Tag-Split Cache for Efficient GPGPU Cache Utilization

Lingda Li Ari B. Hayes Shuaiwen Leon Song* Eddy Z. Zhang
Department of Computer Science, Rutgers University
*Pacific Northwest National Lab
lingda.li@cs.rutgers.edu arihayes@cs.rutgers.edu
Shuaiwen.Song@pnnl.gov eddy.zhengzhang@cs.rutgers.edu

ABSTRACT

Modern GPUs employ cache to improve memory system efficiency. However, large amount of cache space is underutilized due to irregular memory accesses and poor spatial locality which exhibited commonly in GPU applications. Our experiments show that using smaller cache lines could improve cache space utilization, but it also frequently suffers from significant performance loss by introducing large amount of extra cache requests. In this work, we propose a novel cache design named *tag-split cache (TSC)* that enables fine-grained cache storage to address the problem of cache space underutilization while keeping memory request number unchanged. TSC divides tag into two parts to reduce storage overhead, and it supports multiple cache line replacement in one cycle. TSC can also automatically adjust cache storage granularity to avoid performance loss for applications with good spatial locality. Our evaluation shows that TSC improves the baseline cache performance by 17.2% on average across a wide range of applications. It also outperforms other previous techniques significantly.

CCS Concepts

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

Keywords

GPGPU; Cache Organization; Spatial Locality

1. INTRODUCTION

Nowadays GPUs become a highly attractive platform for general purpose computing due to their cost effectiveness and energy efficiency. Modern GPUs are equipped with on-chip caches [1, 2, 3] to minimize the gap between throughput scaling and memory performance scaling. However, it is usually difficult to fully harness the power of on-chip caches because of the large amount of cache contention caused by massive multi-threading. The fact that many general purpose workloads exhibit irregular memory behaviors exacerbates this problem.

One severe problem for the current GPU cache is that only very small portion of intra cache line space gets utilized. Figure 1 shows the fraction of L1 cache lines that get 0, 25%, 50%, 75% and 100% of line utilization before eviction for cache sensitive Rodinia [5] and Parboil [33] benchmarks. Here we split every 128B cache line into 4 continuous equal-size chunks, and define line utilization as the portion of chunks in a fetched cache line that actually gets reused before eviction. First, we observe that many cache lines never get any portion within them reused. The major reason behind it is the severe cache contention caused by thousands of threads sharing one L1 cache simultaneously. Furthermore, among reused cache lines, only a small fraction of them are actually useful for most applications. For instance, 90.3% of reused cache lines of `mummergpu` have only 1/4 of them actually reused. Only in `histo` there are more than half of cache lines that have more than 1/2 reused.

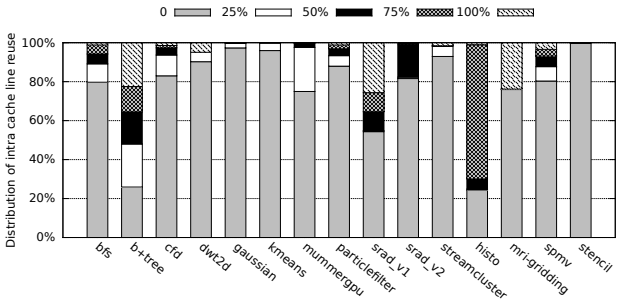


Figure 1: The distribution of reused percentage within a 128B cache line before eviction.

The observation that a significant portion within a cache line never gets used has two-fold implications. The first one is that the memory bandwidth is utilized inefficiently. Most of the data transfer to L1 caches is a waste of interconnect bandwidth and should be avoided. The second implication is that the cache space within lines is underutilized in the current cache design. If unused space within cache lines is recycled to store useful data from other cache lines, cache space utilization can be improved significantly. The severe cache contention that is common in GPU can also be relieved because the cache is able to store more cache lines, and every line stays longer in cache to have better chance to get

ICS '16, June 01-03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926253

reused. In this paper, we aim to address this cache space underutilization problem with a cache design that allows flexible cache line storage.

The locality-aware memory hierarchy (LAMAR) [27] discovers the memory bandwidth waste issue in GPU and proposes a method to address it, but it cannot address the cache space underutilization problem we find. LAMAR employs sector cache [19] and only transfers the fraction within cache line that is actually requested. But unrequested fraction still occupies cache space to prevent them from utilized by other cache lines. Some approaches have been proposed to address cache space underutilization problem in CPU caches [8, 14, 25, 30, 32, 35]. They either have very complex design and large overhead which are incompatible with the simple and efficient design of GPU, or sacrifice certain degree of storage flexibility. Our experiments also demonstrate that simply adopting small cache line size is not an efficient solution due to caused extra overhead.

In this paper, we propose *tag-split cache (TSC)*, which organizes cache space in the unit of small chunks to enable fine storage granularity for efficient cache utilization, while keeping a coarse access granularity to avoid the increment of cache request number. By default, TSC only stores requested chunks within a cache line to save space on cache misses, and thus it does not require complex mechanism used in previous techniques to predict appropriate storage granularity [14, 35]. To reduce tag storage overhead, several chunks share a fraction of tag in TSC. Besides, TSC is able to replace chunks from multiple cache lines simultaneously, while previous methods can replace at most one cache line at a time [14, 25, 30, 32, 35]. TSC also reduces interconnect network traffic by only transferring missed chunks on cache misses. Overall, the contributions of this paper are as follows:

- We discover that while using fine storage granularity can improve cache utilization, a coarse access granularity is also critical for GPU caches since its performance is sensitive to memory access number.
- Our work is the first cache design that enables fine storage granularity for GPU architecture. It not only maximizes cache space utilization but also minimizes on-chip memory interconnect traffic.
- We develop a simple adaptive scheme that dynamically switches between fine-grained storage mode and coarse-grained storage mode for TSC to avoid performance loss for applications with good spatial locality.
- Our proposed design is effective and yet has lower overhead compared with previous methods. It improves the baseline cache performance by 17.2% on average for a wide range of applications.

2. BACKGROUND AND MOTIVATION

2.1 Background

This work proposes simple yet effective architectural extensions to improve the efficiency of on-chip cache in GPU architectures. A GPU processor consists of multiple SIMD units, which are further organized into streaming multiprocessors[1] (SMs or SMXs) in NVIDIA GPUs [2, 3] or Computing Units in AMD GPUs [1]. Each SIMD unit contains 32 lanes for vector instructions, and a group of 32 threads that runs on one SIMD unit is called a warp. A GPU program follows the single instruction multiple threads (SIMT) execution model. Due to the native support for diverging scalar threads, memory addresses are determined at a per-thread granularity, which means a warp can generate up to 32 independent memory transactions. A *memory coalescing unit* is commonly implemented in GPUs to reduce the control overhead of a memory operation. It can aggregate the memory requests (ranging from 32B to 128B) from the active threads in each warp, with a minimum memory access granularity of cache line size. Based on the previous literature [6, 27], it is a common belief that the main memory of current GPUs (e.g., NVIDIA Fermi and Kepler) are optimized for such coarse-grained memory accesses, based on the fact that the main memory (e.g., GDDR5) of many commercial GPUs enable a 64B (64 bits $\times$ 8-bursts) minimum access granularity per channel.

