An Efficient GPU Cache Architecture for Applications with Irregular Memory Access Patterns

BINGCHAO LI, Civil Aviation University of China & Tianjin University, China

JIZENG WEI and JIZHOU SUN, Tianjin University, China

MURALI ANNAVARAM, University of Southern California, USA

NAM SUNG KIM, University of Illinois at Urbana-Champaign, USA

GPUs provide high-bandwidth/low-latency on-chip shared memory and L1 cache to efficiently service a large number of concurrent memory requests. Specifically, concurrent memory requests accessing contiguous memory space are coalesced into warp-wide accesses. To support such large accesses to L1 cache with low latency, the size of L1 cache line is no smaller than that of warp-wide accesses. However, such L1 cache architecture cannot always be efficiently utilized when applications generate many memory requests with irregular access patterns especially due to branch and memory divergences that make requests uncoalesced

Extension of conference paper: this journal article is extended from a 10-page conference paper “Elastic-Cache: GPU Cache Architecture for Efficient Fine- and Coarse-Grained Cache-Line Management,” accepted by the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017, May 29–June 2). We do the further study and extend the conference paper in several directions:

- We propose an enhanced version of Elastic-Cache named Elastic-Plus that can issue 32-byte requests in parallel to utilize the bandwidth provided by L1 cache more efficiently than Elastic-Cache and reduce the processing latency of memory instructions. (Section 4.4.1).
- We design the architecture for Elastic-Plus, including the control path and data path of how fine-grained requests are issued in parallel. To reduce the chunk-tag access conflicts, we also rearrange the storage of chunk-tags. To reduce the overhead of comparison logic for chunk-tags, we propose to compare common-tags and chunk-tags serially. (Section 4.4.2 and Section 4.4.3).
- We specify how to make decisions about cache hit or miss based on common- and chunk-tags for fine- and coarse-grained requests. (Section 4.1 and Section 4.2)
- Different from the shared architecture of L1 cache and the shared memory in the conference paper, L1 cache and the shared memory are separated in this paper, which is consistent with that of recent GPUs. And we also re-design the architecture of Elastic-Cache for this new feature. (Section 4.3).
- We run new experiments with some new benchmarks to evaluate and analyze the performance of multiple cache architectures and demonstrate that the proposed Elastic-Plus approach brings performance improvement and is more energy-efficient than Elastic-Cache. (Section 6).

This work is partly supported by the Natural Science Foundation of Tianjin (18JCQNJC00400), Scientific Research Foundation of Civil Aviation University of China (2017QD12S), the Fundamental Research Funds for the Central Universities of Civil Aviation University of China (3122018C023, 3122018C021), National Natural Science Foundation of China (61702521), U.S. National Science Foundation (CNS 17-05047).

Authors' addresses: B. Li, School of Computer Science and Technology, Civil Aviation University of China, 2898 Jinbei Road, Dongli District, Tianjin, 300300, China; email: bcli@cauc.edu.cn; J. Wei and J. Sun, College of Intelligence and Computing, Tianjin University, 135 Yaguan Road, Haihe Education Park, Tianjin, 300350, China; emails: {weijizeng, jzsun}@tju.edu.cn; M. Annavaram, Department of Electrical and Computer Engineering, University of Southern California, Hughes Aircraft Electrical Engineering Center, 3740 McClintock Avenue, Los Angeles, California, 90089-2562, USA; email: annavara@usc.edu; N. S. Kim, Department of Electrical and Computer Engineering, Coordinated Science Laboratory, 1308 West Main Street, Urbana, Illinois, 61801-2307, USA; email: nskim@illinois.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2019/06-ART20

<https://doi.org/10.1145/3322127>

and small. Furthermore, unlike L1 cache, the shared memory of GPUs is not often used in many applications, which essentially depends on programmers. In this article, we propose Elastic-Cache, which can efficiently support both fine- and coarse-grained L1 cache line management for applications with both regular and irregular memory access patterns to improve the L1 cache efficiency. Specifically, it can store 32- or 64-byte words in non-contiguous memory space to a single 128-byte cache line. Furthermore, it neither requires an extra memory structure nor reduces the capacity of L1 cache for tag storage, since it stores auxiliary tags for fine-grained L1 cache line managements in the shared memory space that is not fully used in many applications. To improve the bandwidth utilization of L1 cache with Elastic-Cache for fine-grained accesses, we further propose Elastic-Plus to issue 32-byte memory requests in parallel, which can reduce the processing latency of memory instructions and improve the throughput of GPUs. Our experiment result shows that Elastic-Cache improves the geometric-mean performance of applications with irregular memory access patterns by 104% without degrading the performance of applications with regular memory access patterns. Elastic-Plus outperforms Elastic-Cache and improves the performance of applications with irregular memory access patterns by 131%.

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**;

Additional Key Words and Phrases: GPU, cache, shared memory, thread

ACM Reference format:

Bingchao Li, Jizeng Wei, Jizhou Sun, Murali Annavaram, and Nam Sung Kim. 2019. An Efficient GPU Cache Architecture for Applications with Irregular Memory Access Patterns. *ACM Trans. Archit. Code Optim.* 16, 3, Article 20 (June 2019), 24 pages.
<https://doi.org/10.1145/3322127>

1 INTRODUCTION

GPUs rely on massive thread level parallelism to achieve high throughput. The thread level parallelism generates a large number of concurrent memory requests, since every thread is capable of generating an individual request. In GPUs, each core (e.g., streaming multiprocessor (SM) in NVIDIA GPUs) typically supports high-bandwidth/low-latency on-chip shared memory and L1 cache to efficiently service the large number of concurrent memory requests (to contiguous memory space). In particular, to support concurrent accesses to L1 cache, GPU L1 cache lines are very wide compared with CPU L1 cache lines (128 versus 64 bytes). Typically a single cache block can support the memory access of all 32 threads in a warp, provided these accesses are all coalesced into a 128-byte contiguous address space [25]. Such coalescing reduces the number of accesses to the memory systems and makes the memory systems more efficient.

Some application classes, such as graphics, generate memory requests with regular access patterns and hence can greatly benefit from such high-bandwidth L1 cache architecture [6]. As GPU popularity continues to grow, however, many applications with irregular data structures (e.g., graph algorithms) are being ported to run on GPUs. These applications in particular cannot take advantage of such wide cache lines, because irregular data structures often incur branch and memory divergences [3, 4, 22]. Divergence leads to irregular memory access patterns. This in turn increases the number of memory accesses to non-contiguous memory space, prevents GPU caches from efficiently servicing disparate memory requests, and reduces the efficiency of cache line usage, since a cache miss fetches an entire cache line but only a fraction of the cache line is demanded. As such, many words brought to cache lines are never used until the cache lines are evicted, as (1) the size of each memory request to a cache line is mostly smaller than the maximum size that a cache line can service and (2) the spatial locality of uncoalesced memory accesses is often poor. Uncoalesced memory accesses substantially reduce the effective capacity of L1 cache, which in turn incurs frequent replacements of cache lines, wastes the bandwidth of L2 cache and off-chip main memory, and significantly degrades the performance.

Apart from L1 cache, GPUs also provide shared memory to enable communication and data reuse among threads of a thread block running on an SM. However, it is the responsibility of programmers to efficiently use the shared memory. As such, we observe that some applications do not use the shared memory space at all, which agrees to many prior studies (e.g., Reference [36]).

Given these observations, the goal of this work is to improve the efficiency of L1 cache usage while exploiting the unused shared memory space to achieve this efficiency. In this article, we propose Elastic-Cache, which can efficiently support both fine- and coarse-grained L1 cache line management for applications with both regular and irregular memory access patterns. For applications with regular access patterns, Elastic-Cache operates as traditional L1 cache and does not negatively impact performance. However, for applications with irregular memory access patterns, Elastic-Cache exploits *unused shared memory space* to provide more tags for each cache line and allow each (128-byte) cache line to store (four 32- or two 64-byte) words in *non-contiguous* memory space.

Similar to Elastic-Cache, Sector-Cache manages cache lines at the word-level, but it only allows each cache line to store (e.g., four 32-byte) words in *contiguous memory space* [20, 29]. The main goal of Sector-Cache is to eliminate bringing unwanted data or to store a subset of words without having to allocate a cache line on a store miss. Amoeba-Cache was proposed for CPUs to store multiple non-contiguous words in a single cache line. However, it stores tags for fine-grained cache line management within the cache data arrays [16], thereby reducing the effective size of the cache to support fine-grained cache line management. Furthermore, we demonstrate that some unique aspects of GPU L1 cache architecture diminishes a significant fraction of the performance benefit of Amoeba-Cache. That is, a microarchitectural limitation of GPU L1 cache requires serial accesses of tags and a cache line [14], whereas Amoeba-Cache demands parallel accesses of tags and all the cache lines in a set. In contrast, Elastic-Cache does not reduce the capacity of L1 cache, because it stores these tags in the unused shared memory space instead of cache data arrays, which is unique to GPU architecture. As we will demonstrate throughout the rest of this article, Elastic-Cache resolves several subtle microarchitectural challenges that are specific to a GPU design. Thus, Elastic-Cache is a unique design that takes advantage of GPU-specific microarchitecture features.

