IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 1, 2018, accepted March 11, 2018, date of publication March 22, 2018, date of current version July 30, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2818193

DyCache: Dynamic Multi-Grain Cache Management for Irregular Memory Accesses on GPU

HUI GUO[1], LIBO HUANG[1], YASHUAI LÜ[2], 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 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 GPU utilizes the wide cache-line (128B) on-chip cache to provide high bandwidth and efficient memory accesses for applications with regularly-organized data structures. However, emerging applications exhibit a lot of irregular control flows and memory access patterns. Irregular memory accesses generate many fine-grain memory accesses to L1 data cache. This mismatching between fine-grain data accesses and the coarse-grain cache design makes the on-chip memory space more constrained and as a result, the frequency of cache line replacement increases and L1 data cache is utilized inefficiently. Fine-grain cache management is proposed to provide efficient cache management to improve the efficiency of data array utilization. Unlike other static fine-grain cache managements, we propose a dynamic multi-grain cache management, called DyCache, to resolve the inefficient use of L1 data cache. Through monitoring the memory access pattern of applications, DyCache can dynamically alter the cache management granularity in order to improve the performance of GPU for applications with irregular memory accesses while not impact the performance for regular applications. Our experiment demonstrates that DyCache can achieve a 40% geometric mean improvement on IPC for applications with irregular memory accesses against the baseline cache (128B), while for applications with regular memory accesses, DyCache does not degrade the performance.

INDEX TERMS Accelerator architectures, cache memory, fine-grain cache management, GPGPU computing, irregular memory access, memory divergence, memory management.

I. INTRODUCTION

GPU utilizes the wide cache line (128B) on-chip cache to provide high bandwidth memory accesses in order to satisfy the demand of data for the massive parallel computing. For each memory access instruction, the Load/Store (LDST) unit in each CUDA core generates 32 memory requests simultaneously, due to the the single-instruction-multiple-data (SIMD) architecture and the warp execution model. And if the threads in each warp request continuous data in the same 128B cache line, the LDST unit will combine these memory requests into one 128B memory request to reduce the number and overheads of memory accesses to on-chip cache. By coalescing memory requests, GPU can provides sufficient memory bandwidth for each CUDA core when data structure of applications are well organized.

Currently, emerging applications [1], [2], such as Graph Computing, Deep Learning, Neutral Network, etc., have been researched extensively. And GPU has been used to accelerate these applications in some scenarios. However, some of these applications use more complex and irregular data structures and exhibit irregular control flows and memory access patterns. To be specific, for these irregular applications, accessed data are scattered in a wide range of address space and the LDST unit has to generate more than one memory request to fetch all the demand data of one memory access instruction. However, due to the lock-step design, GPU can continue executing next instruction only when all the requests of a memory access instruction are completed. If these requests have different access latencies, especially some requests cause cache misses, we call this situation memory divergence. When all

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

the active warps suffer the memory divergence, GPU has to stall all the pipelines and its throughput drops dramatically, even though GPU is latency-insensitive.

Besides, due to the memory divergence, the size of accessed data becomes smaller than the wide cache line size (128B), hence irregular memory accesses fetch many cache lines with a lot of unwanted data. As a result, some words in a certain cache line are never being read or written before the cache line is evicted. In the other hand, since the number of cache line allocated to each warp is very limited, memory divergence will increase the cache miss rate and cause high frequent cache line replacement, and then consume off-chip bandwidth with the unwanted data.

Prior studies have shown that due to irregular memory accesses, such as gather/scatter access, pointer chasing, etc., coarse-grained memory accesses waste a great portion of bandwidth and degrade the efficiency of L1 cache [3]–[5]. Researches on increasing cache efficiency for fine-grain memory accesses focus on implementing fine-grain cache managements to improve the data array utilization of cache. However, these prior works either only support one small granularity, or use static cache managements, which are not transparent to users. Moreover, considering the variety between and in applications, we need a flexible fine-grain or multi-grain cache management to make L1 cache much more efficient for all kinds of applications.

In this paper, we introduce a dynamic multi-grain cache management design, called DyCache. By splitting the wide cache line into small chunks, DyCache can store small words in non-continuous memory space in the same wide cache line. And DyCache retains the wide cache line design to allow applications with regular memory accesses to take advantages of it. DyCache also supports multiple granularities in order to dynamically find the most appropriate cache management granularity for various applications and improve the cache efficiency according to the memory access pattern. To figure out the memory access pattern, we profile memory access patterns and cache statistics to evaluate the efficiency of L1 data cache and memory divergence in the runtime, and then choose an appropriate cache management granularity based on the memory access pattern. To summarize our most important contributions:

- This paper provides an analysis of memory divergence and inefficient use of L1 data cache in GPU architecture. We find that the mismatching between the size of accessed data and wide cache line is the primary cause of inefficient caching and poor performance for irregular memory accesses.
- Unlike prior works on static fine-grain cache managements, we propose a dynamic multi-grain cache line management that resolves inefficient use of L1 data cache and improves the throughput of GPU for applications with irregular memory accesses.
- We demonstrate that DyCache does not degrade the performance of applications with regular memory accesses, while improving cache efficiency and GPU throughput for irregular applications.

II. MOTIVATION

A. MEMORY DIVERGENCE AND INEFFICIENT UTILIZATION OF L1 CACHE

For a regular memory access, the memory access instruction, such as a load or store instruction, can fetch all the demand data of the 32 threads in a warp by a single 128B memory request, if all the threads access words which are in the continuous 128B memory space. Moreover, the LDST unit in GPU has a memory coalescing unit to combine memory accesses. This greatly reduces memory access overheads and improves the efficiency of memory bandwidth. However, as applications become more and more various, data structure is increasingly complicated and irregularly organized. This induces that LDST unit has to generate more than one memory request to fetch all the demand data, because the accessed data are non-continuous in memory space and in different 128B cache lines. These memory requests have different access latencies and even some of them may cause cache misses. We call this situation memory divergence and like branch divergence, memory divergence decreases GPU throughput because GPU's memory hierarchy can't handle these irregular accesses efficiently.

