Study of Elastic Cache and L2 cache lockdown for modern GPU architectures

Vivekanandan Kulumani Rajarajan (0030903255), Raunaq NandyMajumdar (0030819073)

Abstract

GPUs provide high-bandwidth/low-latency on-chip shared memory and L1 cache to efficiently service a large number of concurrent memory requests (to contiguous memory space). To support warp-wide access to L1 cache, GPU L1 cache lines are very wide. 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. In this paper, we implement **Elastic-Cache** that can efficiently support both fine- and coarse-grained L1 cache-line management for applications with both regular and irregular memory access patterns with a maximum speed up of 20%. We also extend this idea to L2 cache just for experimental performance analysis. We also implement **L2 Cache Lockdown** where some of the frequently accessed addresses in L2 cache are locked and are not evicted.

1. Introduction

GPUs rely on massive thread level parallelism to achieve high throughput. In GPUs, each streaming multiprocessor typically supports high bandwidth and low latency on chip shared memory and L1 cache to efficiently service many concurrent memory requests to contiguous memory space. In particular for supporting a warp-wide access to L1 cache, GPU L1 cache lines are wider than CPU. The L1 cache line in the GPU is 128 bytes long whereas in the CPU it is around 64 bytes long. Typically a single GPU cache line can support an access request for the entire warp provided that these accesses are coalesced into 128 byte contiguous memory spaces. Such coalescing reduces the number of memory accesses and makes the memory system more efficient.

Some applications such as graphics generate memory requests with regular access patterns and hence can benefit greatly from such L1 cache architecture where contiguous cache blocks in the same cache line stores data from the contiguous addresses. As GPU popularity grows and with more and more general purpose workloads starting to use GPUs, workloads with irregular memory access patterns and with irregular data structures like graphs are being ported to run on GPUs. These applications in particular cannot take advantage of wide width cache lines because irregular data structures often incur memory and branch divergence. These divergences will lead to irregular access patterns in the memory system and an increase in the number of memory accesses to non-contiguous memory space. Thus, this prevents GPU caches from efficiently servicing disparate memory requests and reduces the efficiency of cache line usage since a miss will evict and bring in an entire cache line even when the warp requested only a part of that. As such, many words brought to cache lines are never used until the cache lines are evicted. This is because (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 resulting in frequent replacements of cache line and wastage of the L2 cache bandwidth and off-chip main memory thus significantly degrading the performance.

Apart from L1 cache, GPUs also provide shared memory to enable communication and data reuse amongst threads of a thread block running on an SM. However, it is the responsibility of the programmer to efficiently use the shared memory. As such, we observe

that some applications do not use the shared memory space at all, which agree to many prior studies (e.g., [3]). For NVIDIA's Volta architecture, adaptive cache configuration has been implemented to handle this inefficiency effectively [5]. As the data in L1 cache and shared memory share the same physical space, when a certain application does not use the entire available shared memory, the remaining shared memory is allocated to L1 cache thus increasing its size, reducing capacity misses, reducing cache thrashing and improving IPC. But that still does not address the fact that benchmarks accessing irregular memory access patterns won't be able to leverage the full efficiency of a wide cache line. The same argument goes for Turing architecture as well [6]. For Pascal architecture the data cache is configurable between L1 cache and texture cache only [4]. There already has been work on implementing elastic cache to improve L1 cache efficiency for Fermi architecture [1] but our work compares the elastic cache performance primarily for Volta architecture where L1 cache size is increased and is implemented as streaming cache along with the sector cache implementation. Additionally, we extend the elastic cache features to L2 in Volta to study its performance. We also study the performance impact of locking down some of the frequently accessed addresses in the L2 cache of Volta.