We design Elastic-Cache to improve the cache efficiency by providing fine-grained cache line management. However, one downside of Elastic-Cache is that the high bandwidth of L1 cache is wasted, since only 25% is utilized with fine-grained accesses (32-byte requests). In addition, we also observe that a memory instruction of a warp can generate multiple fine-grained requests that cannot be coalesced in applications with irregular memory access patterns. That is, it takes GPUs multiple cycles to process these requests serially. Based on this insight, to efficiently utilize the 128-byte bandwidth provided by L1 cache, we further propose an enhanced version of Elastic-Cache, which is named Elastic-Plus. Elastic-Plus is capable of issuing up to four 32-byte requests originated from the same warp instruction in parallel. As a result, the bandwidth utilization of L1 cache is improved as long as two or more requests are issued simultaneously, which, however, also reduces the average processing latency of memory instructions and finally improves the performance.

Compared with conventional L1 caches, Elastic-Cache improves the geometric-mean performance by 104% for applications with irregular memory access patterns. Besides, Elastic-Cache outperforms the same size Amoeba-Cache by at least 72%. Moreover, Elastic-Cache does not hurt the performance of applications with regular memory access patterns. Lastly, with Elastic-Plus, the performance is further improved by 27% over Elastic-Cache for applications with irregular memory access patterns through issuing requests in parallel.

The remainder of this article is organized as follows: Section 2 describes the background. Section 3 demonstrates that applications with irregular memory access patterns inefficiently use on-chip shared memory and L1 cache. Section 4 describes Elastic-Cache and Elastic-Plus for applications with both regular and irregular memory access patterns in detail. Sections 5 and 6 provide the experimental methodology and evaluation results, respectively. Section 7 provides the discussion about how Elastic-Cache and Elastic-Plus work with other architectural features. Section 8 discusses related work. Section 9 concludes this article.

2 BACKGROUND

2.1 On-Chip Memory Architecture

GPUs provide on-chip memories for every SM to reduce the memory access latency and content to off-chip memory. The on-chip memories include software-managed shared memory and hardware-managed L1 cache. The shared memory that is explicitly managed by a programmer enables inter-thread communication within a thread block. That is, different thread blocks cannot share data or communicate through the shared memory. Accessing the shared memory is at least $100\times$ faster than accessing global memory [9]. The shared memory consists of 32 banks with 1,024-bit (128-byte) I/O in total. When accessing the shared memory, GPUs can issue up to 32 32-bit memory requests simultaneously to different row addresses as long as the requests access different banks of the shared memory. In contrast, L1 cache is managed by hardware and is transparent to programmers. Although programmers can use inline assembly or compilation flags to determine whether global memory accesses are cached in L1 cache or not with the most recent GPUs [25], they cannot determine which cache line is accessed. In addition, if thread blocks of a kernel consume too much resources of an SM, then L1 caching will be disabled automatically. L1 cache can also be used as texture cache by a texture unit that implements various addressing modes and data filtering. Moreover, GPUs can issue only one request per access to get data from a 128-byte cache line when accessing L1 cache [25, 32].

In brief, GPUs can service only 128 bytes for either L1 cache or shared memory for a given cycle. For a set associative cache, therefore, GPUs cannot support parallel accesses of tags and all cache lines in a set as a conventional CPU L1 cache does. That is, GPUs must look up the tag array first and then determine which cache line to access based on the tag comparison [14]. These serial accesses of tags and a cache line can be acceptable for GPUs, because GPUs can often tolerate latency in some degree and emphasize power efficiency. The serial accesses are more power-efficient than the parallel accesses (e.g., References [2, 31]).

2.2 Amoeba-Cache

In this section, we briefly present the details of the most relevant prior work, Amoeba-Cache [16]. Amoeba-Cache was originally proposed for CPUs and supports variable cache line sizes. Amoeba-Cache stores extra tags next to its corresponding data in the cache data array, as shown in Figure 1(a), and uses a tag-bitmap shown in Figure 1(b) to indicate which words in the data array represent tags. Tags are used to index fixed-size regions, and the tag field of each region not only contains tag bits of this region but also start and end bits to specify small cache lines within this region, as depicted in Figure 1(b). When a request accesses the cache, its `set` index is used as the address of accessing tag-bitmap (Figure 1(b)①) and data array (Figure 1(b)②). Because adjacent words cannot be both tags, $N/2$ multiplexers controlled by the tag-bitmap signal (Figure 1(b)③) are required to route one of the adjacent words to the comparators; N is the number of words in one set. Then the hit signal generated by the comparators is used to control the word selector (Figure 1(b)④).

Fig. 1. Amoeba-Cache.

One significant difference of our proposed Elastic-Cache over Amoeba-Cache is that Amoeba-Cache stores extra tags within the cache data arrays. As such, the effective cache capacity is reduced especially when more smaller cache lines are used. The percentage of tag storage in one set is m/N , where m is the number of tags in one set. *Another drawback of Amoeba-Cache is that it is designed for cache in which all cache lines of a set can be accessed simultaneously, whereas GPU cache implementations do not support such simultaneous accesses of all cache lines in a set, as described in Section 2.1.* Moreover, in many contemporary cache implementations for CPUs (e.g., References [2, 31]), not to unnecessarily increase the number of large and power-hungry sense amplifiers in data arrays, the bitlines of only one cache line in a set can be connected to the sense amplifiers through column multiplexers in data arrays. Such implementations do not increase the cache access time, since tag comparisons can be done before the required timing to assert the column multiplexer selection signals in many designs. Our experiment using CACTI 5.3 [23] shows that reading all cache lines in a set in parallel consumes $3.4\times$ more energy than reading only one cache line in a set; we input 32nm technology, 1 read/write port, and ITRS-LSTP devices for tag and data arrays to CACTI. Therefore, Amoeba-Cache in GPUs must read out cache lines in a set one-by-one until it finds a hit. Furthermore, Amoeba-Cache can also result in empty bytes, as shown in Figure 1(a), when the sum of tags and data is not the size of a cache line, which degrades the cache utilization.

3 INEFFICIENT USE OF GPU ON-CHIP MEMORY

3.1 Inefficient Use of L1 Cache

A warp can generate a single 128-byte memory request when all 32 threads of this warp access 32 consecutive 32-bit words that belong to the same 128-byte cache line. GPUs are provisioned with coalescing hardware within the memory pipeline to enable such a coalescing. However, if threads within the same warp access words located in different cache lines, then multiple memory requests are generated and the requested word size is usually smaller than the full cache line size. Figure 2 shows that each memory instruction generates more than four requests (see Section 5 for detailed evaluation methodology) in 12 out of 24 benchmarks. For example, graph applications with irregular memory access patterns such as Survey Propagation (SP) generate more than 12 requests per memory instruction, exhibiting very poor coalescing capability. In addition, the stacked bars in Figure 2 show the percentage of 32-, 64-, and 128-byte memory requests. The request size is determined by the coalescing unit using the following protocol: if only one of two 64-byte (or four 32-byte) chunks of a 128-byte cache line is referenced, then it is classified as a 64-byte (or 32-byte) memory request [25]. As observed, most requests in applications on the left side of Figure 2 are very small. On average, the percentage of 32-byte memory requests is 64% of total memory requests, whereas that of 64- and 128-byte memory requests are only 3% and 33%, respectively.

Fig. 2. Average number of requests per memory instruction and percentage of 32-, 64-, and 128-byte memory requests (stacked bars) when L1 cache is configured as 48KB.

Fig. 3. Cache efficiency for the baseline GPU and the percentage of the unused shared memory when the on-chip memory is configured to provide 48KB L1 cache and 48KB shared memory.

Poor coalescing by itself is not a concern if neighboring warps are able to access other data brought into the cache line. To measure how much of a cache line that is demand fetched is eventually used before it is evicted, we measure cache efficiency using $N_{32B\text{-referenced}} / (N_{128B\text{-evicted}} \times 4)$ where $N_{32B\text{-referenced}}$, $N_{128B\text{-evicted}}$, and the denominator denote the number of referenced 32-byte words (or chunks), the number of evicted 128-byte cache lines, and the number of evicted 32-byte chunks. Figure 3 shows the arithmetic mean of cache efficiency of all 24 benchmarks is 64%. Among 24 benchmarks, 7 benchmarks exhibit cache efficiency lower than 50% for 48KB L1 cache.

The primary reason for poor cache efficiency is that the residency time of a cache line is quite small in GPUs. And the per-thread cache capacity in GPUs is also very small, in the order of a few bytes, since an SM concurrently runs thousands of threads. Thus, when a cache line is demand fetched within a short time window, all the words within the cache line must be consumed before the cache line is evicted. Prior studies have also observed that many words in a cache line are often evicted even before they are referenced [5].