However, there is lack of study for the appropriate granularity of GPU caches. To better exploit locality and utilize memory bandwidth, multiple types of cache are provided by GPUs, e.g., hardware-managed L1 D-caches and software-managed shared memory (software cache). Main memory (or global memory) accesses are served through L1 D-cache by default. All the SMs (SMXs) are connected by an interconnected network to a partitioned memory module, each with its own L2 data cache and main memory partition. For Kepler, the latency for L1 cache is around 20 cycles. The latency for L2 of a GPU is around 200 cycles. The latency of off-chip main memory is around 500 cycles. The gap between the latency of the L2 cache and off-chip memory is not as large as the gap between that of L1 cache and L2 cache. For this reason, in this paper, we focus on finding out the most appropriate granularity of L1 cache and proposing new cache design to enhance caching efficiency. However, our technique can be well applied to L2 cache as well.

2.2 Motivation

To improve the intra cache line space underutilization shown in Figure 1, the most straightforward method is to shrink the cache line size so that cache has a finer storage granularity.

Figure 2 shows the performance of cache sensitive benchmarks from Rodinia [5] and Parboil [33] when the cache line size is 32B, 64B, 128B and 256B for a 16KB L1 cache. Their performance is measured by instructions per cycle (IPC), and all the performance data in the figure is normalized to that when using the 128B cache line size, which is the default line size for the current NVIDIA GPUs such as Fermi and Kepler[2].

From Figure 2, we observe that different applications have different preference for cache line size. We categorize these applications into two categories: `CLP` and `CLN` (shown in Table 1), where `CLP` (Cache Line Size Positive) represents applications that benefit from larger cache line size (e.g., 128B),

[1]NVIDIA terminology will be used throughout this paper to illustrate our technique. However, the proposed idea applies to a wide range of throughput architectures.

[2]See Section 4 for the detailed configuration of the baseline architecture.

Type	Benchmark	Source
CLN	bfs	Rodinia
	cfd	Rodinia
	kmeans	Rodinia
	mummergpu	Rodinia
	particlefilter	Rodinia
	srad_v2	Rodinia
	spmv	Parboil
CLP	b+tree	Rodinia
	dwt2d	Rodinia
	gaussian	Rodinia
	srad_v1	Rodinia
	streamcluster	Rodinia
	histo	Parboil
	mri-gridding	Parboil
	stencil	Parboil

Table 1: Benchmarks categorized by performance sensitivity to the cache line size.

and `CLN` (Cache Line Size Negative) represents applications that prefer smaller cache line size.

CLN Cases.

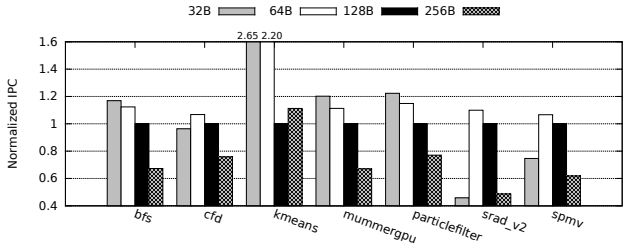
For these cases shown in Figure 2(a), we observe performance improvement by using smaller cache line sizes such as 32B and 64B over the larger ones (i.e., 128B and 256B). This is because many GPU applications show poor spatial locality as suggested in Figure 1. Using a smaller cache line in these cases releases the wasted cache space, which can be used to retain more data that will be potentially reused.

To be more specific, we use `kmeans` to showcase why `CLN` benchmarks prefer smaller cache line sizes. `kmeans` has highly divergent memory accesses within the same warp [28], which benefits most from using a smaller cache line. Figure 3 shows the kernel that is responsible for most cache misses in `kmeans`. Almost all the accesses to the "input[ ]" array miss when using the default 128B cache line. From the code, we observe that there is spatial locality across loop iterations in the same thread if there is only one thread running. However, many threads that run at the same time (within or across thread warps) access different cache lines since the value of the variable `nfeatures` is large. Therefore, there is a big chance that between two adjacent loop iterations (i.e., in between the reuse of a cache line), several other warps execute the same load instruction and fill the L1 cache with their data. As a result, most cache lines will be evicted before the spatial locality within them is exploited, resulting in a cache miss rate of 95.5% at 128B line size for `kmeans`. When we apply a smaller line size such as 32B, L1 cache can retain 4 times of cache lines compared to the default 128B configuration, and thus the probability that a cache line is evicted before spatial locality is exploited is significantly reduced. As a result, the L1 cache miss rate of `kmeans` is drastically reduced to 20.5% and the overall performance is improved by 165%.

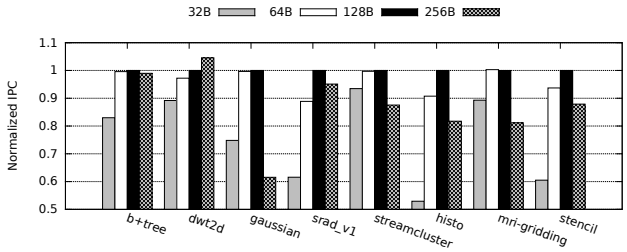
These `CLN` applications demonstrate that reducing the cache line size helps improve cache space utilization and therefore the overall performance.

CLP Cases.

Although using a smaller cache line size can improve cache space utilization, it can also cause significant performance degradation for many applications (i.e., `CLP` cases shown in



(a) `CLN` cases.



(b) `CLP` cases.

Figure 2: Performance under different L1 cache line sizes.

Figure 2(b)). There are *two potential causes* for such performance loss:

1. Using a smaller cache line size can increase the miss rate if an application already shows good spatial locality. For these applications, larger cache lines are preferred.

2. Since the memory coalescing is done using the granularity of cache line size, using smaller cache lines results in more cache requests. More cache requests can have the following negative impact on performance: (a) under the same volume of misses, smaller cache line size means more miss requests are sent to the lower level memory (i.e., L2 cache), resulting in more interconnect traffic and longer miss latency; (b) cache resources (e.g., MSHRs, miss queue entries) are saturated more easily, resulting in less miss memory instructions can be processed in parallel; and (c) more cache requests also mean longer hit latency.

```
__global__ void invert_mapping(float * input, float * output,
                               int npoints,  int nfeatures)
{
  int point_id = threadIdx.x + blockDim.x*blockIdx.x;
  int i;

  if ( point_id < points )
    for ( i=0; i < nfeatures; i++ )
      output[point_id + npoints*i] = input[point_id*nfeatures + i];
  }
  return;
}
```

Figure 3: The kernel code of `kmeans`.

To investigate the impact of the first potential cause (i.e., more misses caused by using smaller cache lines), we evaluate and compare the L1 miss rates under different line sizes for both `CLN` and `CLP` applications, shown in Figure 4. Since

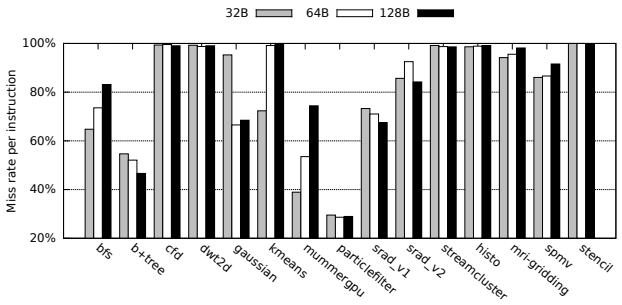


Figure 4: L1 cache miss rate under different line sizes.