Since L1 cache is the first level storage to service memory requests from CUDA cores, the efficiency of L1 cache utilization determines the efficiency of the whole GPU memory hierarchy. L1 cache can provide high bandwidth accesses by using the wide cache line (128B), when 32 threads of a warp request a 128B-data that is continuous in memory space. However, when suffering memory divergence, GPU issues fine-grain memory requests, which causes the inefficiency that the demand data of a request is only a small fraction words in the fetched cache line. Also, another impact is that more than one cache line needs to be fetched for the diverged memory access. Due to the limited cache lines allocated to each warp, the possibility of being evicted of each cache line increases and even worse, some cache lines are evicted before they are referenced [6]. Therefore, memory divergence makes L1 cache inefficient, due to the contradiction between fine-grain demand data and coarse-grain cache line. Li *et al.* [5] uses the ratio of the number of referenced 32B words to the number of evicted 32B chunks to measure cache efficiency. Their result shows that for all 32 benchmark the arithmetic mean of cache efficiency is 58%. As the result of inefficient caching, a mass of memory requests are sent to off-chip memory and memory channels are congested by these requests.

B. FINE-GRAIN CACHE MANAGEMENT

Emerging applications, like graphics, have irregular memory access patterns. Modern GPU suffers the significant inefficiency in cache and off-chip memory system. Fine-grain cache management is a solution to eliminate the

contradiction between the fine-grain demand data and the coarse-grain cache line and improve the cache efficiency. Fine-grain cache management uses extra bits to maintain valid and dirty statuses for the fine cache granularity rather than the whole cache line. Sector cache [7] splits each cache line into multiple sectors (32B) so that words (32B) in continuous memory space can be stored. It provides extra valid and dirty bits for each word to manage at the small granularity. Based on Sector Cache, Seznec [8] and Rothman and Smith [9] propose elastic mapping between sectors and tags to reduce the miss rate. And Veidenbaum *et al.* [10] proposes the adaptive line size cache that can alter the cache line size gradually. Elastic-Cache is the most recently proposed fine-grain cache management for GPU. It uses unused shared memory to store tags for chunks of each cache line, so that the cache size does not shrink. Also, it allows chunks to store words (32B, 64B) from non-continuous memory space in the same cache line, which is different from Sector Cache. Their design shows great potential of fine-grain cache managements to make GPU L1 cache more efficient for application with irregular memory accesses.

C. PROBLEMS OF STATIC FINE-GRAIN CACHE MANAGEMENT

Although Elastic-Cache and other fine-grain cache managements can improve the L1 Cache efficiency and GPU throughput for applications with irregular memory accesses. However, there are some problems to be solved before fine-grain cache management becomes practical.

First, for a static fine-grain cache management, it depends on the user who programs applications or configures GPU to achieve the best performance. Due to the static configuration, users should configure GPU or annotate in code to set up the right cache granularity before running applications. Therefore, users should not only know GPU architecture well, but also get the full knowledge of memory access patterns for each application. In this paper, we are pursuing a more flexible cache management that automatically and dynamically chooses cache granularity according to the runtime memory access pattern.

Second, the memory access pattern of an application may vary with computing phases. For example, an application executes a matrix addition to get a matrix for data indexing in the next phase. The access pattern to the matrix is regular and coarse-grain. In the next phase, the application uses the result matrix indexes to access other data. For index-based indirect memory accesses, these accesses will be irregular and fine-grain. Fig. 1 shows the variations of the ratio of memory access requests accessing 32B, 64B and 128B data along with time for two representative applications, Needleman-Wunsch and StreamCluster. We find that for Needleman-Wunsch, the distribution of accessed data sizes varies along with application execution dynamically, while StreamCluster has a relative stable distribution of accessed data sizes. These two applications represent two memory access patterns. Static fine-grain cache management can improve cache efficiency

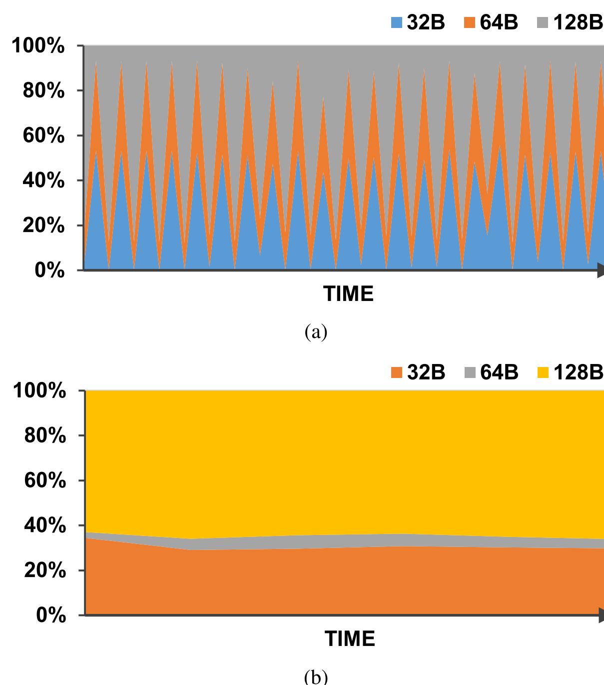

FIGURE 1. The distribution of accessed data sizes varies with time differently between applications. (a) Needleman-Wunsch. (b) StreamCluster.

for applications with access patterns like StreamCluster, but it cannot work efficiently for those like Needleman-Wunsch.

Therefore, in this paper, we aim to find a dynamic multi-grain cache management that can be flexible to adjust cache granularity based on the memory access pattern.