It appears that cache efficiency for irregular applications may be improved by reducing the cache line size to 64 or 32 bytes. But naively shrinking the cache line size will hurt regular applications that traditionally exhibit strong coalescing behavior. Such applications must split 128-byte memory requests into smaller requests to match the cache line size, which can substantially degrade performance of such applications. Hence, what is needed is a cache design that can support wide width accesses when the application exhibits good coalescing behavior, and narrow width cache lines when the applications are unable to use the wide cache lines. By reducing the cache line width, more distinct cache lines may be accommodated in a given cache size.

3.2 Inefficient Use of Shared Memory

The shared memory is a software-managed on-chip memory and thus its use is determined by programmers. Meanwhile, we note that the shared memory space is not fully used in many applications. The lines in Figure 3 show how much shared memory space is occupied when its capacity is 48KB. As shown, 14 benchmarks do not use the shared memory at all and four other benchmarks

(PVC, SM, WC, and II) show little utilization of the shared memory. The compiler can determine whether or not a shared memory is going to be used in advance. On average, the percentage of the unused space of the shared memory is 96%.

Multiple factors can affect the usage of the shared memory. When applications start to use the shared memory, an instruction must be issued first to transfer the data from L2 cache or off-chip main memory to the shared memory. After the data is stored in the shared memory, another instruction needs to load the data to execution units. Thus, using the shared memory often consumes more time and instructions if the data reuse is infrequent [35]. However, the shared memory is only shared by threads within the same thread block. Threads within a thread block use their thread-index to select shared data arrays [19], whereas different thread blocks cannot communicate through the shared memory. Therefore, it is not efficient to use the shared memory for data reuse if data locality exists among thread blocks. Furthermore, GPUs statically determine the number of thread blocks that can be assigned to an SM based on the amount of the shared memory assigned to each thread block, or the number of registers that are needed by each thread, or a hardware resources limit, depending on which constraint is reached first [36]. It has been shown that register file usage or hardware resources limit the number of thread blocks assigned to an SM in many applications well before reaching the shared memory usage limit [11]. The register file usage and hardware resources are implicit while the shared memory is explicit for programmers. Allocating too much shared memory to a thread block will limit the number of thread blocks dispatched to an SM, significantly impacting the parallelism and the performance of GPUs [36]. Lastly, some applications are developed to fit the small shared memory capacities of early GPUs [7].

4 ELASTIC-CACHE

Based on our observations in Section 3, we propose Elastic-Cache, which supports both fine- and coarse-grained cache line to efficiently manage L1 cache and eventually improve the performance. In Elastic-Cache, the number of sets, associativity, and capacity are still the same as those of the baseline L1 cache. To support fine-grained cache line management, we divide a 128-byte cache line into n logical chunks. For example, there are four 32-byte logical chunks for $n = 4$ in Figure 5. Each chunk is associated with a tag denoted by chunk-tag in this article.

The novel aspects of Elastic-Cache are as follows: (1) It stores chunk-tags in the unused shared memory instead of storing such tags in a dedicated array [33] or data arrays [16]. Thus, the cache size is not compromised due to the reduced cache line width, which is a critical consideration for GPUs, since the per-thread cache size is already quite small. (2) It can store chunks from non-contiguous memory space in a cache line, in contrast to Sector-Cache [20, 29], which can only store chunks from contiguous memory space in a cache line. While Elastic-Cache in principle can support 64-byte chunks, we focus on illustrating and evaluating chunk size of 32 bytes in this work, as 64-byte requests in our benchmarks are much fewer than 32-byte requests (cf. Figure 2).

4.1 Chunk- and Common-Tags

The shared memory is divided into 32 banks in NVIDIA GPU architecture [25]. It is organized such that successive 4-byte words are assigned to successive banks, and the bandwidth is 32 bits per bank per cycle. To utilize the 4-byte words on each shared memory bank efficiently for chunk-tag storage, we propose to use 16 bits for the width of chunk-tags in this article. That is, each entry in a bank of the shared memory can store and provide two chunk-tags in Elastic-Cache.

In modern GPUs, the total capacity of the main memory is usually more than 1GB. Thus, 16 bits are not sufficient to store all the necessary tag bits for each 32-byte chunk. To efficiently provide the full tag bits of chunk-tags, we leverage the original tag bits for each 128-byte cache line and use them as a common-tag. We assume that the full address space is composed of 32 bits (4GB), as shown in Figure 4. For a conventional 48KB cache consisting of 64 sets, each of which containing

Fig. 4. Division of address bits and common- and chunk-tags fields for 6-way 48KB Elastic-Cache.

Fig. 5. An L1 cache line is divided into four 32-byte chunks (CK) associated with one common-tag (CT) and four chunk-tags (CKT).

six 128-byte cache lines, the default length of set-index and byte-index are composed of 6 and 7 bits, respectively. Then the remaining 19 bits are used as tags.

In Figure 4, as each cache line contains four 32-byte chunks, we borrow the upper 2 bits from the byte-index to compose the lower 2 bits of a 16-bit chunk-tag. The lower 2 bits denoted as lower-tag is used to index one of the four 32-byte chunks in a cache line. The upper 14 bits denoted as upper-tag are obtained from the lower 14 bits of the 19-bit tag. These two parts are compared separately when accessing L1 cache. The results are upper-result and lower-result, respectively. The remaining upper 5 bits of the 19-bit tag is used as common-tag. In summary, a 128-byte cache line has one 19-bit tag (where 5 bits are used as a common-tag when supporting fine-grained cache line managements) and four 16-bit chunk-tags. Note that the four upper-tags located in the four chunk-tags of a 128-byte request are the same and common-tag+upper-tag is equivalent to the 19-bit tag of the baseline L1 cache. With a 5-bit common-tag, a cache line of Elastic-Cache can store any 32-byte chunks in an address space of 2^{32-6-5} bytes (2M bytes or 64K 32-byte chunks), where 6 and 5 represent the bit width of set-index and common-tag fields, respectively. That is, whether 32-byte chunks are in contiguous memory space or not, as long as they have the same common-tag, they are qualified to be stored into any of the four 32-byte chunk slots in a cache line indexed by the same set-index. Note that if more bits are used for addressing for larger memory space, then we just need to extend the common-tag, and the chunk-tag is not affected and does not need to change.

4.2 Basic Cache Operations of Elastic-Cache

We use one bit to indicate whether a cache line is used by a 128-byte block or four 32-byte chunks. As shown in Figure 5, 0 and 1 are used for four 32-byte chunks and a 128-byte block, respectively.

And the four continuous chunk-tags of a 128-byte block are stored in the unused shared memory as well. All common-tags and chunk-tags of a set are used for comparison simultaneously in Elastic-Cache. In addition, one bit is assigned for each request to identify request size when it is generated after coalescing from the Load/Store unit. For a 32-byte request, a hit can be found in either a 32-byte chunk or a 128-byte block. For a 128-byte request, a hit can be found only in a 128-byte block.

When a 32-byte request accesses a cache line, both `upper-result` and `lower-result` are required to determine a cache hit. We consider a 32-byte access as a hit only when its common- and chunk-tags are matched with one common-tag in a set and one chunk-tag that belongs to the cache line associated with the matched common-tag, respectively. For example, as depicted in Figure 5, a 32-byte request (Request-A) to address `0x040000C0` needs to access L1 cache. The common- and chunk-tags of Request-A are `0x01` and `0x0002`, respectively. Then all common-tags (CT-0 to CT-5) and chunk-tags (CKT-00 to CKT-53) associated with every common-tag are used for comparison. And finally, CT-1 is hit for common-tag and CKT-11 is hit for chunk-tag; CT, CKT, and CK denote common-tag, chunk-tag, and chunk, respectively. Subsequently, Request-A accesses CK-11. However, for another 32-byte request (Request-B) to address `0x10000080`, its common-tag (`0x04`) is matched with CT-3. But its chunk-tag (`0x0000`) is not matched with any chunk-tags (CKT-31 – CKT-33) in the cache line associated with CT-3. In this case, a miss occurs and subsequently one chunk in this set should be chosen and evicted based on a replacement policy (see Section 4.5 for replacement policy). If the common-tag of a (32-byte) request is not matched with any common-tags in the set, then an entire cache line should be evicted based on a replacement policy.

When a 128-byte request accesses a cache line that stores a 128-byte block, the four upper-tags of the four chunk-tags of this 128-byte request are the same. Consequently, we only use the upper-result to determine a cache hit for a 128-byte request. A 128-byte access is considered a hit only when its common-tag and upper-tag are matched with one common-tag in a set and the corresponding upper-tag of the cache line associated with the matched common-tag, respectively, since the 5-bit common-tag + 14-bit upper-tag is equivalent to the 19-bit tag in the baseline L1 cache. As shown in Figure 5, a 128-byte request (Request-C) has a common-tag `0x23`, which is matched with the common-tag of a cache-line (CL-5) storing a 128-byte block. We ignore the lower 2 bits (lower-tag in Figure 4) of the chunk-tags when comparing chunk-tags and the upper-tag denoted as UT is matched. Finally, Request-C hits in L1 cache. At the same time, its four chunk-tags are also stored in the unused shared memory so subsequent 32-byte accesses to the cache line can evict and replace 32-byte chunks of this block.

