the visited list are still in flight in the memory pipeline. As a result, these loads cause the misses in L1 cache. However, even though the data are not fetched in time, prefetching requests also benefit the normal loads to the graph data structure, since the requests have been processed in advance.

The performance of DSAP varies for different datasets, shown in Fig. 8. For example, for the USA dataset, the speedup of DSAP reaches to 48.4%, while for the OR dataset, DSAP only is 16% faster. Two reasons cause the performance variety. The first reason is that the miss rates of each data structure for the OR dataset is much lower than those for the USA dataset. In Fig. 10, the miss rates of the vertex list, the edge list and the visited list for the OR dataset are 40.8%, 67.5% and 55.2%, while the miss rates for the USA dataset are 82.6%, 90.9% and 81.8% respectively. The second reason is that the miss rate reductions of each data structure for the USA dataset is much larger than those for the OR dataset. From Fig. 10, the miss rate reductions of each data structure for the USA dataset are 73.78%, 64.49% and 41.81%, while the miss rate reductions for the OR dataset are 16.51%, 24.9% and 17.66% respectively.

From the above results, we conclude that DSAP can detect the memory access pattern of BFS with the knowledge of the data structure access pattern of BFS, reduce the cache miss rate and achieve an obvious performance improvement. Fig. 11 depicts the increase of the memory bandwidth of GPU and the utilization of the prefetched data after introducing the data prefetching of DSAP. For all the datasets, the memory bandwidth increases no more than 7% and at least 75% of prefetched data benefit BFS.

Fig. 12 and Fig. 13 compares the performance improvement and the L1 cache miss rates of DSAP with and without the adaptive fine-grain prefetching management for each dataset. The difference between DSAP with and without the adaptive fine-grain prefetching management is that DSAP without the adaptive fine-grain prefetching management stops generating any prefetching requests when the utilization of the prefetched data is lower than the threshold, while DSAP with the adaptive fine-grain prefetching management can continue to generating prefetching requests, but the requests to some data structures are banned. From the results in Fig. 12, DSAP with the adaptive fine-grain prefetching management performs better than DSAP without the adaptive fine-grain prefetching management for an average of 2%. The miss rates of each data structure in Fig. 13 show that DSAP with the adaptive fine-grain prefetching management achieves lower cache miss rates, especially for the vertex list and the edge list. The adaptive fine-grain prefetching management not only prefetches more data, but also keeps the utilization of the prefetched data at a relative high level, which makes DSAP balance the performance and cache efficiency effectively.

To estimate the power consumption of DSAP, we use the GPUWattch [16], a tool integrated with GPGPU-sim simulator, to calculate the increment of power and the results are shown in Fig. 14. The results show that the power of the whole GPU when running BFS for different datasets increases slightly for an average of 8.3% and up to 11.8%. Also, we calculate the energy the whole GPU spends, and it shows that due to the big performance improvement on IPC, the energy costs for all the datasets are decreased. Fig. 14 shows that the energy cost decreases 15.1% in average and up to 27%.

B. SENSITIVITY STUDY

In this section, we test DSAP with some important factors. The cache volume determines how many cache lines can be prefetched and stored in the L1 cache. The threshold of the adaptive fine-grain prefetching management affects the efficiency and flexibility of DSAP. The scheduling policy of GPU considers the commonality of DSAP.

1) CACHE VOLUME

The cache volume limits the performance of DSAP. Since DSAP brings the necessary data of the next iteration for