2. Background

GPUs implementing Volta architecture provide 128KB on chip memory shared between L1 and shared memory. By default, L1 is 32KB and shared memory is 96KB [5]. The on-chip memory structure with 1024-bit (128-byte) I/O consists of numerous banks, each of which consists of 1024 rows. Depending on the need of shared memory for the application, the L1 cache size gets adaptively configured. When accessing the memory structure as shared memory, the GPU can issue up to 32 32-bit memory requests simultaneously to different row addresses as long as the requests access different banks. In contrast when accessing the memory structure as L1, the GPU accesses the same row address across the cache banks in lock-step to get a 128-byte cache line [1][8]. For example, if an application only assigns 8KB of the shared memory, then the L1 cache size is adapted to 120 KB. The L1 cache is implemented as streaming caches in Volta, which theoretically can handle an entire cache capacity of misses and is implemented by dedicating an MSHR for each of 1024 lines of the on-chip memory inside the GPGPU-Sim simulator after tedious micro benchmarking. In GPUs, L2 cache is shared across many cores. There are multiple memory partition units, each containing sub partitions, which contains an L2 cache. In Volta, both L1 and L2 are implemented as a sector cache. As discussed above, cache line size is 128B and each sector size is 32B in congruence with minimum granularity of GDDR5 memory technology. The benefit of sector cache is that it allows 32B access in the cache line and when a miss happens in one sector chunk in the cache line, an entire line gets evicted and partial fill is allowed. It does allow a better flexibility over baseline implementation, but it only allows contiguous addresses to be stored in different sectors across a cache line and does not address the concerns of irregular memory access patterns. Also, when a miss happens on a sector, the entire cache line gets evicted which is an overkill given that we have enough cache ports. Moreover, in Volta, L1 data cache is implemented in 4 banks and for efficient use, different sectors across the cache lines are interleaved across different banks to maximize parallelism. L2 is shared across cores and is divided across multiple memory partition unit and is not implemented as a streaming cache. The baseline sector cache implementation of L1 is fully-associative for faster access and L2 is 24 way set associative. The L2 cache size is 96 KB per memory sub-partition and 4.5MB in total. The tag and data array are separate physical structures. When a cache access occurs, the tag is looked up first and then in the

next cycle the data is accessed. If the tag is matched, the read/write operation is performed on the data. In case of a miss, the transaction is pushed into a miss-queue and the corresponding read/write operation is done depending on whether the access is to global memory or local memory.

3. Elastic Cache

We propose Elastic-Cache to support both fine- and coarse-grained cache-line management. In Elastic-Cache, the set number, associativity and the capacity of L1 cache are not changed. The baseline L1 cache has 1 set and 256 way set associativity and the L2 cache has 32 sets with a 24-way set associativity. To support fine-grained cache-line management, we divide a 128-byte cache line into *n* chunks. For example, there are four 32-byte logical chunks for $n = 4$. The analysis is done for n=4 because L1 cache has 4 banks and all the sectors in the sector cache are interleaved across those banks. Hence to leverage parallelism, we have 4 chunks each of size 32 bytes stored in separate banks. Each chunk is associated with a tag denoted by a chunk tag in this paper. A novel aspect of Elastic-Cache is as follows. (1) It stores chunk tags in a separate on-chip storage thus increasing the metadata associated with a cache line. (2) It can store chunks from non-contiguous memory space in a cache line in contrast to Sector-Cache [1] that can only store chunks from contiguous memory space in a cache line. Logical chunks of 64 bytes could also be analyzed but it will be detrimental as it will not be leveraging parallelism from all the 4 banks of L1 cache.

A. Design Considerations and Relaxations

[1] implements elastic cache with the chunk tag storage clubbed inside the unused portion of the shared memory for Fermi where the size of shared memory is rigidly configurable between 48KB or 16KB. But for Volta, the unused portion of shared memory is allocated to L1 thus improving performance. We didn't want to degrade the performance by reducing the adaptive cache size by limiting the allocation of chunk tag arrays into unused portions of shared memory thus reducing the degree of reconfiguration of L1 size. Also, using unused portions of shared memory for storing chunk tags leads to bank conflicts with data array accesses [1] which is avoided in our implementation. Hence, we implement chunk tag arrays as a separate physical storage to not interfere with the adaptive cache features and avoid bank conflicts. For the same reason, we didn't change anything in the shared memory design space. Our implementation will incur area overhead for both L1 and L2 tag arrays and it will be a little higher in L2 as it is shared across SIMT cores and is larger in size.