III. DYNAMIC MANAGEMENT OF MULTI-GRAINED CACHE

As we have discussed above, it is difficult to fully utilize GPU's massive parallel computing power for applications with irregular memory accesses due to memory divergence. One of the impact of memory divergence is that L1 data cache performs much less efficiently than it does when GPU executes the kernels using well-organized data structure. Also, due to the high miss rate, L1 cache sends a mass of fine-grain requests to lower-level memory, which puts a huge burden on the off-chip memory and degrades throughput of the CUDA core pipeline. Fine-grain cache is a good solution to improve cache efficiency when application generates many fine-grain memory requests due to memory divergence. In general, fine-grain cache splits the 128B cache line into small chunks (like 32B or 64B) statically and chooses fine or coarse mode before application runs. Since the hardware configuration is fixed for a fine-grain cache, it is not able to adjust the cache granularity when current cache granularity is not suitable for the memory access pattern any more. Given to these reasons, we introduce a multi-grain cache design, called DyCache, which not only supports both fine-grain and coarse-grain, but also supports the dynamic switch between different cache granularities according to the runtime memory

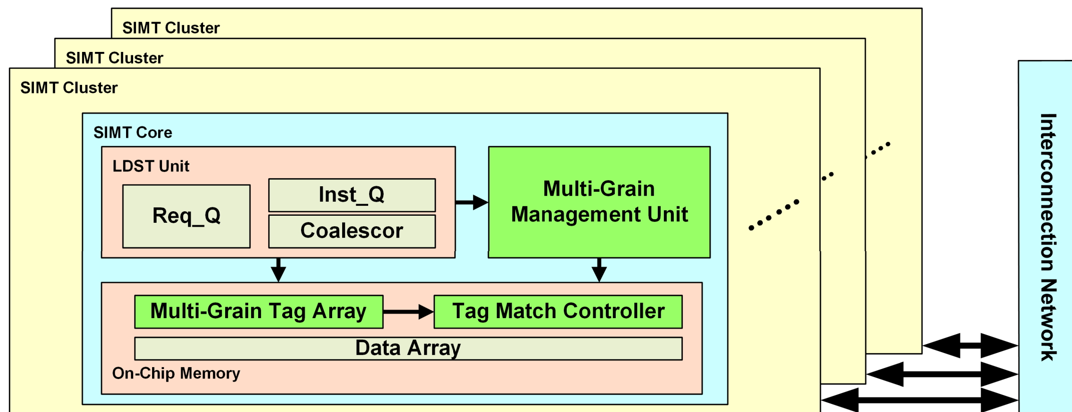

FIGURE 2. **Overview of GPU architecture design for dynamic multi-grain cache management.**

access pattern. In this section, we first depict the architecture overview of DyCache and hardware supports for the dynamic multi-grain cache management, and then describe the basic cache operations and cache granularity selection. Finally, we discuss cache coherence and replacement policy.

A. DYNAMIC MULTI-GRAINED CACHE ARCHITECTURE

1) OVERVIEW

GPU consists of hundreds or thousands of CUDA cores. These cores are grouped into SIMT clusters and for each SIMT cluster, there is a shared common port to connect the interconnection network. Each CUDA core has a LDST unit and a private on-chip L1 data cache. LDST unit is responsible for generating data requests, issuing requests to L1 cache and writing back fetched data to registers. Fig. 2 shows the architecture overview of DyCache. The multi-grain management unit takes the responsibility for profiling memory access pattern and selecting cache granularity for the next period. The selected granularity is sent to the tag match controller. To support multi-grain management, tags of each cache line are split into two parts, which is very similar to Elastic-Cache [5]. One part is called common-tag and the other is chunk-tag. The tag match controller compares chunk-tags of requested data with chunk-tags of chunks in each cache line. In multi-grain mode, all the chunks in the same 128B cache line must have the same common-tag. Moreover, the multi-grain tag array uses both the common-tag and chunk-tags to consider a memory request as a hit or a miss.

2) HARDWARE SUPPORTS FOR DYNAMIC MULTI-GRAINED CACHE MANAGEMENT

Fig. 3 shows the detailed architecture supports for dynamic multi-grain management. When receiving a memory access instruction, LDST unit first tries to coalesce 32 data requests of a warp according to the address of accessed data. And then, LDST unit generates coalesced memory requests and puts them at the end of request queue, waiting to be serviced by L1 data cache.

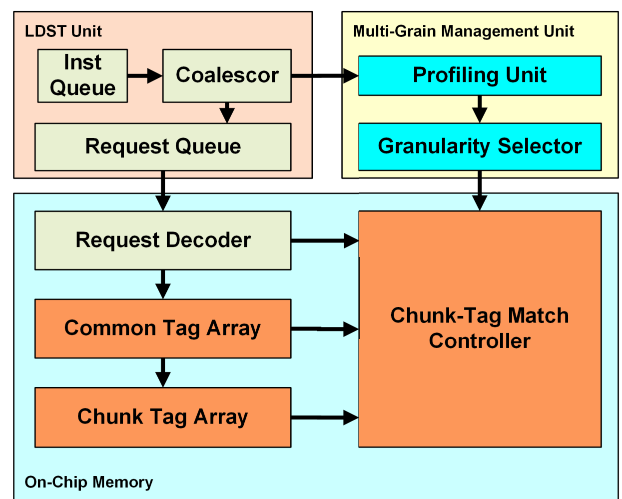

FIGURE 3. **Detailed hardware supports for dynamic multi-grained cache management.**

By coalescing, if the accessed data of a warp are stored in a single 128B cache line, a single memory request can fetch all the requested data from memory. Otherwise, more than one memory requests are generated. For every memory access instruction, the profiling unit in the multi-grain management unit records memory access information (such as data size requested of each memory request, etc.) from LDST unit. After running a certain number of cycles or memory instructions, the granularity selector calculates the assessed value based on the recorded memory access information, compares with the threshold value to decide which cache granularity is the best for the profiled memory access pattern. At last, it sends the control command with the calculated cache granularity to the chunk-tag match controller of L1 cache.

To support the multi-grain management, DyCache splits the tag into two parts, called common-tag and chunk-tag respectively. Assuming the full address space is 4GB and DyCache uses 16KB cache with 32 128B sets, the default address consists of a 20-bit tag, a 5-bit set-index and a 7-bit byte-index. To support the multi-grain cache granularity management, Because the minimum accessible data is 32B,

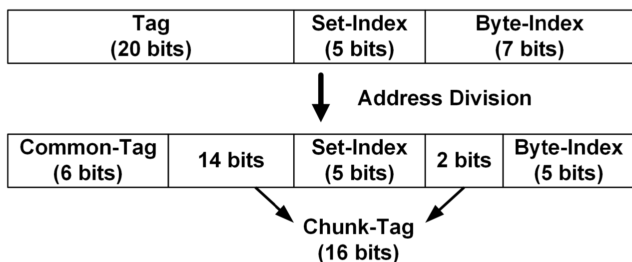

FIGURE 4. An example of address division. Assuming the cache is 16KB and it has 32 4-way sets.

so the byte-index should be 5 bits and the rest of 2 bits can be used as the chunk-tag. Moreover, for the size of chunk-tag, Li *et al.* [5] compares different sizes of the chunk-tag for fine-grain cache in and concludes that the 16-bit chunk-tag is the best choice for fine-grain cache. In conclusion, the 32-bit address is split into a 6-bit common-tag, which is the upper 6-bit of a 20-bit tag, a 16-bit chunk-tag, which is comprised of the lower 14-bit of a 20-bit tag and a upper 2-bit of byte-index, a 5-bit set-index and a 5-bit byte-index. Fig 4 shows the detailed address division for the example. DyCache uses both the common-tag and the chunk-tags to consider a memory request as a hit or a miss. For instance, when current L1 data cache granularity is 32B and there is a 128B data request, DyCache can consider this access as a hit only when the common-tag is matched with one of the common-tags in the indexed set and all the four chunk-tags match with those of the cache line with the same common-tag.