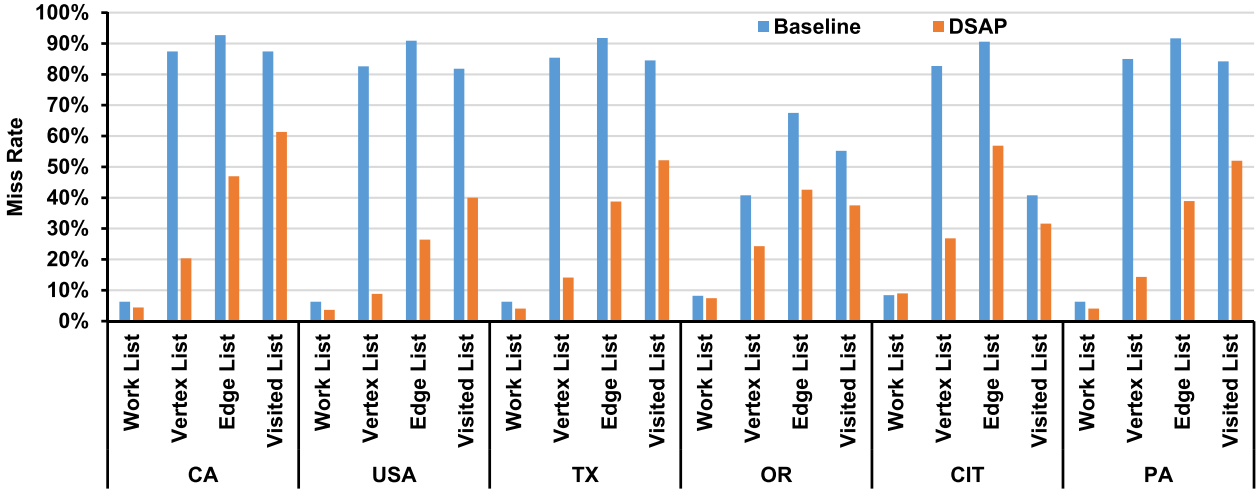

FIGURE 10. The miss rates of accesses to each graph data structure for DSAP and the baseline GPU.

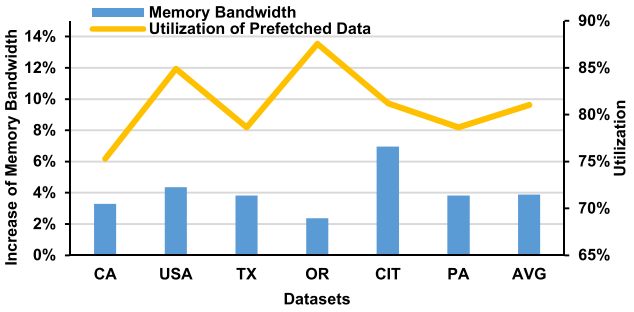

FIGURE 11. The increase of the memory bandwidth when using DSAP and the utilization of prefetched data for each dataset.

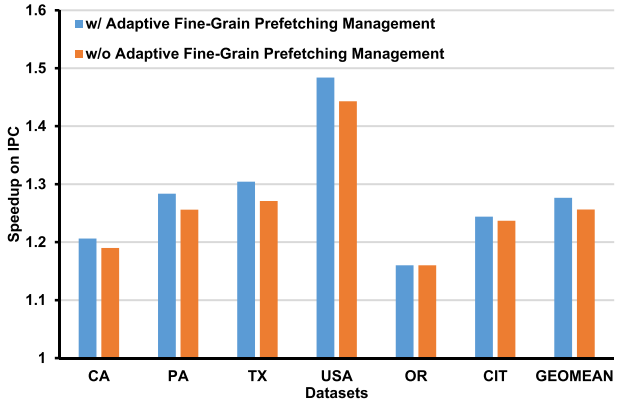

FIGURE 12. The speedups of DSAP with and without adaptive fine-grain prefetching management for each dataset.

every warp in advance, the contention for the on-chip memory space becomes more severe. And as a result, DSAP has to prefetch fewer cache lines to reduce the early eviction rate. Fig. 15 shows the speedups of DSAP with two different L1 cache volumes. For all the datasets, the small cache volume (16KB) degrades the performance of DSAP for 20% in average, compared with the large cache volume (48KB). There are two possible solutions. One solution is to increase the size of on-chip memory, but it will increase the hardware cost as well. Another solution is to adopt the fine-grain cache management for the L1 cache. The sizes of the data for each memory access to the vertex list and the visited list are 4B or 8B and far smaller than the size of L1 cache line (128B). Also, the accesses to the vertex list and the visited list have few localities, therefore most of the data in the cache line will be wasted. Fine-grain cache management allows data from different cache lines to be stored in the same L1 cache line by splitting the large L1 cache line into small chunks. Many works have researched fine-grain cache management and we have proposed a dynamic multi-grain cache management for general purpose GPU applications with irregular memory accesses. Since it is out of scope of this paper, more details can be found in [18].