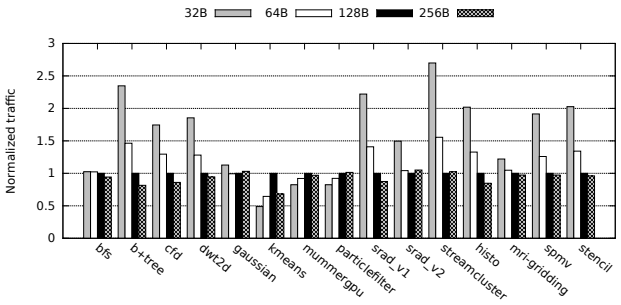


Figure 5: L1 to L2 traffic under different cache line sizes.

the cache request number varies under different cache line sizes, to make a fair comparison, miss rate is calculated by dividing the total number of memory instructions that incur misses by the total number of memory instructions, which is a constant.

We observe that the miss rate using a 32B cache line is less than or similar to that using a 128B line for the majority of cases except `gaussian`. It is obvious that `CLN` cases using a 32B line incur less misses due to the better cache space utilization, however, this is also true for some `CLP` cases. For instance, although `mri-gridding` suffers from performance degradation when using a 32B cache line as shown in Figure 2(b), it still shows less L1 misses. Also, `stencil` has a nearly 100% miss rate under any of the three cache line sizes, but its performance of using 32B cache lines is significantly lower than that using 128B lines. Therefore, the argument that a smaller cache line increases miss rate for `CLP` cases is not consistent among all the benchmarks, thus it is not the major reason for performance loss.

We then further investigate if the interconnect and resource contention caused by more cache requests (i.e., the second potential cause) is the primary reason for performance loss in `CLP` cases using smaller cache lines. Figure 5 shows the interconnect network traffic from L1 cache to L2 cache, which is a direct indicator for the number of L1 miss requests. The results show that a smaller cache line size does increase L1 to L2 traffic significantly for many cases, e.g., as much as 170% for `streamcluster`. For `stencil`, the L1 to L2 traffic increases by 103% when using 32B cache lines, which explains the reason of its degraded performance although its miss rate does not change (Figure 4).

Discussion.

In summary, we draw two conclusions from our observations. On one hand, it is important to enable **fine-grained data storage** in cache (e.g., 32B cache line size) in order to achieve better cache space utilization, as suggested by `CLN` applications. On the other hand, the cache should still have a **coarse access granularity** (i.e., coalescing granularity) to avoid the performance loss caused by more cache requests in `CLP` cases. Therefore, simply applying a smaller or bigger cache line size is not a good solution.

We argue that 128B is the most reasonable **access granularity** for GPU L1 cache for the following reasons. In the most common cases of memory access, each thread requests an integer or single-precision float variable, and both of them are 4B. As a result, a thread warp which contains 32 threads will request 128B data in total, and thus a 128B access granularity matches it perfectly. In addition, access granularity that is larger than 128B is not able to further reduce the cache request number. As shown in Figure 5, using 256B cache lines has almost the same volume of L1 to L2 traffic as that using 128B cache lines in most cases. For some cases, it incurs more traffic because there are more L1 misses with 256B cache line size. Therefore, we choose 128B as the access granularity of our design, and requests from load/store unit are coalesced as if the cache line size is 128B. We will discuss the implementation of **fine-grained data storage** in details next.

3. IMPLEMENTATION

To support coarse access granularity and fine storage granularity with low overhead, we design *tag-split cache (TSC)*.

3.1 Structure

As discussed above, 128B is the most appropriate access granularity for TSC. In other words, a cache line still virtually represents a contiguous 128B memory region, although our design may not store a cache line in the way of a normal cache using 128B cache lines. To enable fine-grained storage, we organize the data RAM of tag-split cache in the unit of data *chunks*. The chunk size is smaller than the cache line size 128B, and a cache line can have 1 to (line size/chunk size) chunks stored in the tag-split cache. In the rest of this paper, we assume the chunk size (i.e., the storage granularity) is 32B by default. We will study the impact of chunk size in Section 5.4.

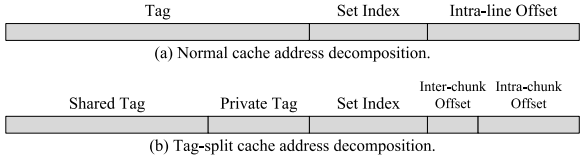


Figure 6: Address decomposition of normal cache and tag-split cache.

Figure 6 compares the address decomposition of normal cache and tag-split cache on cache access. To support the 128B access granularity, we need to make sure all chunks within a 128B cache line fall into the same cache set. To achieve this goal, tag-split cache uses exactly the same segment in address as the normal cache for cache set index.

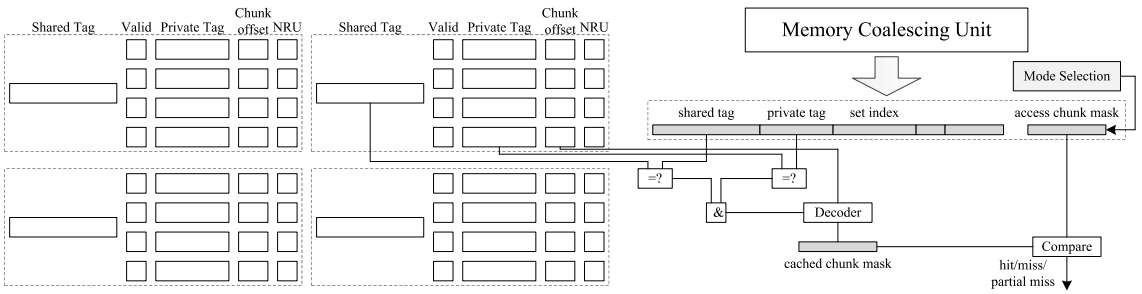


Figure 7: The cache set structure (left) and tag comparison logic (right) of tag-split cache.

Thus the set lookup process on cache accesses is not changed. The remaining 2 bits between set index and intra-chunk offset are called *inter-chunk offset*, which represent the relative position of a chunk within its corresponding 128B cache line. Note that our design is also applicable for hash based set index used in recent GPUs [22], and we use it in our experiments.

Theoretically, every chunk needs a tag entry since they can store chunks from different cache lines. However, if every chunk is assigned with a separate tag entry, there will be a large storage overhead, and more tag comparison on cache accesses will also increase cache access latency. To reduce overhead, we divide traditional tag segment within memory address into two parts: higher bits are called *shared tag* and lower bits are called *private tag*. A set of chunks have the same shared tag and each chunk has its own private tag. The motivation for this approach is that programs tend to access a small memory region in a short amount of time, and thus different cache lines are likely to share higher bits in their tags. We study the impact of private tag length to the performance of tag-split cache in Section 5.1.

Figure 7 shows the structure of a cache set in tag-split cache. In our default configuration with 32 cache sets, a cache set includes 16 chunks. They are divided into 4 *chunk groups*, and all 4 chunks in one group have the same shared tag. In doing this, a chunk group can store chunks from multiple 128B cache lines, and thus **fine storage granularity** is supported to increase the effective cache capacity.

The purpose of having a chunk group of 128B is, in the case that applications show good spatial locality and fine-grained storage is harmful, tag-split cache can fall back to a normal cache. In such cases, a chunk group can always be used to store a whole cache line. Besides private tag, each chunk also contains a valid bit, inter-chunk offset bits, and an NRU bit for replacement.

Another thing worth noticing about tag-split cache is that a cache line can have its chunks spread across multiple chunk groups. Thus, chunks of a cache line can be stored anywhere among 16 chunks belonging to the corresponding cache set. To be able to access any subset of chunks in a cache set, we use 16 SRAMs to compose a 16-bank data RAM. Different banks can be accessed simultaneously and each bank holds one and only one chunk for every cache set, making it feasible to read any subset of chunks belonging to one cache set simultaneously.