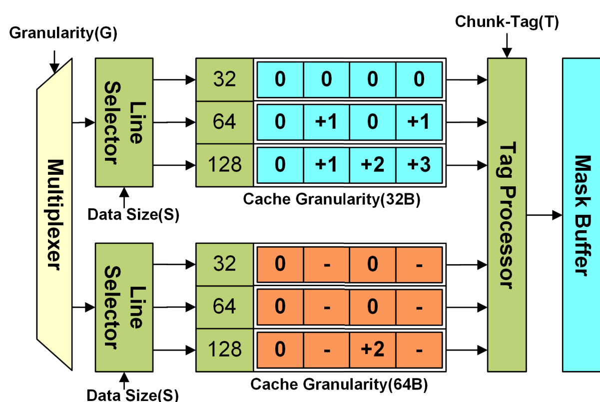

FIGURE 5. Hardware components of the chunk-tag match controller. Two tables store tag masks for 32B cache granularity and 64B cache granularity. "0", "+N" and "-" denote assignment, addition and void operations on chunk-tags respectively.

Because it is a bit complicated to compare chunk-tags for multi-grain cache, DyCache utilizes a chunk-tag match controller to control the chunk-tag comparison. Fig. 5 shows the structure of the controller and how it works. The main components are two tables of tag masks for each fine-grain cache granularity (32B and 64B). Each table has three tag masks for each accessed data size(32B, 64B and 128B). When cache granularity is 128B, only the common-tag and the chunk-tag of the first chunk need to be compared, therefore DyCache don't have the tag mask table for 128B. For each memory access, its multi-grain management information, including the current cache granularity, the accessed data size and the chunk-tag of the data, is sent to the chunk-tag match controller. Based on the current granularity and the accessed data size, the controller looks up the table and fetches the tag mask. The operations on the tag include three types, which are assignment, offsetting and void denoted by "0", "+N" and "-" respectively in Fig.4. Then, tag processor calculates the final chunk-tags for chunk-tag matching, according to the input chunk-tag and tag mask. Finally, the result chunk-tags are stored in the mask buffer.

B. TAG MATCH

For DyCache, each cache line (128B) can store words (32B or 64B) in non-continuous memory space. Therefore, a memory access can be considered as a hit, only when the common-tag of the accessed data is matched with one of common-tags in the indexed set and chunk-tags are the same with those in the cache line with the same common-tag. Fig. 6 illustrates how DyCache processes a memory access.

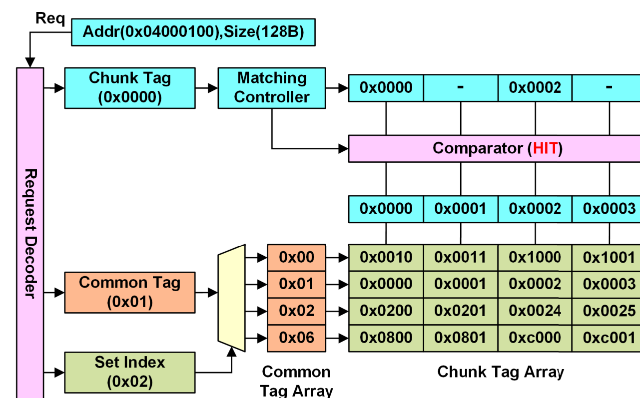

FIGURE 6. A case explains how DyCache processes tag match for a memory access.

Assuming cache size is 32KB and current cache granularity is 64B, a 128B data request with address (0×04000100) is fetched from the top of the request queue of LDST unit. First, the address of the 128B data request is decoded into chunk-tag (0×0000), common-tag (0×01) and set-index (0×02) by the request decoder. Then, common-tag (0×01) is compared with all the four common-tags in the set indexed by set-index (0×02). If there is no match, DyCache pushes this request to the miss queue and fetches another memory request from the LDST unit. If the common-tag matches with one of those common-tags in the set, then the chunk-tags of the corresponding cache line are read out and stored in the buffer. At the meantime, the chunk-tag (0×0000) of the request is sent to the chunk-tag match controller. According to the structure of the chunk-tag match controller showing in Fig. 4, the controller first selects the tag mask

(0, -, +2, -) based on the cache granularity and the accessed data size. Then, the tag processor of the controller calculates the result chunk-tag array (0 × 0000, -, 0 × 0002, -) with the tag mask (0, -, +2, -) and the chunk-tag (0 × 0000) and stores the chunk-tag array (0 × 0000, -, 0 × 0002, -) in the buffer. At last, the comparator compares the chunk-tags (0 × 0000, 0 × 0001, 0 × 0002, 0 × 0003) of the cache line with the common-tag (0 × 01) and the calculated chunk-tag array (0 × 0000, -, 0 × 0002, -). If chunk-tags are matched and all the words are valid, this request is considered as a hit. Otherwise, DyCache considers this request as a miss and puts the request into the miss queue.

In the design of dynamic multi-grain cache management, DyCache supports downward compatibility. That means when cache granularity changes from big size to small one, DyCache won't invalidate data in the cache, because it maintains the chunk-tags and the chunk-statuses of the smallest granularity (32B). Every time filling fetched data in cache, the chunk-tag for each 32B word is updated simultaneously in the chunk-tag array. But, when the cache granularity changes from small size to big one, DyCache has to flush the cache due to the cache coherency.