2) THE THRESHOLD FOR THE ADAPTIVE FINE-GRAIN PREFETCHING MANAGEMENT

Fig. 16 shows the speedups of DSAP with different thresholds for the adaptive fine-grain prefetching management. UT means the Utilization Threshold. DSAP periodically receives the utilization of the prefetched data from L1 cache and the adaptive fine-grain prefetching management changes the status of the generator mask according to the data utilization, which is described in Section III. From the results, the performance gaps among the different utilization thresholds for each dataset are not obvious, except the CIT dataset. The difference between the best and the worst for the CIT dataset is only 3%. This demonstrates the adaptive fine-grain prefetching management can make a good balance between the data prefetching and the cache efficiency and make the performance change smoothly when the utilization of the prefetched data varies.

3) SCHEDULING POLICY

Fig. 17 shows the performance of DSAP when GPU uses the Loose Round-Robin scheduling policy (LRR) and the

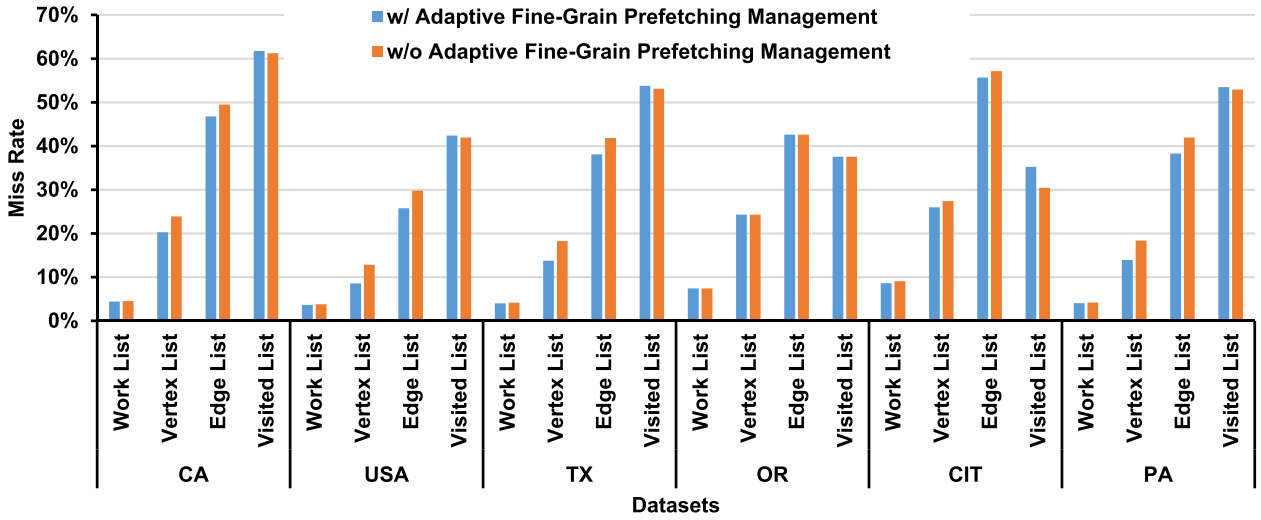

FIGURE 13. The miss rates of DSAP with and without adaptive fine-grain prefetching management for each dataset.