3.2 Cache Access Processing

On a load/store instruction, the streaming multiprocessor first coalesces data requests of all thread in a warp according to the access granularity of 128B as usual. It also calculates which chunks within the requested 128B cache line are actually required to generate a 4-bit *access chunk mask*. Then the cache line request along with its corresponding access chunk mask is passed to the tag-split cache.

After TSC receives a request and reads out all tag information of the corresponding cache set, it starts to perform tag comparison. As shown in Figure 7, shared tags and private tags are compared in parallel. Note that shared tag comparison is performed for every chunk group and private tag comparison is performed for every chunk. If both the shared and private tags of a valid chunk match those of the current request, its chunk offset is used to set the corresponding bit in the *cached chunk mask*. Then, the cached chunk mask is compared with the access chunk mask coming with the request to determine the access status.

If all required chunks are already cached, the current requests hits in the tag-split cache. The location of cached chunks are used to index the data RAM to get desired data. Then TSC combines those returned data to form a virtual cache line before returning to SM.

If there is no required chunk found, a cache miss occurs. On a miss, TSC allocates an new MSHR entry or uses an existing MSHR entry to record this miss as will be discussed in Section 3.4. MSHR will send the miss request to the L2 cache when feasible. Then it allocates cache space for missed chunks, which will be described in Section 3.3.

A special access status for TSC is *partial miss*, which occurs when a part of required chunks are not present in the cache. In such scenarios, the cached chunks are addressed as a cache hit and missed chunks are addressed as a cache miss. While missed chunks are requested from L2 cache, cached chunks are read from data RAM.

3.3 Allocation and Replacement

In order to allocate cache space for missed chunks on misses and partial misses, at the first step, we check if there are available invalid chunks. In our design not all invalid chunks can be used. An invalid chunk is available for the current miss only if its corresponding chunk group does not have any valid chunk, or its shared tag matches with the current miss. Otherwise, if there is valid chunk in the target group which has at least one invalid chunk and the shared tag of the target group is different from that of the missed chunk, allocating those invalid chunks for the current miss will have to invalidate all valid chunks in the target chunk group since the shared tag is changed.

If there are not enough invalid chunks found on a miss, one or more valid chunks need to be replaced. We use Not

Recently Used (NRU) algorithm [24] to select victim chunks since it has low storage requirement and yet is efficient in practice [10, 17]. Each chunk is associated with one NRU bit, and all NRU bits are initialized to 0s. On an access to a chunk, its NRU bit is set to 1 to indicate it has been accessed. If all NRU bits in a cache set are 1s, they are reset to 0s as an aging mechanism. On replacement, we preferentially select chunks whose NRU bits are 0s. If there are more chunks with 0 NRU bits than required, we select randomly among them. Otherwise if there are not enough chunks with 0 NRU bits, chunks with 1 NRU bits are selected randomly.

After the victim chunks are selected, they can be simply invalidated simultaneously by reseting their valid bits. The reason why we can do this is GPU L1 cache uses a write-evicted policy and thus no data writeback is required. Therefore the replacement of tag-split cache is very easy to implement. GPU L1 caches adopt a write-evicted policy because of its simple cache coherence implementation. Snoop and directory based implementation of cache coherence is well known for their poor scalability with the number of processors [20], while GPUs typically have dozens of streaming multiprocessors for high throughput. Current GPUs choose to directly write data to L2 caches and evict cached written blocks in L1 to make sure other SMs can get updated data from shared L2 caches.

On the other hand, previous methods that change cache storage granularity perform replacement in the unit of cache line since only one cache line can be written back at a time [32, 25, 14]. As a result, it is possible that after one replacement there is still not enough cache space available and thus multiple-round replacement is desired. Multiple-round replacement not only increases access latency which adversely affects performance, but also increases design complexity and verification difficulty. TSC always finishes its replacement in one round and thus does not suffer from these problems.

3.4 MSHR

Miss Status Holding Register (MSHR) is responsible for recording cache misses and sending miss requests to lower memory. To be compatible with our tag-split cache and reduce interconnect traffic, we extend the conventional MSHR with an *issued chunk mask* for each entry. An MSHR entry still keeps track of all miss requests to a 128B cache line as usual. The issued chunk mask records which chunks within that cache line have been requested from L2 cache to avoid redundant requests.

When a new MSHR entry is allocated on a miss, i.e., the current miss is the first miss to a 128B cache line, a miss request including the current missed chunk mask is sent to L2 cache, and L2 cache will only return those missed chunks to reduce transfered data. Besides, the issued chunk mask of the MSHR entry is initialized to the current missed chunk mask. When a miss request is added to an existing MSHR entry, i.e., there are on-going misses to the same 128 cache line, we compare its missed chunk mask with the issued chunk mask to avoid redundant data transfer. Only chunks whose requests have not been sent before are requested from L2 cache. If the requests of all current missed chunks have been sent before, no new request will be generated by MSHR.

3.5 Adaptive Storage Granularity Modes

Some GPU applications have good spatial locality such as `histo` in Figure 1. Merely storing current requested chunks on misses for these applications will destroy spatial locality and result in additional misses to adversely affect performance. To solve this problem, in addition to the default *fine-grained storage mode* where only current desired chunks are stored in the tag-split cache, we employ another mode called *coarse-grained storage mode*. In coarse-grained mode, tag-split cache will fetch and store all chunks within a 128B cache line on miss to exploit spatial locality. Switching from fine-grained mode to coarse-grained mode is implemented by simply marking all bits in the access chunk mask which is generated by memory coalescing logic to 1s, as shown in the upper right corner of Figure 7.

Now the question is how to decide which mode should be adopted. We design a simple mechanism to dynamically make the mode selection decision at runtime. Among all cache sets in a L1 cache, we randomly select two subsets of cache sets. One subset is dedicated to fine-grained mode and the other is dedicated to coarse-grained mode, and these cache sets are called *sampler sets* since they always use the assigned mode. Then the mode selection decision is made by monitoring the access behavior of sampler sets, which is similar to the *Set Dueling* method [26]. Set Dueling is developed to choose the better performing one between 2 candidate replacement policies for CPU caches by comparing their miss number.

Unlike Set Dueling, which compares the miss number of sampler sets to judge their performance, we consider both miss number and generated traffic of two competitive modes in order to decide which one is better. The motivation behind it is our mode selection decision can directly affect interconnect traffic, and previous work has demonstrated that interconnect network traffic has a large impact on GPU cache performance [27, 6]. If one mode is learned to have better performance than the other, the remaining sets called *follower sets* will all adopt the winner mode.

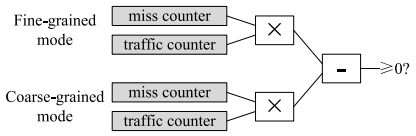


Figure 8: Mode selection logic.

Figure 8 shows the logic for mode selection. For the sampler sets of every mode, there are two counters to record their miss number and the corresponding traffic size generated on misses. On a miss request in the sampler set, the miss counter of the corresponding mode is increased by 1, and the corresponding traffic counter is also increased by estimated traffic generated to serve this miss request. We estimate traffic on cache misses as follows. The interconnect network between L1 and L2 caches are packet based network. On a miss, L1 cache sends one request packet to L2 cache at first, and then L2 cache returns n data packets to L1. Since the packet size is equal to the default chunk size of our design (32B), n is equal to the number of requested chunks. Therefore, the total packet number generated for a miss is $1 + n$, where n is the missed chunk number.