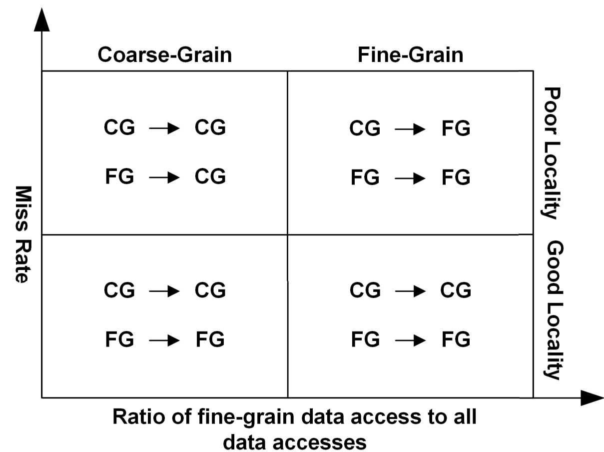

FIGURE 7. DyCache utilizes cache miss rate and the ratio of fine-grain data access to all data accesses to evaluate cache efficiency. According to the two dimensions, memory access patterns are categorized into four types. DyCache makes the decision of the change of cache management granularity based on the type of memory accesses. "CG" and "FG" denote coarse-grain cache management and fine-grain cache management respectively and the arrow means the change of cache management granularity.

C. CACHE GRANULARITY SELECTION

The cache granularity selection of DyCache consists of sampling characteristics of memory accesses, profiling the access pattern and selecting an appropriate cache granularity by comparing with the threshold. The multi-grain management unit in LDST unit samples every memory access instruction until the number of sampled memory requests reaches the upper limits (such as 1000 memory requests). At the end of sampling, the multi-grain management unit processes the sampled raw data of memory accesses, including the size of requested data per memory request, etc. After setting up the new cache granularity, the multi-grain management unit begins a new cycle of sampling.

To select an appropriate cache granularity, we should answer two questions that whether DyCache is efficient at current cache granularity for the memory access pattern of the application and what cache granularity DyCache should choose in the next period, if DyCache is inefficient. DyCache solves the questions in two steps. First, DyCache evaluates its efficiency from two dimensions, the cache miss rate and the ratio of fine-grain data accesses (32B and 64B) to all the data accesses. The ratio of fine-grain data accesses to all the data accesses is calculated by the distribution of the accessed data sizes of sampled memory accesses. The cache miss rate reflects the spatial locality of data accesses and the ratio of fine-grain data accesses to all the data accesses reflects the proportion of fine-grain data access.

Therefore, memory access patterns are categorized into four types, which is shown in Fig. 7. When memory accesses have good locality, no matter the data access is coarse-grain or fine-grain, DyCache will keep cache management granularity unchanged. For example, assuming current cache granularity is coarse-grain (128B) and the majority of data access are fine-grain (say 32B), if these data access have good locality, this means fine-grain data accesses benefit from data prefetching of coarse-grain cache management. Hence, DyCache continues using coarse-grain cache management. When memory accesses have the poor locality, two cases will cause the change of cache management granularity. One case is when data access is coarse-grain and current cache management granularity is fine-grain, poor locality will cause frequent cache line replacement. Although fine-grain cache management supports coarse-grain memory access, the overheads of cache coherence increases due to the frequent cache line replacement. Therefore, Dycache switches cache management granularity from fine-grain to coarse-grain. Another case is when data access is fine-grain and cache management granularity is coarse-grain. Due to memory divergence, coarse-grain cache management fetches a great deal of unwanted data, which decreases cache efficiency. Therefore, DyCache switches cache management granularity from coarse-grain to fine-grain.

If DyCache decides to utilize the fine-grain cache management, it will compare the number of 32B data requests and 64B data requests. If the number of 32B data requests is more than that of 64B data requests, DyCache chooses 32B as the new cache granularity. Otherwise, it chooses 64B.

D. SUPPORTED CACHE GRANULARITY

Currently, the cache granularities supported by DyCache are 32B, 64B and 128B. This is because modern GPU only can issue 32B, 64B or 128B memory requests. Therefore, smaller granularity (such as 16B, 8B, etc.) cannot benefit cache efficiency and throughput. However, as new GPU applications emerge, smaller granularity may become important in the future as well.

E. CACHE LINE REPLACEMENT POLICY AND CACHE COHERENCE

For the cache line replacement in coarse-grain mode, the whole cache line should be replaced when a request is a miss. Hence, DyCache keeps using the default cache line replacement policy (e.g. FIFO, LRU). While in multi-grain mode, DyCache uses different policies to process a 128B request miss and a fine-grain data (32B or 64B) request miss respectively. For an 128B request miss in multi-grain mode, DyCache can also use FIFO or LRU replacement policy because DyCache maintains LRU information for the 128B cache line. For a 32B or 64B data request miss, two cases should be considered. One case is that when the common-tag of a certain cache line is matched with the common-tag of requested data in the indexed set. In this case, DyCache needs to choose one or two chunks to evict. By comparing the last access time of each chunk, DyCache will evict the least recent used chunk. The other case is that there is no common-tag of all cache lines in the indexed set matching with the common-tag of the requested data. For this case, DyCache just chooses a 128B cache line based on LRU policy to evict, writes back all the dirty chunks and fills the fetched data in the first 32B or 64B chunk.

For the multi-grain cache management, DyCache also needs to consider the cache coherence problem. The cache coherence problem happens when the size of accessed data is bigger than the cache granularity. For example, when the current cache granularity is 32B, if only the chunk-tag of the first 32B word of a 64B data matches with a 32B chunk-tag of a certain cache line in the indexed set with the same common-tag, this causes a partial hit. In this partial hit case, DyCache invalidates all the partial hit 32B chunks and writes back the dirty ones. Then, DyCache reissues this 64B request into memory pipeline.

IV. METHODOLOGY

We choose 19 representative applications from Rodinia [11] and GraphBig [12]. For conventional general purpose computing on GPU, Rodinia provides a collection of programs. For emerging applications, GraphBig includes a broad scope of graph computing applications. Table 1 shows the the abbreviation names and descriptions of all the applications.

We implement our idea on GPGPUSim-3.2.2 [13]. And we use the default GTX-480 configuration for the simulator. To compare with Elastic-Cache, we implement it on GPGPUSim simulator as well. Table 2 shows the basic configuration of the simulator.