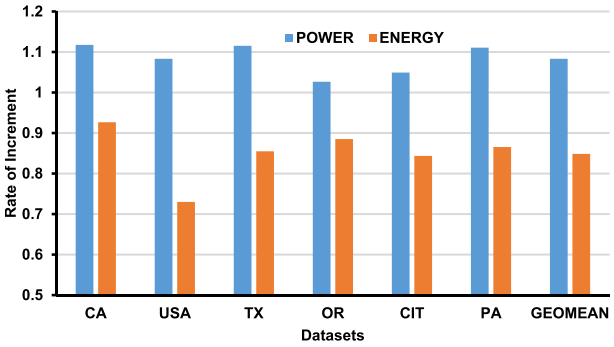

FIGURE 14. The rates of power and energy increment for GPU with DSAP for each dataset.

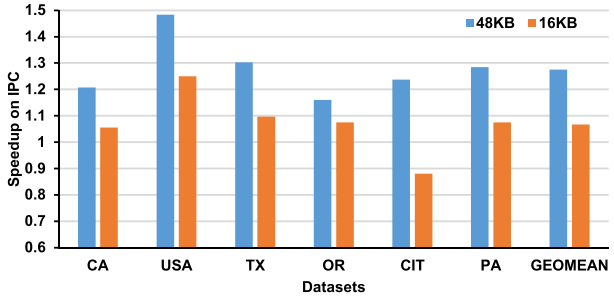

FIGURE 15. Speedups of DSAP with different cache volumes for each dataset.

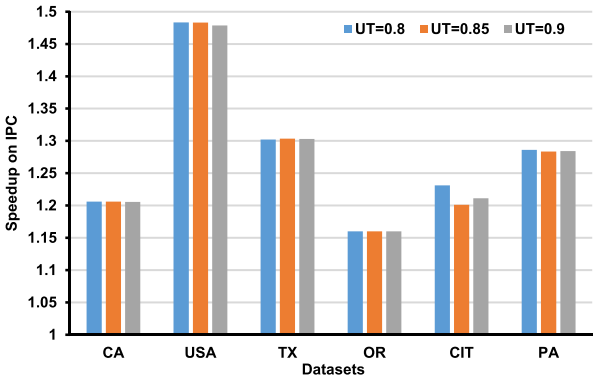

FIGURE 16. Speedups of DSAP with different thresholds for the adaptive fine-grain prefetching management.

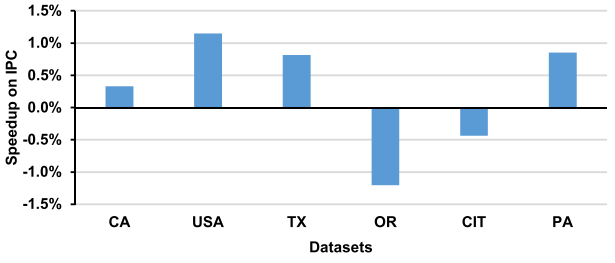

FIGURE 17. Speedups of DSAP when GPU adopts Loose Round-Robin scheduling policy. The baseline is that GPU uses GTO scheduling policy.

baseline is that GPU uses the Greedy Then Oldest scheduling policy(GTO). From the results, scheduling policies have different impacts on different datasets, but the impacts of scheduling policies are negligible. For all the datasets, the fluctuations of the performance for different scheduling policy are between −1.5% and 1.5%. Therefore, it demonstrates that the performance of DSAP does not depend on the scheduling policies of GPU.

C. HARDWARE COST

Because DSAP does not have complex computations, its arithmetical logic is very simple. The biggest cost for DSAP is the on-chip storage for saving the information of the graph data structure and runtime statuses. For basic BFS, the address range table needs 8 64-bit registers and each entry of the runtime information table costs 36B. Since GTX-480 supports at most 48 warps per SM core, the runtime

information table needs an extra 1728B memory space and costs only 2.6% of the on-chip shared memory/L1 cache. Therefore, the extra memory space can be allocated in shared memory easily. Even if GPU supports more warps per core in future, we can introduce an allocating mechanism to control the size of the runtime information table. Also, we estimate the area cost of the DSAP unit for each SM by using the McPAT [17]. The chip area cost slightly increases 0.53% for each SM, which is a moderate cost in comparison to the big performance improvement.