With these two counters, we use the product of them to represent the performance overhead of the correspond-

ing mode. The smaller the product is, the less performance loss we consider the corresponding mode incurs. Finally, the mode which has the smaller product is selected as the operating mode of all follower sets.

To adapt to memory access behavior changing, we employ an aging mechanism for these counters. Once the miss counter of any mode exceeds a threshold, we shift all 4 counters to the right by 1 bit. In this way, recent memory access behavior can be reflected by the values of these counters more quickly. The experiments show that a threshold of 1024 works well for our design.

Although using a larger number of sampler sets can make the mode selection more accurate, it increases the performance overhead since sampler sets cannot change their operating mode. To reduce learning overhead for mode selection, all sampler sets are selected from the L1 cache of a certain SM, which is SM_0 in our experiments. L1 caches of other SMs follow the mode selection decision made by SM_0. We also find that selecting 8 out of 32 total cache sets as sampler sets is enough to get satisfying mode selection result. Among them, 4 sets are dedicated to fine-grained mode and the other 4 sets are dedicated to coarse-grained mode.

4. EXPERIMENTAL METHODOLOGY

We model and evaluate our design using a widely-used cycle-accurate GPU simulator GPGPU-Sim 3.2.2 [4]. The microarchitecture configuration of the simulator is shown in Table 2. We models a typical GPU architecture. The configuration of our design is illustrated in the bottom of Table 2. GPU power consumption is estimated through GPUWattch [15].

SIMT Core (SM)	15 SMs, SIMT width=32, 5-stage pipeline, 1.4GHz, 2 GTO schedulers [28]
SM Limit	32768 registers, 1536 threads, 48 warps, 48KB shared memory
L1 DCache	16KB/SM, 128B line, 4-way, 32 MSHRs, hash set index [22]
L2 Cache	12 banks, 64KB/bank, 128B line, 8-way, 32 MSHRs
Interconnect	32B width, 700MHz
Memory Controller	FR-FCFS, 924MHz, 6 channels
DRAM	tCL=12, tRP=12, tRC=40, tRAS=28, tRCD=12, tRRD=6
Tag-Split Cache	32B chunk size, 4 chunk groups per set, 4 chunks per group, 1-cycle extra hit latency, 12-bit shared tag, 8-bit private tag

Table 2: Configuration of the baseline architecture and tag-split cache.

Our design is evaluated with Rodinia 3.0 [5] and Parboil [33] benchmarks. Out of all 22 Rodinia benchmarks, we select 11 benchmarks. We also select 4 benchmarks from Parboil. The remaining benchmarks are not cache sensitive and their performance does not exhibit noticeable improvement (less than 3%) even under the perfect cache which only incurs compulsory misses. All the benchmarks are compiled with nvcc 4.0. Most of the benchmarks run to completion on the simulator. For a few of them that have long simulation time, we execute them long enough so that the variation of IPC is very small (1 billion instructions).

5. EVALUATION

5.1 Impact of Private Tag Length

One major design choice for tag-split cache is the length of private tag field. In a common configuration of 32-bit memory address (i.e., 4GB GPU memory), the total length of shared and private tags is 20 bits. Larger private tag and smaller shared tag mean more 128B cache lines are potentially able to share the same chunk group and thus better cache space utilization, while smaller private tag means less storage overhead since each chunk has a private tag. The selection of private tag length is essentially a tradeoff between performance and overhead.

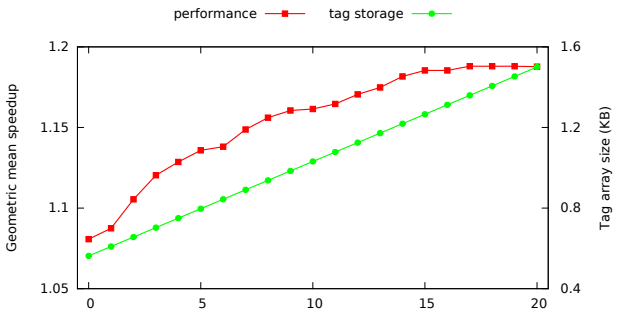
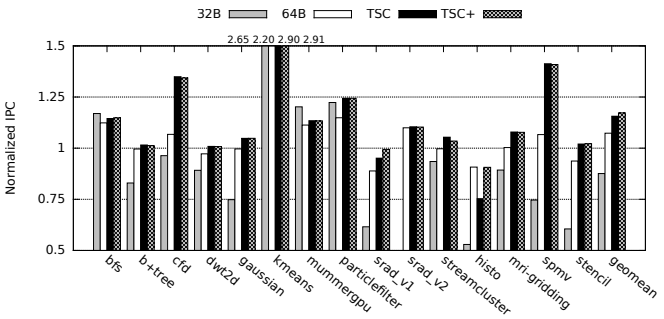


Figure 9: The performance and tag storage of tag-split cache for different private tag length.

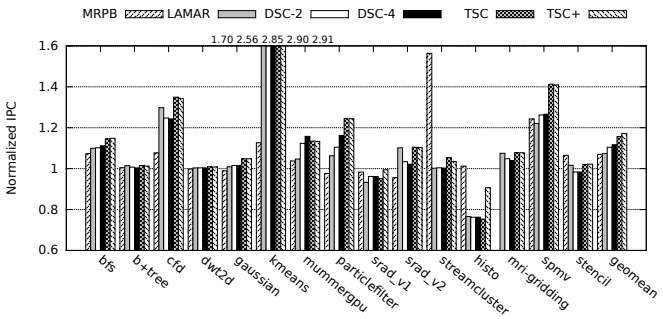
Figure 9 shows the variation of performance and tag storage when the private tag length changes from 0 to 20. Y axis represents the geometric mean speedup of all 15 benchmarks in terms of IPC compared to the baseline cache using 128B lines. Here the evaluated version of tag-split cache does not include mode switch and always adopts fine-grained mode. When the private tag length is 0, tag-split cache is equivalent to sector cache and there is no cache space saving. When it is 20, chunks within a chunk group no longer share any fraction of tag. We observe that with the increment of private tag length, the performance improvement becomes slower. There is a performance leap from 6-bit to 7-bit since `streamcluster` starts to benefit from tag-split cache at that point. The results demonstrates the assumption behind our tag split design, that most programs tend to access a limited size of memory region in a short time. We adopt 8-bit private tag for tag-split cache in the rest of our evaluation since it is a good tradeoff point between performance and overhead. 8-bit private tag means a chunk group can store any chunks within 1MB memory region.

5.2 Performance

In this section, we evaluate the performance of two versions of tag-split cache, including: (1) basic design always enabling fine-grained storage, i.e., no adaptive mode switch (TSC), and (2) TSC with mode switch for good spatial locality applications (TSC+). Besides caches using various fixed cache line sizes (32B and 64B), we also compare tag-split cache with three related schemes: LAMAR [27] (state-of-the-art technique for GPU memory bandwidth utilization optimization), decoupled sectored cache (DSC) [32] (classical cache space utilization optimization scheme for CPU caches with relative low overhead), and MRPB [11] (state-of-the-art GPU memory request reordering approach to re-



(a) Comparison with fixed cache line size.



(b) Comparison with other work.