TABLE 1. Benchmarks for evaluation.

Abbreviation	Description
BP	Back propagation
BT	B+tree search
BFS	Breadth first search
CFD	Computational Fluid Dynamics
GE	Gaussian elimination
HW	Heart wall
HP	HotSpot
KC	KMeans clustering
LUD	LU Decomposition
NW	Needleman-Wunsch
PF	Particle filter
SRAD	Speckle Reducing Anisotropic Diffusion
SC	Streamcluster
BC	Betweenness centrality
CC	Connected component
DC	Degree centrality
GC	Graph coloring
SSSP	SSSP
TC	Triangle count

TABLE 2. Basic GPGPU-Sim configuration.

Number of SMs	15
Threads per SM	1536
Threads per warp	32
Warp scheduling policy	GTO
L1 cache size	16KB
L1 cache association	4-way
L1 cache block size	128B

V. EVALUATION

A. PERFORMANCE

The biggest advantage of DyCache is that DyCache is very flexible to choose an appropriate cache granularity for applications in real time according to the memory access pattern, while Elastic-Cache uses a static management policy which is not able to change cache granularity for different memory access patterns and only supports one small cache granularity (32B or 64B). Fig. 8 shows the normalized IPCs of Elastic-Cache and DyCache with different configurations for the above 19 applications and the baseline is the basic wide cache line cache. Elastic-Cache-32 and Elastic-Cache-64 represents Elastic-Cache using 32B-chunk and 64B-chunk as the cahce management granularity respectively. DyCache-N means DyCache sets N% as the threshold of cache miss rate and the ratio of fine-grain accessed data to distinguish good/poor locality and fine-/coarse-grain memory access. In general, the geometrical mean normalized IPC of the 19 applications for Elastic-Cache-32, Elastic-Cache-64, DyCache-25, DyCache-50 and DyCache-75 are 1.12, 1.09, 1.12, 1.18, and 1.18 respectively. Moreover, the geometrical mean performance for regular applications (BP, BC, BT, GE, GC, HW, HP, LUD, NW and SRAD) are 0.92, 0.94, 0.92, 0.99 and 1.01 respectively, while for irregular applications (BFS, CFD, CC, DC, KC, PF, SC, SSSP and TC) are 1.36, 1.25, 1.37, 1.40, 1.39. From these results, we can see that DyCache improves the IPC by 40% for applications with irregular memory accesses and doesn't harm the performance for applications with regular memory accesses at the meantime. Comparing with static cache management, DyCache performs 9% and 4% better than Elastic-Cache for regular and irregular memory accesses respectively.

Although DyCache can perform better than Elastic-Cache for most of applications, however for a few applications, Elastic-Cache outperforms DyCache. For example, BFS

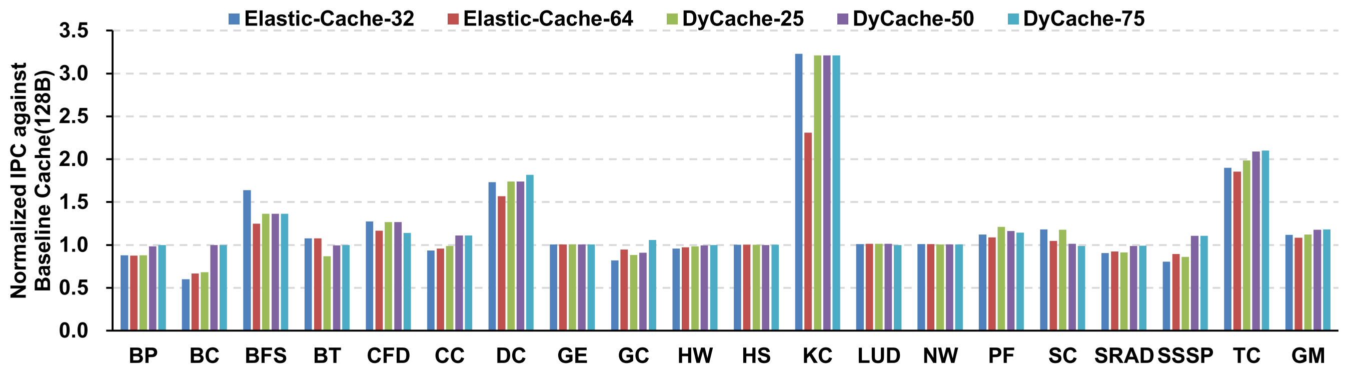

FIGURE 8. Normalized IPCs of Elastic-Cache and DyCache for all the 19 applications. Baseline is basic cache with wide cache line(128B).

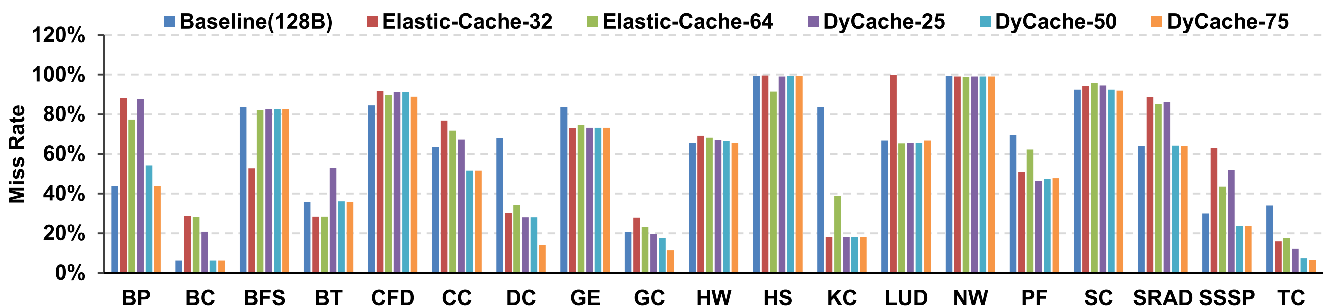

FIGURE 9. Miss rates of the baseline cache, Elastic-Cache and DyCache for all the 19 applications.

performs 20% better on Elastic-Cache-32 than on DyCache. The reason is that BFS has two kernels which have different memory access patterns. One kernel performs irregular memory accesses, while the memory access pattern of another kernel is regular. Therefore, in this case, DyCache has to switch cache granularity between fine- and coarse-grain frequently, which degrades its performance dramatically.