4.3 Elastic-Cache Architecture

In this section, we describe how the chunk-tags are stored in the unused shared memory and the architecture of Elastic-Cache.

4.3.1 The Storage of Chunk-tags of Elastic-Cache in Shared Memory. In fine-grained cache line management mode, Elastic-Cache simultaneously reads n common-tags (in the original tag array) and $n \times 4$ chunk-tags (in the unused shared memory) for an access where n is the associativity of L1 cache. Then the common- and chunk-tags of the access are compared against all n common-tags and $n \times 4$ chunk-tags to determine a cache hit.

The shared memory is composed of 32 banks, each of which has a 32-bit I/O. The banks can be managed independently. Specifically, GPUs can issue up to 32 requests to 32 banks even as they access different rows. To read all chunk-tags of one set at a time, we must make chunk-tags of a set stored across different banks of the shared memory. For a 6-way set associative cache, there are in total 24 16-bit chunk-tags in a set. Therefore, 12 banks of the shared memory are needed to store

Fig. 6. The storage of chunk-tags in the shared memory for Elastic-Cache. Chunk-tags of a set composed of 6 ways are stored across 12 banks, and hence all of them can be accessed simultaneously.

(a) Architecture

(b) Flow chart

Fig. 7. Architecture and flow chart of Elastic-Cache.

the chunk-tags of a set. Figure 6 depicts how chunk-tags are stored in the shared memory. Note that chunk-tags of set-2 and set-5 are stored in different rows, but current GPUs are still able to read them at the same time as long as they are located in different banks. For a 48KB L1 cache, we need in total 3KB ($= 16 \times 64(\text{sets}) \times 6(\text{ways}) \times 4(\text{chunks})$ bits) of unused shared memory to store all chunk-tags.

4.3.2 Cache Architecture. The architecture of Elastic-Cache is shown in Figure 7(a). The cache access is divided into two steps: Step-1 is tag comparison and Step-2 is data access. To support fine-grained cache line management, we divide every 128-byte cache line into four 32-byte chunks, each of which has a fixed chunk-ID (Figure 7(a)①). The associativity of L1 cache is kept unchanged. In Step-1, set-index is used as the address of accessing common-tags stored in the original tag array, and chunk-tags stored in the unused shared memory. After comparison, hit or miss is determined based on the process described in Section 4.2 (Figure 7(a)②). We also

get the line-index, which is the index of the matched common-tag and is used as the address of accessing the data array (Figure 7(a)③). In addition, chunk controllers are used to (1) select the chunk that is going to be accessed for 32-byte requests and (2) select all four chunks of a particular cache line for 128-byte requests. When 32-byte requests access L1 cache, chunk-index is selected to compare with the chunk-ID of every chunk. If the chunk-ID equals the chunk-index of the 32-byte request, then the output of the chunk controller is 1. If ② in Figure 7(a) is also 1, then ③ in Figure 7(a) is enabled so the corresponding chunk can be accessed. If the chunk-ID does not equal the chunk-index of the 32-byte request, then the output of the chunk controller is 0, which represents that the chunk cannot be accessed. When 128-byte requests access L1 cache, chunk-ID is selected and thus the output of each chunk controller is always 1, which represents all four chunks are going to be accessed if ② in Figure 7(a) is also 1. The access flow of Elastic-Cache is shown in Figure 7(b).

4.4 Elastic-Plus Architecture

4.4.1 Motivation of Elastic-Plus. Elastic-Cache aims to improve the cache efficiency by supporting both fine- and coarse-grained cache line management. In GPUs, the bandwidth of L1 cache is 128 bytes per cycle, which is sufficient to service a 128-byte request in one cycle. However, with fine-grained cache line management of Elastic-Cache, only 32 bytes of a cache line are required for one request. That is, the bandwidth utilization of L1 cache is only 25% for 32-byte requests. However, the Load/Store unit can process only one memory instruction for one warp at a time. Consequently, other memory instructions have to wait till the current instruction in the Load/Store unit is completed. However, as shown in Figure 2, a memory instruction may generate multiple requests, depending on the locality among threads, especially in applications on the left side. For such memory instructions, multiple cycles are needed, as the memory requests are processed one-by-one. Based on the above insight, we further propose Elastic-Plus, which can parallel-issue multiple 32-byte requests per cycle to L1 cache. On the one hand, parallel issue can efficiently utilize the high bandwidth provided by L1 cache in fine-grained cache line management mode. On the other hand, parallel issue of requests also reduces the processing latency of memory instructions, which eventually increases the throughput of the Load/Store unit. To achieve this goal with small overhead, we modify the pattern of how chunk-tags are stored in the unused shared memory and the architecture of Elastic-Cache.

4.4.2 The Storage of Chunk-tags of Elastic-Plus in the Shared Memory. In Elastic-Cache, common-tags and chunk-tags are accessed in parallel and hence the comparison can be accomplished in one cycle. Since Elastic-Plus can issue up to four 32-byte requests simultaneously, four groups of tag comparison logic are needed. If common-tags and chunk-tags are still accessed in parallel as done in Elastic-Cache, then it requires that the tag comparison logic shown in Figure 7(a) are duplicated four times for Elastic-Plus. To reduce the overhead of the comparison logic, we propose to compare common-tags and chunk-tags serially. Specifically, common-tags are compared first, then we only need to access chunk-tags that associated with common-tags that are matched with the requests.

Furthermore, to support parallel issue of requests that may access different sets, chunk-tags of these sets also need to be accessed in parallel. However, accessing chunk-tags in parallel from multiple sets may lead to conflicts, according to the chunk-tag storage shown in Figure 6. Therefore, we need to make sure that the required chunk-tags of these sets are stored across different banks to avoid conflicts. That is, the storage position of chunk-tags in the shared memory needs to be rearranged for Elastic-Plus. We divide the 32 banks of the shared memory into 4 groups, as shown in Figure 8. Unlike the chunk-tag storage in Elastic-Cache, the 24 16-bit chunk-tags of

Fig. 8. The storage of chunk-tags in the shared memory for Elastic-Plus. Chunk-tags of a set composed of 6 ways are stored in 3 rows across 4 banks. Thus, 32 banks can store chunk-tags of 8 sets in 3 rows.

Table 1. Encode Table of Priority Encoder

priority encoder0		priority encoder1		priority encoder2		priority encoder3	
input(⑤)	output	input(⑥)	output	input(⑦)	output	input(⑧)	output
1###	①	#1##	②	##1#	③	###1	④
0 1##	②	#0 1#	③	##0 1	④	1 ##0	①
0 0 1#	③	#0 0 1	④	1 #0 0	①	0 1 #0	②
0 0 0 1	④	1 0 0 0	①	0 1 0 0	②	0 0 1 0	③

a set in Elastic-Plus are stored in 3 consecutive rows of 4 banks so chunk-tags of 8 sets can be stored independently. The throughput of a bank group is 4 chunk-tags (which correspond to one common-tag) per cycle. Consequently, if a request gets two or more hits for common-tags, then the accesses to the corresponding chunk-tags are processed serially and accesses to chunk-tags from other requests cannot be issued until current accesses are accomplished. However, we observe that the situation in which a request hits two or more common-tags seldom happens in benchmarks we evaluated.

4.4.3 Cache Architecture. The architecture and corresponding access flow of Elastic-Plus are shown in Figure 9. Unlike Elastic-Cache, there are three steps in Elastic-Plus for a request, which are common-tag comparison (Step-1), chunk-tag comparison (Step-2), and data access (Step-3), respectively. Furthermore, since up to four groups of chunk-tags need to be accessed for four 32-byte requests, we place the requests generated by a memory instruction in four queues instead of one queue of Elastic-Cache according to the queue-indices of requests, which are determined by the result of set-index MODE 4. Each queue can issue a 32-byte memory request every cycle. Note that only one 128-byte request is allowed to issue among four queues in Elastic-Plus. Common-tags are compared first. If no common-tags are matched for a request, then we can assert that a hit is impossible to happen. Therefore, there is no need to access chunk-tags and data. If common-tags are matched for a request, then it is likely that a hit can happen, which depends on the comparison result of chunk-tags. We use the line-index obtained from the matched common-tag as the address of accessing chunk-tags (Figure 9(a)①②③④). After the comparison of chunk-tags, the result is sent to the chunk selector (Figure 9(a)⑤⑥⑦⑧). The chunk selector is used to generate the control signal for the priority encoder. Each chunk selector has a fixed chunk-ID. Its inputs also include the chunk-indices of the four requests (Figure 9(a)①②③④), which are used to compare against the chunk-ID. The comparison result (1 or 0) is the input A of the multiplexer. If the chunk-tag is not matched (Figure 9(a)⑤⑥⑦⑧ are 0), then the output of the multiplexer is 0 (input C). If the chunk-tag is matched and the request size is 128-byte, then the output of the multiplexer is 1 (input B). If the request size is 32-byte, then the output of the multiplexer is A. The outputs of the four multiplexers form a 4-bit control signal (Figure 9(a)⑤⑥⑦⑧) that is used to select the valid address of accessing data arrays from the four line-indices (Figure 9(a)①②③④) in the priority encoder. Table 1 lists the encode table of every priority encoder. # means that both 1 and 0 are eligible.