Figure 10: The performance of various schemes. All data are normalized to that of the baseline cache using 128B lines.

duce intra- and inter-warp contention). DSC-2 and DSC-4 represent the DSC configuration of using 2 and 4 times of tag entries respectively compared with a sector cache [19]. Note that we also adopt our modified MSHR design for all DSC configurations so that they can also reduce interconnect traffic as our design.

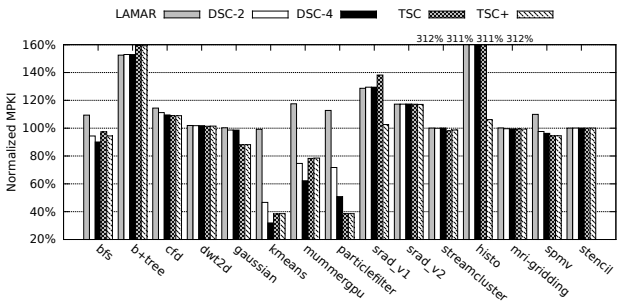


Figure 11: Normalized MPKI of L1 cache for various schemes.

Figure 10 shows the performance speedup of various techniques over the baseline cache with 128B cache line size. At first, Figure 10(a) compares the performance of TSC with normal caches using small line sizes. TSC outperforms small cache line size for almost every benchmark since it does not suffer from the performance overhead caused by cache request increment when using fine access granularity.

TSC also outperforms the baseline with 128B cache lines except `srad_v1` and `histo`. These two benchmarks have good spatial locality and TSC+ can largely counteract the performance loss of TSC by switching to coarse-grained mode. For `srad_v1`, the performance of TSC+ is almost the same as that of the baseline. The reason why there is still performance degradation for TSC+ in `histo` is the training overhead of sampler sets, which always use fine-grained mode. These sets suffers from significantly more misses. Although TSC+ correctly learns that `histo` prefers coarse-grained mode and uses this mode for almost all accesses, it has a limited number of total thread blocks, so that other SMs cannot amortize the performance loss of SM_0 that owns sampler sets by executing more thread blocks. As a result, the slowest SM_0 dominates the overall performance. Except `histo`, all other benchmarks have plenty of thread blocks so that non-sampler faster SMs are assigned with more thread blocks at runtime to relieve workloads from the sampler SM. Therefore the performance difference between TSC and TSC+ is negligible for other benchmarks as shown in Figure 10(a).

Figure 10(b) compares TSC's performance with those of other techniques. Overall, TSC and TSC+ improve the performance of the baseline cache by a geometric mean of 15.6% and 17.2% respectively across all evaluated applications, while MRPB, LAMAR, DSC-2 and DSC-4 achieve an average performance improvement over the baseline by 7.0%[3], 7.4%, 10.4% and 11.7% respectively. Unlike our technique, MRPB improves performance by rescheduling memory requests from different warps to improve temporal locality. The fact that MRPB has better performance for some benchmarks like `streamcluster` implies that our technique can be applied together with MRPB to further improve performance. We leave such exploration as future work.

To have further insight on why TSC can outperform LAMAR and decoupled sectored cache, Figure 11 shows the normalized Miss Per Kilo Instructions (MPKI) of L1 cache for various techniques. First, we observe that LAMAR always has the same or more misses compared with the baseline. It is because LAMAR uses sector cache and cannot harness wasted cache space within cache lines, while DSC and our TSC can both make use of these wasted space.

Compared to decoupled sectored cache (DSC), tag-split cache has three major advantages. First, DSC enforces a restrictive way in multiple cache lines sharing the same cache space in order to avoid using data pointer. For instance, assume there is a free 128B space in cache, if two 128B cache lines both want their first 32B to be stored, DSC cannot store these two lines simultaneously because they will occupy the same 32B place. TSC does not have such restriction and can store both cache lines at the same time. Second, TSC is able to always finish replacement in one cycle while DSC may need multi-cycle replacement. That is why for `bfs` and `kmeans` DSC-4 incurs less misses than TSC but TSC still outperforms DSC-4. Third, TSC has less storage overhead since DSC uses a separate tag entry for each 32B data as will be shown in Section 5.6. The advantage of using separate

[3]MRPB cannot work correctly for `mri-gridding` in our experiments, and its geometric mean speedup is calculated for the rest 14 benchmarks.

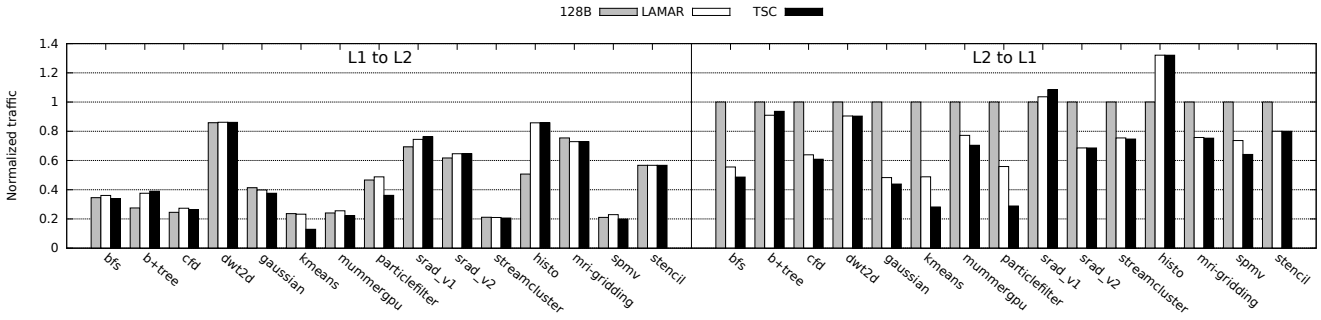


Figure 12: The normalized interconnect traffic between L1 and L2 for various schemes.

tags is that DSC suffers from less misses compared with TSC in some benchmarks, because some chunks cannot be utilized in TSC if there are not enough cache lines sharing the same shared tag.

For `b+tree`, `cfd`, `srad_v1`, `srad_v2` and `histo`, the MPKI of TSC is higher than that of the baseline cache. Among these benchmarks, TSC+ learns that coarse-grained mode works best for `srad_v1` and `histo` and reduces misses significantly by switching to it. For the remaining 3 benchmarks, although TSC incurs more misses using fine-grained storage mode, it produces less interconnect traffic as will be shown in Figure 12. Taking both of them into consideration, TSC+ decides to stay in fine-grained mode rather than switch to coarse-grained mode for these benchmarks.

From this point on, we mainly show the results of TSC since most benchmarks prefer fine-grained storage mode and adaptive mode switch is not necessary for them.

5.3 Interconnect Network Traffic

Figure 12 shows the interconnect network traffic between L1 and L2 caches for TSC and LAMAR. All results are normalized to the "L2 to L1" traffic of the baseline cache. There are two types of traffic between L1 and L2: "L1 to L2" includes load miss requests and write data from L1 caches to L2 caches, and "L2 to L1" includes loaded data from L2 caches to L1 caches. "L2 to L1" is the main contributor of total traffic since most GPU accesses are load accesses and loaded data are larger than requests.

Figure 12 shows that for most applications TSC can reduce more memory traffic than LAMAR for both "L1 to L2" and "L2 to L1", up to 45.4% and 71.8% respectively (`kmeans`). The major reason behind this is that TSC encounters fewer L1 cache misses than LAMAR, as shown in Figure 11, and thus miss requests sent to L2 and returned data from L2 are both reduced. We also observe significant "L2 to L1" traffic reduction from LAMAR for some cases since it also shrinks the returned data size on misses. However, LAMAR cannot make use of wasted cache space to reduce L1 misses. As a result, its "L1 to L2" traffic increases compared with the baseline. These results show that besides cache space utilization improvement, tag-split cache is also efficient for interconnect traffic reduction.