For the dynamic cache management of DyCache, it evaluates cache efficiency from two dimensions, miss rate and the ratio of fine-grain memory accesses to all memory accesses, which we have discussed in the section of cache granularity selection. The threshold for miss rate and the ratio of fine-grain is a key factor that directly affects the performance. If the threshold is too low, DyCache may treat a coarse-grain memory access pattern as a fine-grain memory access pattern by mistake, which brings unnecessary overheads of the fine-grain management. On the other hand, if the threshold is too high, DyCache will be insensitive to the change of the memory access pattern, which makes DyCache inefficient. According to our results, thresholds between 50% and 75% will be a good choice for most of applications.

Besides, another problem for DyCache is the overhead of changing cache management granularity. Due to the downward compatibility, there are nearly no overheads when cache granularity changes from big size to small size, because the management information of chunks is able to be inherited. But, when cache granularity changes from small size to big one, all the information of the chunks will be invalided due to the consideration of cache coherence. The performance degradations of BFS, CFD and SC are caused by this problem. One solution is retaining the data whose size is equal to or bigger than the new cache management granularity. For example, if cache granularity changes from 32B to 64B, DyCache retains all the 64B and 128B data and invalidates all the 32B data.

Moreover, by using the dynamic multi-grain management, DyCache decreases the cache miss rate and reduces the number of memory requests sent to the off-chip memory due to the cache miss. Fig. 9 shows the cache miss rates of Baseline (128B), Elastic-Cache, and DyCache. Obviously, DyCache has the lowest cache miss rate for most of applications and the miss rate goes down by 10% to 60%.

B. SENSITIVITY STUDY

1) CACHE VOLUME

For GTX 480, L1 cache and shared memory share the whole 64KB on-chip memory. There are two cache volume configurations (16KB and 48KB) for L1 cache. Fig. 10 shows the normalized IPCs of 16KB/48KB DyCache using two different thresholds (50% and 75%) respectively. After increasing the cache volume, the speedup of DyCache against the baseline cache (128B) drops. This is because the data set we test is a bit small relative to the 48KB cache volume, which induces that the miss rate of baseline cache drops and the IPC of the

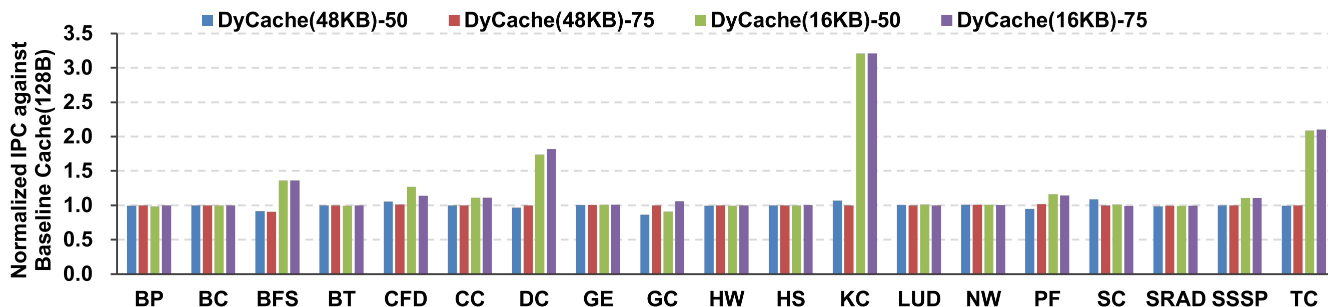

FIGURE 10. Normalized IPCs of 16KB/48KB DyCache using two different thresholds (50% and 75%).

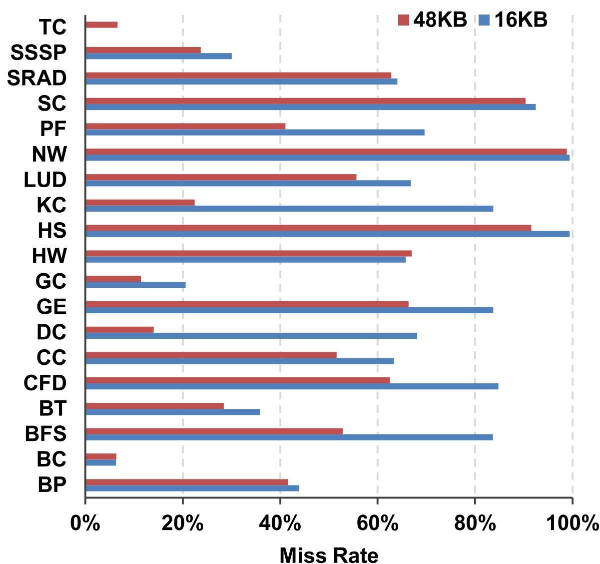

FIGURE 11. Miss rates of the baseline cache drops when its volume increases from 16KB to 48KB.

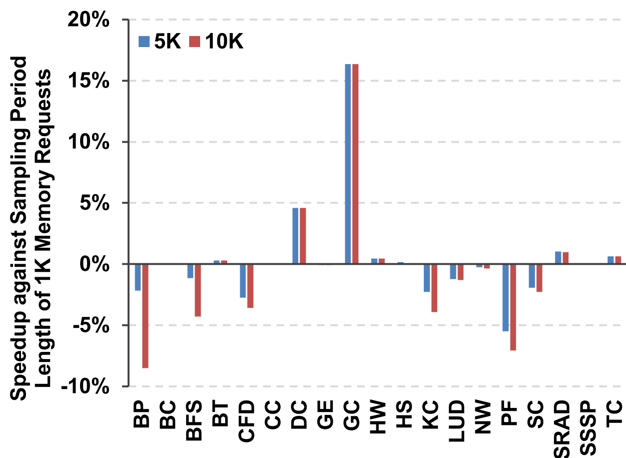

FIGURE 12. Speedups of using long sampling period length. The baseline is DyCache with sampling period length of 1K memory access requests.

baseline increases by 24% on average. Fig. 11 shows the drop of miss rate of the baseline cache (128B) after cache volume increases to 48KB. However, as the practical data sets are increasingly getting bigger, the 48KB on-chip memory is not sufficient at all. Therefore, it's still very important to explore the high efficient cache management for GPU.