(a) Architecture

(b) Flow chart

Fig. 9. Architecture and flow chart of Elastic-Plus.

4.5 Replacement Policy and Cache Coherence

Elastic-Cache/Plus uses a hierarchical replacement policy to evict a (128-byte) cache line and/or a (32-byte) chunk upon a miss. When a miss occurs for a 128-byte request, an entire 128-byte cache line is chosen and evicted based on the pseudo least recently used (LRU) maintained for four 128-byte cache lines per set, which is the same as the baseline L1 cache. We have two possible eviction scenarios when a miss occurs for a 32-byte request: First, when a common-tag of a cache line of an indexed set is matched with the common-tag of a 32-byte request, we need to choose a chunk to evict in that cache line. To obviate the area and timing overhead to offer the (expensive) LRU replacement policy for four 32-byte chunks per cache line, we propose a simple replacement policy. That is, we maintain 2 bits per cache line to indicate the most recently used (MRU) 32-byte chunk; we simply invert these two bits to choose a 32-byte chunk to evict, which avoids evicting the MRU chunk. Second, when no common-tag of all cache lines of an indexed set is matched with the common-tag of a 32-byte request, we need to choose a 128-byte cache line to evict based on the LRU policy and place the fetched 32-byte chunk in the first chunk slot of the cache line. Last, we update both the LRU cache line state of a set and the MRU chunk state of a cache line for each 32-byte request, whereas we update only the LRU cache-line state for each 128-byte request.

Our design takes a simple approach to deal with cache coherence. Cache coherence problems occur with a partial miss, in which only a subset of chunk-tags are matched with the chunk-tag portion of an incoming 128-byte coalesced request. Thus, some of the requested data may be present in different chunk locations. Rather than carefully orchestrate the updates across all the partial chunks, we take a simple approach. Whenever a partial miss is encountered, then we first invalidate all the existing 32-byte chunks in cache lines, after writing back any dirty 32-byte blocks. Then, we reissue the 128-byte coalesced request as a new request to the memory pipeline that will trigger a full cache line miss. Then the cache miss is handled as a regular cache miss in the rest of the memory pipeline. Our experiment shows that 0.5% requests encounter such partial misses with 48KB L1 caches. Last, we still support partial read and write hits of 32-byte chunks to 128-byte cache lines brought by misses of 128-byte requests.

5 METHODOLOGY

We evaluate 24 benchmarks from Polybench, Mars, Lonestar, and some emerging GPGPU benchmark suites [8, 10, 21, 35]. We classify these applications into two categories. The first category includes 12 benchmarks with more than four requests per memory instruction on average or cache efficiency that is lower than 50% (cf. Figure 2 and Figure 3 in Section 3.1): Page View Count (PVC), Page View Rank (PVR), Similarity Score (SS), Word Count (WC), Inverted Index (II), String Match (SM) from Reference [10]; BiCGStab Linear Solver (BICG), Symmetric rank-2k operations (SYR2K), Symmetric rank-k operations (SYRK), Correlation Computation (CORR) from Reference [8]; Survey Propagation (SP) from Reference [21]; Graph Coloring (GCO) from Reference [35]. We call the benchmarks in the first category “irregular applications,” since these benchmarks usually have irregular memory access patterns mostly incurred by branch or memory divergences. The second category includes the remaining 12 benchmarks: All Pairs Shortest Path (APSP), Connected Component Labeling (CCL), Graph Cuts (GCU), Maximal Independent Set (MIS) from Reference [35]; 2 Matrix Multiplications (2MM), 3 Matrix Multiplications (3MM), 2-D Convolution (2DCON), 3-D Convolution (3DCON), Finite Different Time Domain (FDTD), Matrix-multiply C (GEMM), Gram-Schmidt decomposition (GRAM) from Reference [8]; and Single Source Shortest Path (SSSP) from Reference [21]. We call the benchmarks in the second category “regular applications.”

We use GPGPU-Sim (version 3.2.2) [1, 17] for evaluation. We configure the simulator to model a GPU similar to NVIDIA’s GTX480: the number of SMs (15), the shared memory capacity (48KB), L1 cache capacity (48KB), L1 cache throughput (128 bytes per cycle [32]), L1 cache write policy (write eviction [25, 30]), L1 cache hit latency (2, 2–3, and 1–6 cycles for the baseline, Elastic-Cache, and Amoeba-Cache), L2 cache capacity (128KB per DRAM channel), L2 cache latency (120 cycles), the number of DRAM channels (6), and DRAM latency (220 cycles). Note that we model independent shared memory (48KB) and L1 cache (48KB) instead of the configurable structure shared by the shared memory and L1 cache in GTX480 GPUs as the shared memory and L1 cache has been separated since Maxwell GPU architecture in 2014 [24].

6 EXPERIMENTAL RESULTS

6.1 Performance

6.1.1 Reducing Cache Line Size. The performance is represented by instructions per cycle (IPC). Figure 10 shows the performance impact of using a narrower width cache line design to begin with. Recall that narrow cache lines may tackle the underutilization problem but it can hurt the performance of regular applications as more fine-grained requests are generated from coarse-grained requests comparing to the request number in the baseline GPU, which may increase the traffic in the memory system (cf. Section 3). Another shortcoming is that this design breaks the spatial

Fig. 10. Normalized IPCs when using cache lines of 32 bytes and 64 bytes.

Fig. 11. Normalized IPCs for Elastic-Cache with different chunk-tag widths.

locality among coalesced requests. Therefore, some regular applications suffer from the degradation of IPCs especially when the size of cache line is 32 bytes, as shown in Figure 10(b).

Overall, the geometric-mean (GM) improvements with 64- and 32-byte cache lines (64B-CL and 32B-CL) for irregular applications are 52% and 55% over the IPC of the baseline cache (Base-Cache), respectively. However, for regular applications, the normalized IPCs of 64B-CL and 32B-CL are only 88% and 65%, respectively. Thus, it is critical to explore Elastic-Cache, which preserves the wide cache line design when an application is able to take advantage of it, and dynamically transforms it to a narrow cache line design when the application cannot fully utilize the wide cache lines. In other words, Elastic-Cache is explored to increase the IPCs for irregular applications without degrading the IPCs for regular applications.

6.1.2 Chunk-tag Width. To evaluate the impact of chunk-tag widths of Elastic-Cache on IPCs, we show the results for different chunk-tag widths, denoted by Elastic-N, in which N is the number of bits used for chunk-tags in Figure 11. As shown, the IPCs are improved by increasing the width of chunk-tags. Note that Elastic-32 does not show significant differences compared with Elastic-16 despite the fact that the overhead of Elastic-32 is twice over that of Elastic-16, indicating 16 bits are sufficient to manage fine-grained accesses. The IPCs for most regular applications do not change very much when applying Elastic-Cache, as shown in Figure 11(b). That is, Elastic-Cache recovers the degradation exhibited by regular applications in Figure 10(b), as it is able to preserve the 128-byte cache line design for these regular applications, which in turn curtails unnecessary memory traffic.

6.1.3 Different Cache Architectures. We also compare Sector-Cache and Amoeba-Cache with Elastic-16 and Elastic-Plus in Figure 12. In contrast to Amoeba-Cache, Elastic-16 and Elastic-Plus store the extra tags in the unused shared memory without sacrificing the capacity of data array. Furthermore, Amoeba-Cache in GPUs requires longer access latency than Elastic-16 and Elastic-Plus (cf. Section 2). Comparing with Base-Cache, the improvement of Elastic-16 mainly comes from two aspects: One is the improved L1 cache hit rate, which is shown in

Fig. 12. Normalized IPCs for Sector-Cache, Amoeba-Cache, Elastic-128B, Elastic-16, and Elastic-Plus.

Fig. 13. Normalized IPC of Elastic-16 with different replacement policies.