VI. RELATED WORKS

A. GPGPU PREFETCHING

Sethia *et al.* [19] propose a per-warp stride prefetching to reduce the power consumption of GPUs. Lee *et al.* [7] propose a many-thread aware prefetching mechanism based on the stride prefetching. Lakshminarayana and Kim [20] propose a prefetching mechanism for graph applications. And it stores the prefetched data in the register files to reduce the overhead in caches. Ryoo *et al.* [21] demonstrate that prefetching into registers can benefit the performance by binding operations. Also, Yang *et al.* [22] introduce a compiler method for prefetching into registers and several compiler optimizations for applications. Sadrosadati *et al.* [23] propose a two-level hierarchical register structure to reduce the long latency and the high power consumption of register files by prefetching the register working-set. Kim *et al.* [24] present a warp pre-execution method that a warp can pre-execute successive independent instructions in P-mode, while waiting for long-latency operation and the P-mode results are stored in renamed registers. Neves *et al.* [25] design a low-profile prefetcher for L1 caches with a data-pattern description specification. Michelogiannakis and Shalf [26] propose a prefetcher for the last-level cache that detects the correlated access patterns of data parallel applications.

B. ADAPTIVE PREFETCHING

Many prior works use the prefetching accuracy as the feedback metric to adjust prefetching. Dahlgren *et al.* [27] adjust the prefetching distance by measuring the prefetching accuracy. Srinath *et al.* [28] introduce the cache pollution effect and the timeliness to control the impact of prefetching on cache. Some researches focus on managing the priority of prefetching requests. Zhuang and Lee [29] design a cache pollution filter for both hardware and software prefetching to classify prefetching requests. Ebrahimi *et al.* [30] introduce a prefetch coordination mechanism to consider both local and global feedbacks to maximize the benefits of prefetching. Lee *et al.* [31] propose a prefetch-aware DRAM controller to prioritize between demand and prefetching requests. Caragea *et al.* [32] propose a software prefetcher to adjust the distance of prefetching requests based on the number of MSHR entries. Liu *et al.* [33] propose a self-tuning prefetcher to adjust prefetching modes dynamically with the runtime feedback. DSAP adopts the adaptive fine-grain prefetching management to control the prefetching of the graph data structure depending on the utilization of the prefetched data and the efficiency of the L1 cache.

C. CACHE UTILIZATION

Chen *et al.* [34] introduce an adaptive management techniques combining the warp throttling and cache bypassing. Tian *et al.* [35] propose a novel technique to determine the cache bypassing for each static load. Wang *et al.* [36] propose DaCache, consisting of a warp scheduling technique, a cache replacement policy and a cache bypassing method. Komuravelli *et al.* [37] introduce a new local memory that can address globally and reuse data implicitly. To reduce the overheads of prefetching, DSAP does not modify the cache and memory structure. DSAP improves the efficiency of L1 cache and the prefetched data by dynamically controlling the status of the prefetching granularity.

VII. CONCLUSION

We propose a data structure-aware prefetching mechanism for BFS on GPU to prefetch the necessary data into the L1 cache and reduce the latency of irregular memory accesses. Typical prefetching mechanisms are not able to detect the access patterns from the irregular memory accesses of BFS and perform effectively by prefetching. DSAP generates prefetching requests based on the explicit information of the graph data structure and the graph data structure access pattern of BFS. Considering the relative small on-chip memory space, DSAP adopts the adaptive fine-grain prefetching management to maintain the effeciency of L1 cache and the data prefetching. The adaptive fine-grain prefetching management unit monitors the utilization of the prefetched cache lines to adjust the status of the prefetching granularity dynamically. Our experiment demonstrates that DSAP can achieve a 28% geomentric mean performance improvement for BFS for six datasets from three different kinds of applications with an average of 8.3% power increment and 15.1% energy cost reduction.