2) SAMPLING PERIOD LENGTH

The sampling period length is another factor that affects the performance of DyCache. Fig. 12 shows the speedup of two sampling period lengths (5K and 10K memory access requests) against 1K memory access requests. On one hand, longer sampling period length may harm performance, because of the long adjustment interval. In other words, when memory access pattern changes at the beginning of the sampling period, DyCache can change the cache management granularity until the end of this sampling period. However, on the other hand, longer sampling period length can benefit performance for applications with frequent fluctuation of memory access patterns, because longer sampling period length can diminish the frequency of changing cache granularity back and forth. Therefore, an adjustable sampling period length will be a good choice. At the beginning of the application execution, DyCache can use a short sampling period length to character the memory access pattern quickly. Moreover, if the memory access pattern varies frequently, DyCache can increase the sampling period length properly.

VI. RELATED WORKS

A. MEMORY ACCESS COALESCING

Memory access coalescing technique has been studied for both CPU and GPU. To improving memory bandwidth for CPU, Juan [14] studies the multi-porting and banking for superscalar processors, while Davidson and Jinturkar [15] uses wide memory requests by combining several requests. For GPU, memory coalescer in LDST unit is responsible for combining intra-warp requests that access continuous 128B-data in memory space. Kloosterman *et al.* [16] proposes the inter-warp coalescing by merging requests across warps. Yang *et al.* [17] and Baskaran *et al.* [18] use compile techniques to improve the efficiency of coalescing. DyCache utilizes the multi-grain cache management to improve L1 cache efficiency, which is orthogonal to memory

access coalescing techniques. By using both DyCache and memory access techniques, we can build a fine-grain memory system for GPU to provide a high efficient on-chip data storage and a high bandwidth off-chip data transfer.

B. FINE-GRAIN CACHE MANAGEMENT

Prior works propose fine-grain cache managements to improve L1 cache efficiency. Some prior works explore splitting the coarse-grain cache line into small multiple sectors, such as Sector Cache [7], Decoupled Sectored Cache [8] and Pool-of-Subsector-Cache [9]. However, these works can only store words in continuous memory space. DyCache supports the multi-grain management and allows words in non-continuous memory space to be stored in the same 128B cache line. Elastic-Cache [5] explores a fine-grain management to store words in non-continuous memory space and stores chunk-tags in unused shared memory space without shrinking L1 cache size. Different from the static fine-grain management of Elastic-Cache, DyCache can adjust the cache granularity based on memory access patterns. LAMAR [19] utilizes a locality-aware hardware predictor to support the adaptive adjustment of access granularity, while DyCache chooses cache management granularity by profiling memory access pattern and cache miss rate.

VII. CONCLUSION

We propose a dynamic multi-grain cache management for GPU on-chip L1 cache to improve its efficiency for regular and irregular memory accesses. Unlike other static cache managements, DyCache is very flexible to switch among different granularities. DyCache can split 128B-cache line into three different sizes of chunks (128B, 64B and 32B) dynamically, based on memory access patterns. DyCache utilizes the cache miss rate and the ratio of fine-grain memory accesses to all the data accesses to character the memory access pattern and the cache efficiency. If memory accesses are irregular and fine-grain and cache is inefficiently utilized, DyCache chooses a fine-grain cache granularity according to the distribution of the sizes of the accessed data to improve the cache efficiency. Otherwise, DyCache utilizes coarse-grain cache line (128B) to provide high efficient accesses for the regular memory access pattern. Our experiment demonstrates that DyCache can achieve a 40% geometric mean speedup of IPC for applications with irregular memory accesses and doesn't impact the performance of applications with regular memory accesses.

REFERENCES

[1] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. IEEE Int. Symp. Workload Characterization*, Nov. 2012, pp. 141–151.

[2] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A MapReduce framework on graphics processors," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 260–269.

[3] M. K. Qureshi, M. A. Suleman, and Y. N. Patt, "Line distillation: Increasing cache capacity by filtering unused words in cache lines," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, Feb. 2007, pp. 250–259.

[4] R. C. Murphy and P. M. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *IEEE Trans. Comput.*, vol. 56, no. 7, pp. 937–945, Jul. 2007.

[5] B. Li, J. Sun, M. Annavaram, and N. S. Kim, "Elastic-cache: GPU cache architecture for efficient fine- and coarse-grained cache-line management," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May/Jun. 2017, pp. 82–91.

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

[7] J. S. Liptay, *Structural Aspects of the System/360 Model 85, Part II: The cache*. San Mateo, CA, USA: Morgan Kaufmann, 1968.

[8] A. Seznec, "Decoupled sectored caches: Conciliating low tag implementation cost," *ACM SIGARCH Comput. Archit. News*, vol. 22, no. 2, pp. 384–393, 1994.

[9] J. B. Rothman and A. J. Smith, "The pool of subsectors cache design," in *Proc. Int. Conf. Supercomput.*, 1999, pp. 31–42.

[10] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji, "Adapting cache line size to application behavior," in *Proc. 13th Int. Conf. Supercomput.*, 1999, pp. 145–154.

[11] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2009, pp. 44–54.

[12] 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.

[13] 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.*, Apr. 2009, pp. 163–174.

[14] T. Juan, "Data caches for superscalar processors," in *Proc. Int. Conf. Supercomput.*, 1997, pp. 60–67.

[15] J. W. Davidson and S. Jinturkar, "Memory access coalescing: A technique for eliminating redundant memory accesses," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 186–195, 1994.

[16] J. Kloosterman *et al.*, "Warppool: Sharing requests with inter-warp coalescing for throughput processors," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 433–444.

[17] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proc. ACM SIGPLAN Conf. Programming Language Design Implement.*, 2010, pp. 86–97.

[18] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," in *Proc. Int. Conf. Compiler Construction*, 2010, pp. 244–263.

[19] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2013, pp. 86–98.

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 interest includes 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 authored over 20 papers in internationally recognized journals and conferences. His main research interests include processor architecture and computer graphics.

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. His research interests include on-chip networks, SIMD architectures, and arithmetic unit designs. He has authored over 30 papers in internationally recognized journals and conferences.

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, 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 ACM in 2003.