Section 6.2, and the other is the fewer data transfers between L1 and L2 caches. To understand where the improvement of Elastic-16 comes from, we manually set the size of all requests transferred between L1 cache and L2 cache to 128 bytes in the simulator, and this approach is denoted by Elastic-128B. This allows us to exclude the benefits of fewer data transfers for Elastic-16. That is, we see the benefits of reduced cache miss rates only. For irregular applications, Sector-cache and Amoeba-Cache give the geometric-mean performance improvements of 36% and 32%, respectively, whereas Elastic-16 and Elastic-Plus provide the performance improvements of 104% and 131%, respectively. Excluding the benefits of fewer data transfers, Elastic-128B offers the performance improvement of 62% for irregular applications. The performance for regular applications is shown in Figure 12(b). As observed, the performance is decreased by 17% when applying Amoeba-Cache, as Amoeba-Cache uses cache lines to store tags for fine-grained accesses, resulting in less capacity for data storage, and has to take more time to access more cache lines to find a hit. The performance with Elastic-16 is 107%. Elastic-Plus has no notable improvement comparing to Elastic-16, since most of the requests are 128 bytes in regular application (cf. Figure 3), resulting in fewer opportunities for parallel issue.

6.1.4 Replacement Policies. To observe the impact of replacement policies of Elastic-Cache and Elastic-Plus on performance, we run experiments with multiple replacement policies, as shown in Figures 13 and 14. As mentioned in Section 4.5, Elastic-Cache use a hierarchical replacement policy in which a cache line is selected based on LRU, and a chunk within a cache line is selected based on Non-MRU. In Figures 13 and 14, this policy is denoted by LRU-NMRU. Full-LRU represents the policy that all cache lines and chunks are ordered together based on LRU. LRU-LRU means that a cache line is selected based on LRU and then a chunk within this cache line is selected through LRU as well. LRU-FIFO selects a cache line based on LRU and then selects a chunk based on first-in first-out (FIFO). LRU-Random is the policy that a cache line is selected via LRU and then a chunk is selected randomly. Full-LRU, LRU-FIFO, and LRU-LRU have more hardware cost than LRU-NMRU and LRU-Random do. In summary, the average performance improvements with different

Fig. 14. Normalized IPC of Elastic-Plus with different replacement policies.

Fig. 15. L1 cache miss rate.

replacement policies do not have notable differences. However, as observed, LRU-Random performs better than other policies on BICG, SYR2K, and SYRK. We find that BICG prefers to replace the MRU chunk that is more likely to be replaced in LRU-Random. For SYR2K and SYRK, stalls incurred by burst access are reduced by LRU-Random.

6.2 L1 Cache Miss Rate

The cache miss rates of L1 caches are shown in Figure 15. Fine-grained management of cache lines can accomplish lower cache miss rates when spatial locality is poor among requests. For irregular applications, Elastic-16 and Elastic-Plus decrease the average cache miss rate by 21%, whereas Amoeba-Cache decreases it by 19%. In Figure 15(a), we observe that the cache miss rate is significantly reduced in II, GCO, SYR2K, and SYRK. Because of fine-grained cache line management, more space can be provided for 32-byte requests by Elastic-16/Plus. As a result, fewer evictions occur in L1 cache. However, the cache miss rates of BICG do not change very much with Elastic-16/Plus, although the performance is notably improved. We observe that part of the IPC improvement of BICG results from the fewer data transfers between L1 and L2 caches, while another reason for the improvement is that Elastic-16/Plus provides more fine-grained accessible cache blocks for 32-byte requests, which reduces stalls introduced by burst accesses to the same cache set and hence improves the cache efficiency.

6.3 L1 Cache Efficiency

Figure 16 shows the cache efficiencies. The cache efficiencies of most benchmarks are improved by applying Elastic-16 and Elastic-Plus, comparing with the cache efficiencies of the baseline GPU. Note that Amoeba-Cache shows slightly higher cache efficiency than Elastic-16/Plus does especially in SM, CORR, and APSP. Amoeba-Cache supports 32-, 64- and 128-byte cache lines, whereas we make Elastic-16/Plus support only 32- and 128-byte cache lines and treat 64-byte requests as 128-byte, although Elastic-16/Plus is able to support 64-byte chunks in principle. See the second paragraph of Section 4 for this decision. Consequently, Amoeba-Cache can evict a 64-byte

Fig. 16. L1 cache efficiency.

Fig. 17. Normalized energy consumption.

cache line, which is more efficient compared to evicting a 128-byte cache line in Elastic-16/Plus. In addition, 64-byte requests are responsible for a large fraction of total memory requests in APSP, as shown in Figure 3, which also makes the cache efficiency of Amoeba-Cache higher. Overall, Elastic-16, Elastic-Plus, and Amoeba-Cache improve cache efficiency by 41%, 42%, and 44% for irregular applications, respectively.

6.4 Energy Consumption

Figure 17 shows the energy consumptions that are normalized to that of Base-Cache. Since Amoeba-Cache incurs additional cache accesses, the energy is increased except for GCO, SYR2K, SYRK, and BICG. The reason behind this is that the total execution time is significantly degraded by Amoeba-Cache in these four benchmarks (as shown in Figure 12) due to the reduced cache miss rate or increased cache efficiency. On the contrary, Elastic-16 reduces the energy consumption for irregular applications, since the performance is improved. The energy consumption of Elastic-Plus is almost the same with that of Elastic-16, because the amount of work done by Elastic-16 and Elastic-Plus is almost equivalent. Overall, the energy consumption for irregular applications is 107%, 57%, and 56% with Amoeba-Cache, Elastic-16, and Elastic-Plus, respectively.

6.5 Hardware Overhead and Lookup Latency

Elastic-16/Plus requires no extra space to store chunk-tags. A chunk needs 2 bits for status and a 128-byte cache line needs 2 bits to index the MRU chunk and 1 bit to indicate whether it is used by a 128-byte block or four 32-byte blocks. The total overhead of this storage is only 0.5% of the shared memory and L1 cache in the baseline GPU. Besides, as shown in Figure 7, Elastic-16 needs 4 chunk controllers, 24 16-bit comparators, 4 AND gates, and 3 groups of 3-state gates in each SM. We synthesize the chunk controller and the 16-bit comparator in 40nm technology. The areas are $7\mu\text{m}^2$ and $31\mu\text{m}^2$, respectively. We simulate the access latency of Elastic-16 in two cycles (two steps shown in Figure 7(a)). For Elastic-Plus, common-tag is compared first, then

four chunk-tags associated with each common-tag are compared. Therefore, 16 16-bit comparators for chunk-tags are needed. In addition, four chunk selectors (16 multiplexers) and four priority encoders, as shown in Figure 9, are needed to control the access of chunks. The areas of a chunk selector and a priority encoder are $30\mu m^2$ and $16\mu m^2$, respectively. The total area of this extra hardware is only 0.6% of the area of a 48KB L1 cache, which is estimated by CACTI6.5 with 40nm technology [23]. The lookup latency of Elastic-Plus we simulated is three cycles (three steps), as shown in Figure 9(a). In brief, the extra hardware overhead of Elastic-16 or Elastic-Plus is negligible, compared to the hardware of an entire GPU chip.

7 DISCUSSION

7.1 Other Usages of Shared Memory

7.1.1 Using Unused Shared Memory as L1 Cache. It has been demonstrated that the capacity of L1 cache can significantly impact the performance of cache-sensitive applications [27]. In addition, utilizing the unused register file as L1 cache can also improve the performance of cache-sensitive applications [13]. In this article, we also run experiments that use the unused part of the 48KB shared memory as L1 cache. We observe that the performance is improved by 76%, compared to the baseline L1 cache for irregular applications. Furthermore, the shared memory can also be designed to work in Elastic mode to support fine-grained accesses, which will further improve the efficiency of the shared memory and overall performance.

7.1.2 Using Unused Shared Memory to Store Context Information. For applications that seldom use the shared memory, the shared memory can also be used to store temporary context information. For example, to compact divergent threads, the relevant registers of divergent threads can be collected in a warp-specific stack allocated in the shared memory and restores the registers only when the perfect utilization of warp lanes becomes feasible [15]. To maximize the thread parallelism by assigning threads up to the register file limit instead of the scheduling limit [37], the context information of thread blocks that are currently not considered for scheduling can be stored in the shared memory temporarily. Moreover, the shared memory can also be managed as register file to increase the thread parallelism.

7.1.3 Managing Software-controlled Shared Memory. For applications in which the thread parallelism is limited by the capacity of the shared memory, the current shared memory management reserves shared memory too conservatively for the entire lifetime of a thread block. If the shared memory is allocated only when it is actually used and freed immediately after, then more thread blocks can be hosted in an SM without increasing the shared memory capacity [36]. However, warp scheduling policy can also be modified to give thread blocks that are using the shared memory higher scheduling priority. Thus, the shared memory can be released more quickly.