REFERENCES

[1] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Annu. Int. Symp. Comput. Archit.*, May 1990, pp. 364–373.
[2] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," *ACM SIGARCH Comput. Archit. News*, vol. 22, no. 2, pp. 24–33, 1994.
[3] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, no. 5, pp. 609–623, May 1995.
[4] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newslett.*, vol. 23, nos. 1–2, pp. 102–110, 1992.
[5] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2004, p. 96.
[6] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: An adaptive data cache prefetcher," in *Proc. 13th Int. Conf. Parallel Archit. Compilation Techn.*, Oct. 2004, pp. 135–145.
[7] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU applications," in *Proc. IEEE 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2010, pp. 213–224.

[8] Y. Oh *et al.*, "APRES: Improving cache efficiency by exploiting load characteristics on GPUs," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 191–203, 2016.

[9] A. Jog *et al.*, "OWL: Cooperative thread array aware scheduling techniques for improving GPGPU performance," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 395–406, 2013.

[10] A. Jog *et al.*, "Orchestrated scheduling and prefetching for GPGPUs," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 332–343, 2013.

[11] G. Koo, H. Jeon, Z. Liu, N. S. Kim, and M. Annavaram, "CTA-aware prefetching and scheduling for GPU," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2018, pp. 137–148.

[12] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on GPUs," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process. (IPDPS)*, May 2013, pp. 463–474.

[13] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2015, pp. 1–12.

[14] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2009, pp. 163–174.

[15] J. Leskovec and A. Krevl. (Jun. 2014). *SNAP Datasets: Stanford Large Network Dataset Collection*. [Online]. Available: http://snap.stanford.edu/data

[16] J. Leng *et al.*, "GPUWattch: Enabling energy optimizations in GPGPUs," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 487–498, 2013.

[17] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2009, pp. 469–480.

[18] H. Guo, L. Huang, Y. Lü, S. Ma, and Z. Wang, "DyCache: Dynamic multi-grain cache management for irregular memory accesses on GPU," *IEEE Access*, vol. 6, pp. 38881–38891, 2018.

[19] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke, "APOGEE: Adaptive prefetching on GPUs for energy efficiency," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.* Piscataway, NJ, USA: IEEE Press, Sep. 2013, pp. 73–82.

[20] N. B. Lakshminarayana and H. Kim, "Spare register aware prefetching for graph algorithms on GPUs," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 614–625.

[21] S. Ryoo *et al.*, "Program optimization space pruning for a multithreaded GPU," in *Proc. 6th Annu. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2008, pp. 195–204.

[22] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," *ACM SIGPLAN Notices*, vol. 45, no. 6, pp. 86–97, 2010.

[23] M. Sadrosadati *et al.*, "LTRF: Enabling high-capacity register files for GPUs via hardware/software cooperative register prefetching," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2018, pp. 489–502.

[24] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 163–175.

[25] N. Neves, P. Tomás, and N. Roma, "Stream data prefetcher for the GPU memory interface," *J. Supercomput.*, vol. 74, no. 6, pp. 2314–2328, 2018.

[26] G. Michelogiannakis and J. Shalf, "Last level collective hardware prefetching for data-parallel applications," in *Proc. IEEE 24th Int. Conf. High Perform. Comput. (HiPC)*, Dec. 2017, pp. 72–83.

[27] F. Dahlgren, M. Dubois, and P. Stenstrom, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 7, pp. 733–746, Jul. 1995.

[28] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *Proc. IEEE 13th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2007, pp. 63–74.

[29] X. Zhuang and H.-H. S. Lee, "A hardware-based cache pollution filtering mechanism for aggressive prefetches," in *Proc. Int. Conf. Parallel Process.*, Oct. 2003, pp. 286–293.

[30] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2009, pp. 316–326.

[31] C. J. Lee, O. Mutlu, V. Narasiman, and Y. N. Patt, "Prefetch-aware dram controllers," in *Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchitecture*, Nov. 2008, pp. 200–209.