5.4 Basic Storage Granularity Selection

One of the most important design choice of TSC is its storage granularity, i.e., the chunk size. Figure 13 shows the performance of TSC using 8-bit private tag with various

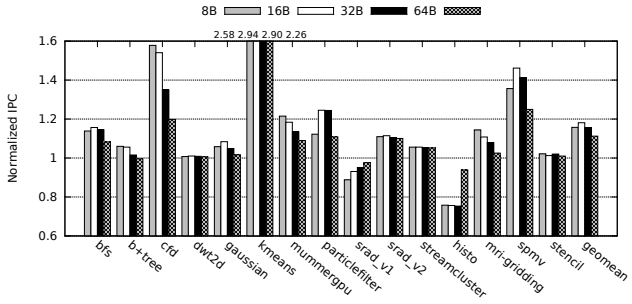


Figure 13: TSC performance with various basic storage granularity.

chunk sizes. Our experimental results show that using 8B, 16B, 32B and 64B chunks can achieve an average performance improvement of 15.7%, 18.1%, 15.6% and 11.7% respectively compared to the baseline. On one hand, smaller chunk size means better cache space utilization and thus performance for benchmarks such as `cfd` and `mri-gridding`. On the other hand, using smaller chunks can further destroy spatial locality for benchmarks like `kmeans` and `srad_v1`. Also, every time the chunk size is reduced by half, the private tag storage is doubled. Taking both performance and overhead into consideration, we select 32B as the chunk size in our experiments. Another reason for selecting 32B is the minimal data transfer unit on the interconnect network between L1 and L2 caches are 32B. Hence a smaller chunk size may not reduce interconnect traffic further.

5.5 The Impact of Access Latency

Because TSC has slightly more complicated cache line lookup process than the baseline, it can increase the hit latency of L1 cache. All our experiments above assume that it increases the L1 cache hit latency by 1 cycle, and we evaluate the performance of TSC under various additional L1 cache hit latency in this subsection. Our experimental results show that the performance impact of increasing hit latency is negligible for TSC. For instance, the maximal performance degradation with an additional hit latency of 3 cycles is less than 1% among all benchmarks for TSC. This is because GPU can effectively hide cache hit latency thanks to its massive multi-threading capability and pipelining. Therefore, we conclude that the performance impact due to the hit latency increment of TSC is negligible.

	Overhead	Speedup		Overhead	Speedup
LAMAR	0.063KB	7.4%	MRPB	1.21KB	7.0%
DSC-2	0.45KB	10.4%	TSC	0.58KB	15.6%
DSC-4	1.33KB	11.7%	TSC+	0.58KB	17.2%

Table 3: Storage overhead of various schemes.

Since recent GPUs have been proved to have large L1 cache latency [21]. We assess the L1 cache latency of various GPU architectures using micro-benchmarks. Using the assessed results, we evaluate the performance of TSC when the default L1 cache latency is 20-cycle (Fermi and Kepler) and 90-cycle (Maxwell) respectively. We assume TSC further increases the latency by 2 cycles in both cases. Under 20-cycle and 90-cycle L1 cache latency, TSC achieves a geometric mean speedup of 15.6% and 15.4% respectively across all benchmarks compared with the default cache. The performance benefit of TSC is similar to that obtained under 1-cycle L1 cache latency (15.6%), which is used by default in our experiments. Therefore, we conclude that TSC also works well under long L1 cache latency.

5.6 Overhead

In this subsection, we discuss various overhead of TSC. Table 3 compares the storage overhead of various schemes for a 16KB L1 cache. The major storage overhead of TSC comes from private tag, valid bit, chunk offset and NRU bit that are private for each chunk. Totally, the overhead is 0.58KB per 16KB L1 cache for 8-bit private tag configuration. TSC+ only requires 4 16-bit counters for the entire GPU to record the performance of two operation modes in addition to the overhead of TSC. TSC has better performance and less overhead at the same time compared with other techniques except DSC-2. Compared with DSC-2, TSC with 5-bit private tag has less storage overhead of 0.44KB, and it still has a higher geometrics performance improvement of 13.6%.

We use CACTI 6.5 [34] is to evaluate the area overhead under the 45nm technology. The result shows that the area overhead is 0.002 mm^2 per L1 cache. The overall extra area for TSC is evaluated to be 0.03 mm^2, which occupies roughly 0.01% of the total area of a typical GPU under 40nm technology [23].

We also used CACTI to evaluate the timing and power overhead of TSC. The data lookup of TSC adds less than 0.02 ns, and thus 1-cycle additional hit latency increment modeled in our experiments is more than enough. CACTI also reports that tag comparison of TSC consumes 0.0008 nJ of extra energy on each data access. It is approximately 0.4% of total cache access energy consumption. The leakage power increment due to extra tag storage is 0.5 mW per L1 cache. In total, the leakage power increases by 0.02% for the whole GPU.

6. RELATED WORK

CPU intra cache line space utilization: Prior work has tried to address the cache space underutilization for CPU caches by altering cache organization. Amoeba-Cache [14] uses an unified storage for tag and data, and it predicts the best access granularity on misses. Amoeba-Cache employs a complex structure for data lookup since tag can be stored anyway in the unified storage. Compared with our design, Amoeba-Cache incurs much higher design overhead and thus is not suitable for GPU cache. Amoeba-Cache and adaptive cache line size [35] also employ complex mechanism to predict the most appropriate cache line size. Line distillation [25] partitions cache into two parts. On a cache miss the whole line is placed into the first place. When a cache line is evicted from the first part, only reused words in that line can enter the second part. Their method cannot resolve the wasted space in the first part of cache. Inoue *et al.* propose a variable line-size cache design for merged DRAM/logic integrated circuit [9]. Their design incurs large tag storage overhead since every subline has its own tag entry. Decoupled sectored cache [32] proposes to add more tag entries for a sector in sector cache [19], so that different cache lines can share space within a sector to improve space utilization. As we discussed in Section 5.2, decoupled sectored cache restricts the space sharing way so that our design outperforms it significantly. Sector pool cache [30] also extends sector cache to make use of spare subsectors. Like decoupled sectored cache, sector pool cache also restricts the way that different cache lines share subsectors and thus is less efficient than TSC. Besides, it uses pointers to indicate the data location. Since pointer chasing can be expensive for hardware, TSC does not use pointers and is less complex.

Compared with previous CPU cache techniques, our technique is different in the following aspects. Firstly, CPU cache work does not consider the impact of cache line size on cache request number, which can significantly affect GPU performance as shown in Figure 2 and 5. TSC can reduce the cache request number by using a coarse access granularity. Secondly, as shown in Figure 4, since most GPU benchmarks have poor spatial locality, the major purpose of TSC is to utilize the wasted cache space caused by poor spatial locality in order to retain more data for exploiting temporal locality, rather than adapting to the spatial locality variation. Thirdly, previous methods usually employ complex cache structure and locality prediction mechanism to achieve fine-grained storage. For instance, Amoeba Cache uses a unified storage for tag and data, which makes the process of cache lookup complicated. TSC incurs significant lower complexity and overhead compared to them. Fourthly, we propose shared/private tag to reduce the tag overhead. Finally, TSC can invalidate/replace multiple cache lines in one cycle thanks to the write-invalid L1 cache in GPU, while previous CPU cache work has to implement multi-round replacement, which increases design complexity and verification efforts.