7.2 Impact on Shared Memory Programming and Requests

7.2.1 Shared Memory Programming. In GPUs working with Elastic-Cache/Plus, using the shared memory as chunk-tags for L1 cache is transparent to programmers. To keep the shared memory software-controlled for programmers, we give the usage of the software-controlled shared memory higher priority over the usage of chunk-tags. Specifically, the compiler computes how much shared memory is used by software first, before applications run, then the rest is used to store chunk-tags. If the unused part is insufficient to store all chunk-tags, then we just allow part of cache sets to work as Elastic-Cache/Plus. Based on the chunk-tag storage of Elastic-Cache shown in Figure 6, the number of cache sets that can work in fine-grained mode is determined by: $\min(N_{set}, \lfloor \frac{S_{USM}}{S_{CKT/set}} \rfloor)$, where N_{set} is the total set number of L1 cache (e.g., 64 in this article), S_{USM} is

the size of the unused shared memory, $S_{CKT/set}$ is the size of chunk-tags per set (e.g., 48 bytes in this article). For Elastic-Plus, since chunk-tags are stored in three rows across four banks, the number of cache sets that can work in fine-grained mode is determined by: $\min(N_{set}, \lfloor \frac{S_{USM}}{S_{CKT/set} \times 8} \rfloor \times 8)$. In brief, the allocation of the shared memory can be handled automatically. Thus, programmers do not need to worry about the programming or modify any codes regarding the shared memory in applications.

7.2.2 Shared Memory Requests. In GPUs, memory instructions accessing the software-controlled shared memory are also processed by the Load/Store unit [18]. Specifically, the Load/Store unit cannot service other memory instructions until current memory instruction is dispatched. As common-tags and chunk-tags stored in the shared memory are accessed simultaneously in Elastic-Cache, no delay occurs for shared memory instructions. In Elastic-Plus, chunk-tag access is decoupled from common-tag access. If the subsequent memory instruction needs to access the shared memory, then one cycle delay is required. Suppose future GPUs have two or more Load/Store units so shared memory requests and L1 cache requests can be dispatched simultaneously; chunk-tag access and normal shared memory access can be processed simultaneously as long as no bank conflicts exist. If bank conflicts happen, then the shared memory access is serviced first to guarantee the low latency.

7.3 Separate Chunk-Tag Space

As mentioned in Section 4.3, 3KB space in the shared memory is needed to store chunk-tags for a 48KB L1 cache. Another option is to reduce the size of the shared memory and place the 3KB chunk-tags in a separate tag array for fine-grained access. The advantage of this design is that the chunk-tag storage is more flexible, which makes accesses to chunk-tags more efficient. For instance, the shared memory and L1 cache can be accessed simultaneously in GPUs that have two or more Load/Store units. The disadvantage is that shrinking the size of the shared memory may reduce the number of thread blocks that can be launched to an SM, which eventually impacts the parallelism and performance of applications that are hungry for the shared memory. In addition, the capacity of the chunk-tag array is unscalable. If the cache size is decreased, then some space of the chunk-tag array is wasted.

7.4 Supporting Multiple Chunk Sizes

Elastic-Cache/Plus is able to support multiple chunk sizes. In this article, because only 3% of requests are 64 bytes on average, as shown in Figure 2, Elastic-Cache/Plus works in fine-grained mode only for 32-byte requests. If 64-byte and even 96-byte requests are also very common (96-byte requests, in fact, are not supported in current GPUs, as 96 bytes are not aligned), then we need to assign 2 bits to each request to identify its size when it is generated from the Load/Store unit. To support multiple request sizes, one approach is to divide 64-byte and 96-byte requests into 32-byte requests. For Elastic-Cache, this approach generates more individual requests, which may result in performance degradation for regular applications, as shown in Figure 10(a). However, Elastic-Plus can alleviate the request pressure because of the advantage of parallel issue.

Another way is to make cache lines in Elastic-Cache/Plus support three chunk sizes (32 bytes, 64 bytes, 96 bytes). In this case, there are in total eight possible combinations in cache lines: 128B, 32B + 32B + 32B + 32B, 64B + 64B, 64B + 32B + 32B, 32B + 64B + 32B, 32B + 32B + 64B, 32B + 96B, 96B + 32B. Consequently, 3 extra bits are used to represent the construction of a cache line. Chunks in a cache line still share one common-tag. 64-byte and 96-byte chunks also save chunk-tags in 32-byte granularity, as a 128-byte cache line does. We make decisions about hit or miss for different chunk sizes by checking comparison results and the construction of this cache line.

Specifically, a hit occurs only when tags are matched and the request size is no larger than the matched chunk. In addition, Elastic-Plus needs to be smarter to choose requests that can be issued in parallel. Specifically, requests that can be issued in parallel must match one of the eight chunk combinations. If the sum of request sizes is larger than the cache line size, then it makes no sense to issue them in parallel due to chunk access conflicts. Regarding the hardware overhead, the size of all extra bits is $N_{CL} \times \log_2 N_{CKC}$, where N_{CL} is the total number of cache lines and N_{CKC} is the number of chunk combinations.

7.5 Interactions with L2 Cache

7.5.1 Hit and Miss in L2 Cache. In this article, L2 cache works as a conventional cache and the size of L2 cache line is the same as the size of L1 cache line (128 bytes). In Base-Cache, a 128-byte (L1 cache line) request is sent to L2 cache when a miss happens in L1 cache and a 128-byte data is accessed when a hit occurs. If it misses in L2 cache, then a 128-byte (L2 cache line) request is sent to access the data in DRAM. With Elastic-Cache/Plus, the size of requests sent to L2 cache from L1 cache may be 32 bytes or 128 bytes, and hence the data accessed is also 32 bytes or 128 bytes, which is different from the situation in Base-Cache. However, the situation of miss in L2 cache is the same as that in Base-Cache, as the size of requests sent to DRAM is the size of L2 cache line as well.

7.5.2 L2 Cache Efficiency. Unlike L1 cache efficiency, which is measured using all requests, L2 cache efficiency is measured using requests that miss in L1 cache. Since the miss rates of L1 cache are different, the cache efficiencies of L2 cache working with Elastic-16 and Elastic-Plus are 5% and 4% higher than the cache efficiency of L2 cache working with Base-Cache, respectively. As shown in Figure 15, the amount of requests that miss in Elastic-16 and Elastic-Plus is much less than that of Base-Cache, which results in fewer early evictions in L2 cache. Note that although requests are issued to L1 cache in parallel with Elastic-Plus, L2 cache still serially fetches requests issued from L1 cache via network on-chip (NoC).

7.6 Write Policies

7.6.1 Write Eviction. GPUs adopt write eviction policy for L1 cache [25, 30]. When a write miss happens in L1 cache, the write request is directly sent to L2 cache without allocating any cache lines in L1 cache (write-no-allocate). When a write hit happens, the corresponding cache line is invalidated and the data is written to L2 cache. Therefore, if a request evicts a cache line that is composed of four 32-byte chunks or a partial miss mentioned in Section 4.5 invalidates some chunks in Elastic-Cache/Plus, then no write back is needed, since no chunks can be dirty.

7.6.2 Write Through. Suppose that L1 cache adopts write-through policy. When a write miss happens in L1 cache, the write request is directly sent to L2 cache without allocation, which is the same as the scenario of write eviction. When a write hit happens, the corresponding cache line is modified and the write request is also sent to L2 cache. Therefore, if a request replaces four dirty chunks or a partial miss invalidates dirty chunks in Elastic-Cache/Plus, then we do not need to write back these chunks as well, since there have been backup data for these chunks in L2 cache.

7.6.3 Write Back. If L1 cache uses write-back policy, then when a miss happens in L1 cache, a cache line needs to be allocated for this request. If the cache line is dirty, then we should write the previous data of this cache line back to L2 cache first. And this cache line is marked as dirty if it is a write operation. When a write hit happens, the data is directly written to the corresponding cache line. In Elastic-Cache/Plus, if a request evicts four dirty chunks or a partial miss invalidates dirty chunks, then the dirty chunks need to be written back as independent requests (burst requests). For

Elastic-Cache, because the input buffer on the SM side of NoC receives one request per cycle, the burst requests have to be processed serially, which incurs extra cycle overhead. For Elastic-Plus, the input buffer on the SM side of NoC is divided into four sub-buffers, each of which maps to one of the four chunks of a cache line and receives one request per cycle. NoC fetches one request from one of the four sub-buffers every cycle in round-robin fashion and routes it to the output buffer on the L2 cache side. Consequently, when a request evicts four dirty chunks, they are sent to their corresponding sub-buffers without any conflicts. However, when a partial miss invalidates some dirty chunks, extra cycles are required to process them serially if some of these chunks map to the same sub-buffer.

7.7 Different Cache Organizations

7.7.1 Cache Line Size. One factor that can constrain the maximum number of requests issued in parallel (N_{PI}) is $\frac{S_{CL}}{S_{CK}}$, where S_{CL} is the size of a cache line and S_{CK} is the size of a chunk. However, to get all chunk-tags of a cache line at a time, chunk-tags of a cache line have to be stored in the same row of the shared memory. We assume the organization of the share memory is not changed (32 32-bit banks); the maximum N_{PI} is also constrained by the following formula: $\frac{32 \times 32 \text{ bits}}{N_{PI}} = N_{PI} \times W_{CKT}$, where W_{CKT} is the width of chunk-tags. In this article, W_{CKT} is 16 bits, hence the maximum N_{PI} is 8. If N_{PI} is larger than 8, then more cycles are required to access all chunk-tags of a cache line. In brief, Elastic-Plus benefits more from larger N_{PI} over Elastic-Cache, as larger cache lines can lead to more fine-grained requests that can be issued in parallel.