[32] G. C. Caragea, A. Tzannes, F. Keceli, R. Barua, and U. Vishkin, "Resource-aware compiler prefetching for many-cores," in *Proc. IEEE 9th Int. Symp. Parallel Distrib. Comput. (ISPDC)*, Jul. 2010, pp. 133–140.

[33] P. Liu, J. Yu, and M. C. Huang, "Thread-aware adaptive prefetcher on multicore systems: Improving the performance for multithreaded workloads," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 13, no. 1, 2016, Art. no. 13.

[34] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu, "Adaptive cache management for energy-efficient GPU computing," in *Proc. 47th Annu. IEEE/ACM Int. Symp. microarchitecture*, Dec. 2014, pp. 343–355.

[35] Y. Tian, S. Puthoor, J. L. Greathouse, B. M. Beckmann, and D. A. Jimenez, "Adaptive GPU cache bypassing," in *Proc. 8th Workshop Gen. Purpose Process. GPUs*, 2015, pp. 25–35.

[36] B. Wang, W. Yu, X.-H. Sun, and X. Wang, "DaCache: Memory divergence-aware GPU cache management," in *Proc. 29th ACM Int. Conf. Supercomput.*, 2015, pp. 89–98.

[37] R. Komuravelli *et al.*, "Stash: Have your scratchpad and cache it too," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 707–719.

HUI GUO received the B.S. and M.S. degrees in computer science and technology from the National University of Defense Technology, China, in 2011 and 2014, respectively, where he is currently pursuing the Ph.D. degree in computer science and technology. From 2015 to 2017, he was a Visiting Student with the University of Michigan, Ann Arbor, MI, USA. His research interests include computer architecture, heterogeneous computing, and SIMD architecture.

LIBO HUANG received the B.S. and Ph.D. degrees in computer engineering from the National University of Defense Technology, China, in 2005 and 2010, respectively. He is currently an Associate Professor with the School of Computer, National University of Defense Technology. He has authored over 50 papers in internationally recognized journals and conferences. His research interests include computer architecture, hardware/software co-design, VLSI design, and on-chip communication.

YASHUAI LÜ received the Ph.D. degree in computer architecture from the National University of Defense Technology in 2009. He is currently with Space Engineering University, China. He has authored over 20 papers in internationally recognized journals and conferences. His main research interests include processor architecture and computer graphics.

JIANQIAO MA received the B.S. degree in software engineering from the National University of Defense Technology, China, in 2016, where he is currently pursuing the M.S. degree in computer science and technology. His research interests include computer architecture, big data, and branch prediction.

CHENG QIAN received the B.S. and master's degrees in computer science and technology from the National University of Defense Technology (NUDT), Changsha, China, in 2012 and 2014, respectively, where he is currently pursuing the Ph.D. degree. His research interests include 3-D memory architecture design and memory access scheduling mechanism.

SHENG MA received the B.S. and Ph.D. degrees in computer science and technology from the National University of Defense Technology (NUDT) in 2007 and 2012, respectively. He visited the University of Toronto from 2010 to 2012. He is currently an Assistant Professor with the School of Computer, NUDT. He has authored over 30 papers in internationally recognized journals and conferences. His research interests include on-chip networks, SIMD architectures, and arithmetic unit designs.

ZHIYING WANG (M'02) received the Ph.D. degree in electrical engineering in computer science and technology from the National University of Defense Technology, Hunan, China, in 1989. He is currently the Deputy Dean and a Professor of computer engineering with the Department of Computer, National University of Defense Technology. He has contributed 10 invited chapters to book volumes, published 200 papers in archival journals and refereed conference proceedings, and delivered over 30 keynotes. His current research projects include asynchronous microprocessor design and nanotechnology circuits and systems based on optoelectronic technology and virtual computer system. His main research fields include computer architecture, computer security, VLSI design, reliable architecture, multi-core memory system, and asynchronous circuit. He became a member of the ACM in 2003.