Memory bandwidth optimization on CPUs: Some recent work has explored changing data access granularity for better memory bandwidth utilization in CMP environment. AGMS [38] relies on programmers to specify the best granularity of data access, while DGMS [39] dynamically adjusts access granularity based on spatial locality at runtime. MAGE [18] integrates an adaptive granularity memory system with error checking for both performance and resiliency. However, these work cannot address the problem of cache space underutilization due to limited spatial locality.

Memory bandwidth optimization on GPUs: The most related work to TSC is locality-aware memory hierarchy (LAMAR) proposed by Rhu *et al.* [27]. They also observe that spatial locality for GPU cache is low for many irregular applications, and they propose to use sector cache (supported by modified sub-ranked memory) and only fetch the demanded parts if the requested cache line is predicted to have poor spatial locality to reduce memory traffic. The

major difference between LAMAR and our work is that we not only improve the memory bandwidth utilization, but also address the cache space underutilization problem that results from poor spatial locality, and thus our method significantly outperforms LAMAR as shown in Section 5.2.

Other related GPU cache work: Throttling the maximum number of current running thread warps can reduce cache contention and thus improve performance. CCWS [28] uses auxiliary tag array in L1 cache to detect locality lost and adjusts maximal concurrent warp number accordingly. Instead of reactive cache contention detection, DAWS [29] proactively predicts the optimal concurrent warp number based on history information to improve performance further. Kayiran *et al.* limit the concurrent warp number at thread block level [13]. MRPB [11] reorders on-the-fly memory requests so that requests from a small group of warps make use of cache resource preferentially. It also employs a reactive bypassing scheme. Some work focuses on improving GPU cache performance through novel cache replacement methods [6, 7, 12, 31, 36, 37]. A decoupled GPU L1 cache is proposed in [16] to enable dynamic locality filtering functionality in the extended tag store for efficient and accurate runtime cache bypassing. Since the main goal of our design is to address intra cache line space underutilization due to lack of spatial locality, our work is complementary with all these request scheduling and replacement techniques that aim to improve the temporal locality of data accesses. They can be used together with TSC for further performance improvement.

7. CONCLUSION

With their high computation throughput, GPUs become a popular platform for general purpose applications. These applications usually show irregular memory access behavior, which causes the underutilization of cache space and memory bandwidth. This work addresses these problems by introducing a new cache architecture called *tag-split cache (TSC)*. TSC efficiently utilizes the cache space by enabling fine-grained cache storage. TSC also dynamically adjust data storage granularity to avoid performance degradation for applications with good spatial locality. Based on our experimental results, TSC outperforms previous techniques significantly while having low overhead, including MRPB, LAMAR and decoupled sectored cache.

8. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their helpful comments. This work is supported by NSF Grant NSF-CCF-142150, Google Faculty Award, Rutgers University Research Council Grant, and DARPA PERFECT Project. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

9. REFERENCES

[1] AMD Graphics Cores Next (GCN) Architecture White paper, 2012.

[2] NVIDIA Kepler GK110 white paper. 2012.

[3] NVIDIA's next generation CUDA compute architecture: Fermi. 2009.

[4] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163–174, April 2009.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[6] X. Chen, L.-W. Chang, C. I. Rodrigues, J. Lv, Z. Wang, and W.-M. Hwu. Adaptive cache management for energy-efficient gpu computing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 343–355, Washington, DC, USA, 2014. IEEE Computer Society.

[7] J. Gaur, R. Srinivasan, S. Subramoney, and M. Chaudhuri. Efficient management of last-level caches in graphics processors for 3d scene rendering workloads. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 395–407, New York, NY, USA, 2013. ACM.

[8] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 338–347, New York, NY, USA, 1995. ACM.

[9] K. Inoue, K. Kai, and K. Murakami. Dynamically variable line-size cache exploiting high on-chip memory bandwidth of merged dram/logic lsis. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 218–222, Jan 1999.

[10] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[11] W. Jia, K. Shaw, and M. Martonosi. Mrpb: Memory request prioritization for massively parallel processors. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 272–283, Feb 2014.

[12] W. Jia, K. A. Shaw, and M. Martonosi. Characterizing and improving the use of demand-fetched caches in gpus. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 15–24, New York, NY, USA, 2012. ACM.

[13] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither more nor less: Optimizing thread-level parallelism for gpgpus. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 157–166, Piscataway, NJ, USA, 2013. IEEE Press.

[14] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba-cache: Adaptive blocks for eliminating waste in the memory

hierarchy. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 376–388, Washington, DC, USA, 2012. IEEE Computer Society.

[15] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 487–498, New York, NY, USA, 2013. ACM.

[16] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-driven dynamic gpu cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 67–77, New York, NY, USA, 2015. ACM.

[17] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng. Optimal bypass monitor for high performance last-level caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 315–324, New York, NY, USA, 2012. ACM.

[18] S. Li, D. H. Yoon, K. Chen, J. Zhao, J. H. Ahn, J. Brockman, Y. Xie, and N. Jouppi. Mage: Adaptive granularity and ecc for resilient and power efficient memory systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[19] J. S. Liptay. Structural aspects of the system/360 model 85: II the cache. *IBM Syst. J.*, 7(1):15–21, Mar. 1968.

[20] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[21] X. Mei and X. Chu. Dissecting gpu memory hierarchy through microbenchmarking. *arXiv preprint arXiv:1509.02308*, 2015.

[22] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A detailed gpu cache model based on reuse distance theory. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 37–48, Feb 2014.

[23] NVIDIA. Whitepaper - nvidia next generation cuda compute architecture: Fermi. NVIDIA, 2011.

[24] H. Packard. Inside the intel® itanium® 2 processor. *Technical White Paper*, 2002.

[25] M. Qureshi, M. Suleman, and Y. Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 250–259, Feb 2007.

[26] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[27] M. Rhu, M. Sullivan, J. Leng, and M. Erez. A locality-aware memory hierarchy for energy-efficient gpu architectures. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 86–98, New York, NY, USA, 2013. ACM.

[28] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society.

[29] T. G. Rogers, M. O'Connor, and T. M. Aamodt. Divergence-aware warp scheduling. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 99–110, New York, NY, USA, 2013. ACM.

[30] J. B. Rothman and A. J. Smith. The pool of subsectors cache design. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 31–42, New York, NY, USA, 1999. ACM.

[31] A. Sethia, D. Jamshidi, and S. Mahlke. Mascar: Speeding up gpu warps by reducing memory pitstops. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 174–185, Feb 2015.

[32] A. Seznec. Decoupled sectored caches: Conciliating low tag implementation cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 384–393, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[33] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[34] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. Jouppi. Cacti 5.1. *HP Laboratories, April*, 2, 2008.

[35] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 145–154, New York, NY, USA, 1999. ACM.

[36] X. Xie, Y. Liang, G. Sun, and D. Chen. An efficient compiler framework for cache bypassing on gpus. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 516–523, Nov 2013.

[37] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for gpus. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 76–88, Feb 2015.

[38] D. H. Yoon, M. K. Jeong, and M. Erez. Adaptive granularity memory systems: A tradeoff between storage efficiency and throughput. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 295–306, New York, NY, USA, 2011. ACM.

[39] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez. The dynamic granularity memory system. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 548–559, Washington, DC, USA, 2012. IEEE Computer Society.