7.7.2 Associativity. The associativity does not impact the parallelism of issuing requests in Elastic-Plus, but it has impact on the accesses of chunk-tags stored in the shared memory. If the associativity is changed (e.g., 16), then the storage of chunk-tags is also changed, which is stored across 4 (or 8) banks but in 8 (or 4) rows.

8 RELATED WORK

To make L1 cache of GPU more efficient, a lot of approaches have been proposed. CCWS [27] and DAWS [28] propose adaptive hardware mechanisms to capture locality lost due to contention, avoiding thrashing in L1 cache. CCA [38] not only limits the number of warps that can allocate cache lines on L1 cache, but also bypasses warps to improve the utilization of bandwidth and execution hardware. MRPB [12] develops a priority reorder buffer to put requests of one warp in a FIFO buffer and schedule requests of this buffer until the buffer is empty, since requests of the same warp have better locality. MRPB also adopts cache bypassing. When a set has no cache lines available or MSHR is full, bypassing is triggered to avoid stalls. CBWT [5] also combines bypassing and warp throttling to determine the number of active warps by detected L1 cache bypass rate, interconnection network congestion, and L1 cache contention at run time. Reference [34] coordinates static cache bypassing in which global loads are identified as caching or bypassing through profiling at compiling time and dynamic cache bypassing where the number of bypassed thread blocks is adjusted by learning from a score table. LAMAR [26] develops a hardware predictor to adaptively adjust the access granularity to L1 cache, maintaining the advantage of spatial locality and temporal locality by coarse-grained accesses and reducing over-fetch by fine-grained accesses. There are also some works addressing cache architectures. Sector cache [20] has been proposed to save tag overhead, which is also used in Reference [26]. Reference [29] decouples address tag from cache line location, in which the address tag location associated with a cache line location is dynamically determined, chosen at fetch time among several possible locations. The hit rate is increased when comparing with sector cache, but it is still constrained by coarse-grained cache block management, since cache blocks associated with an address tag have to be consecutive.

9 CONCLUSION

In GPUs, the shared memory and L1 cache are used to alleviate the penalty of long-latency memory accesses. However, these two on-chip resources cannot be efficiently utilized all the time, especially for applications with irregular memory access patterns. We have shown that the efficiency of L1 cache is very low, because there are many 32-byte requests and poor spatial locality among such requests. We also show that a notable fraction of the shared memory is unused during the lifetime of kernels. In this article, we propose Elastic-Cache architecture that cost-effectively supports both fine- and coarse-grained cache line management. We utilize the unused shared memory to store tags for fine-grained cache line management and maintain the conventional tag array for coarse-grained cache accesses. Furthermore, we propose an enhanced version of Elastic-Cache to efficiently utilize the bandwidth of L1 cache; called Elastic-Plus, it can issue multiple requests to L1 cache simultaneously. The experimental results demonstrate that Elastic-Cache improves the performance significantly and outperforms the baseline GPU by 104% on average in terms of IPC for applications with irregular memory access patterns. Elastic-Plus further improves the performance by 27% over Elastic-Cache.

REFERENCES

- [1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. 163–174.
- [2] J. Chang, S. Chen, W. Chen, S. Chiu, R. Faber, R. Ganesan, M. Grgek, V. Lukka, W. W. Mar, J. Vash, S. Rusu, and K. Zhang. 2009. A 45nm 24MB on-die L3 cache for the 8-core multi-threaded Xeon processor. In *Proceedings of the Symposium on VLSI Circuits*. 152–153.
- [3] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia. 2014. Managing DRAM latency divergence in irregular GPGPU applications. In *Proceedings of the IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. 128–139.
- [4] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'13)*. 185–195.
- [5] Xuhao Chen, Li-Wen Chang, C. I. Rodrigues, Jie Lv, Zhiying Wang, and Wen-Mei Hwu. 2014. Adaptive cache management for energy-efficient GPU computing. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*. 343–355.
- [6] Kayvon Fatahalian and Mike Houston. 2008. A closer look at GPUs. *Commun. ACM* 51, 10 (Oct. 2008), 50–57.
- [7] Mark Gebhart, Stephen W. Keckler, Bruce Khailany, Ronny Krashinsky, and William J. Dally. 2012. Unifying primary cache, scratch, and register file memories in a throughput processor. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. 96–106.
- [8] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Proceedings of the Conference on Innovative Parallel Computing (InPar'12)*. 1–10.
- [9] Mark Harris. 2013. Using shared memory in CUDA C/C++. NVIDIA Developer Blog. Retrieved from <https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/>.
- [10] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: A MapReduce framework on graphics processors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 260–269.
- [11] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. 2015. GPU register file virtualization. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 420–432.
- [12] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2014. MRPB: Memory request prioritization for massively parallel processors. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'14)*. 272–283.
- [13] N. Jing, J. Wang, F. Fan, W. Yu, L. Jiang, C. Li, and X. Liang. 2016. Cache-emulated register file: An integrated on-chip memory architecture for high performance GPGPUs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 1–12.
- [14] M. Khairy, J. Akshay, T. Aamodt, and T. G. Rogers. 2018. Exploring modern GPU memory system design challenges through accurate modeling. Computing Research Repository (CoRR), vol. abs/1810.07269.
- [15] F. Khorasani, R. Gupta, and L. N. Bhuyan. 2015. Efficient warp execution in presence of divergence with collaborative context collection. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'15)*. 204–215.

- [16] Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, and Lesley Shannon. 2012. Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. 376–388.
- [17] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWatch: Enabling energy optimizations in GPGPUs. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA'13)*. 487–498.
- [18] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic GPU cache bypassing. In *Proceedings of the International Conference on Supercomputing*. 67–77.
- [19] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2 (Mar. 2008), 39–55.
- [20] J. S. Liptay. 1968. Structural aspects of the system/360 model 85, Part II: The cache. *IBM Syst. J.* 7 (Mar. 1968), 15–21.
- [21] Keshav Pingali, Martin Burtcher, Rupesh Nasre. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'12)*. 141–151.
- [22] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA'10)*. 235–246.
- [23] Rajeev Balasubramonian, Naveen Muralimanohar, and Norman P. Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report. HP Laboratories. <https://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf>.
- [24] NVIDIA Corporation. 2014. *NVIDIA GeForce GTX 980*. <https://www.techpowerup.com/gpu-specs/docs/nvidia-gtx-980.pdf>.
- [25] NVIDIA Corporation. 2015. *NVIDIA CUDA C Programming Guide*. https://docs.nvidia.com/cuda/archive/8.0/pdf/CUDA_C_Programming_Guide.pdf.
- [26] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A locality-aware memory hierarchy for energy-efficient GPU architectures. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. 86–98.
- [27] T. G. Rogers, M. O'Connor, and T. M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'12)*. 72–83.
- [28] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-aware warp scheduling. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. 99–110.
- [29] A. Sez nec. 1994. Decoupled sectored caches: Conciliating low tag implementation cost. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA'94)*. 384–393.
- [30] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. 2013. Cache coherence for GPU architectures. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*. 578–590.
- [31] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. 2015. IBM POWER8 processor core microarchitecture. *IBM J. Res. Dev.* 59, 1 (Jan. 2015), 2:1–2:21.
- [32] Manuel Ujaldon. 2015. Inside Kepler. GPGPU workshop of Department of Computer Science, University of Cape Town. Retrieved from <http://gpu.cs.uct.ac.za/Slides/Kepler.pdf>.
- [33] Alexander V. Veidenbaum, Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, and Xiaomei Ji. 1999. Adapting cache line size to application behavior. In *Proceedings of the IEEE/ACM International Conference on Supercomputing (ICS'99)*. 145–154.
- [34] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. 2015. Coordinated static and dynamic cache bypassing for GPUs. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'15)*. 76–88.
- [35] Qiumin Xu, Hyeran Jeon, and M. Annavaram. 2014. Graph processing on GPUs: Where are the bottlenecks? In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'14)*. 140–149.
- [36] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou. 2012. Shared memory multiplexing: A novel way to improve GPGPU throughput. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 283–292.
- [37] M. K. Yoon, K. Kim, S. Lee, W. W. Ro, and M. Annavaram. 2016. Virtual thread: Maximizing thread-level parallelism beyond GPU scheduling limit. In *Proceedings of the IEEE/ACM International Symposium on Computer Architecture (ISCA'16)*. 609–621.
- [38] Z. Zheng, Z. Wang, and M. Lipasti. 2014. Adaptive cache and concurrency allocation on GPGPUs. *IEEE Comput. Architect. Lett.* PP, 99 (Sep. 2014), 1–1.

Received November 2018; revised February 2019; accepted March 2019