B. Chunk tags and common tags

We follow the use of 16 bits for the width of chunk tags as indicated in [1]. In modern GPUs, the total capacity of the main memory is usually more than 1GB. Thus, a single 20-bit tag is not sufficient to store all the necessary tag bits for each 32-byte chunk. To efficiently provide the full tag bits, we make use of the parts of the original tag bits of each 128-byte cache line and use them as a common tag. We assume that the full address space consists of 32 bits (4GB). For a conventional 32KB cache with 128-byte sets, 7 bits are used for the byte index. For L2, the default number of sets is 32, each with 128-byte line size. Thus, the byte index and set index are 7 and 5 bits respectively. The remaining 20 bits are used as tags. In Figure- 1, as each cache line contains four 32-byte chunks, we borrow the upper 2 bits from the byte-index to comprise the lower 2 bits of a 16-bit chunk tag. These two bits will be used to index into one of the four 32-byte chunks in a cache line.

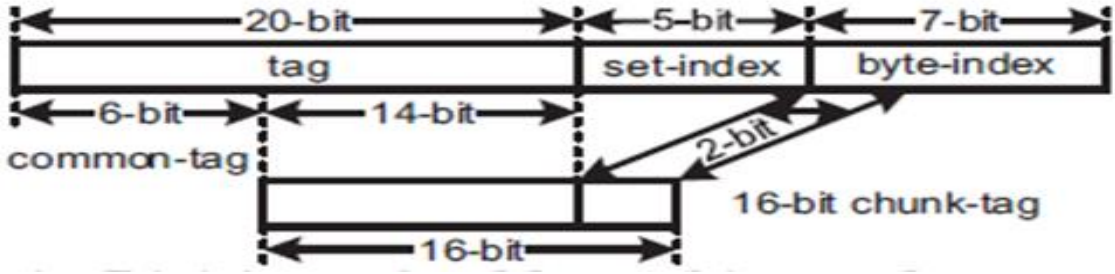

Figure-1: Elastic- Cache: Break up of address bits and common and chunk tags fields for 4-way 16KB Elastic-Cache [1].

The upper 14 bits are obtained from the lower 14 bits of the original 20-bit tag. Consequently, the common tag consists of only the upper 6 bits of the 20 bit tag. In summary, a 128 byte cache line has one 6-bit common tag and four 16-bit chunk tags. Note that if larger memory space is addressable, only the size of common tags is increased while that of chunk tags is unaffected. With a 6-bit common tag, a cache line of Elastic-Cache can store a 32-byte chunks in one of the 2^{32-5-6} bytes (2M bytes or 64K 32-byte chunks), where 5 and 6 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 in any of the 4 32 byte chunks slots in a cache line indexed by the same set index. We assumed an extra bit in the tag array of L1 cache to indicate whether the cache line contains one 128 byte line or multiple smaller chunks. So, now each cache line is associated with one common tag and 4 chunk tags whereas sector cache had only 1 tag. For a 128 byte contiguous accesses, the chunk tags will be contiguous and for non-contiguous accesses they won't be contiguous. The same breakdown is applied for L2 cache also, although later we conclude that making L2 also elastic is detrimental to the performance.

C. Basic Cache operation

In fine-grained cache-line management mode, Elastic-Cache simultaneously reads *n* common tags and *n*×4 chunk tags for an access where *n* is the number of ways. Then the common and chunk tags of an access address are compared against all *n* common tags and *n*×4 chunk tags. We can consider a 32 byte access as a hit only when its common tag is matched with one of the common tags in the set in addition to a chunk tag match with one of the chunk tags of the cache line. For example, let us assume that a 32 byte request (Request-A) to address 0x040000C0 needs to access L1 cache. As depicted in Figure-2, its common tag(0x01) and chunk tag(0x0002) are matched with CT-1 and CKT-11 respectively (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 of the chunk tags from CKT-31 to 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. This type of miss is similar to sector miss in the sector cache and we term it as "word-miss". On carefully observing the difference here, we see that in the sector cache for sector miss, we don't evict a sector but instead evict an entire line. However, in case of a word miss in elastic cache, we evict only one word in the appropriate block thus reducing traffic towards lower level memory. 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. For a 128-byte request, we still store the common tag in the

common tag array and at the same time, its four chunk tags are also stored in the chunk tag array. This is done so that subsequent 32-byte accesses to the same cache line can access or evict and replace only 32-byte chunks if needed. Lastly, only when a common tag and its four chunk tag are matched for a subsequent 128-byte request, a hit occurs. There are 4 chunk tag arrays for the entire cache, one for each chunk and it is only guaranteed for a 128 byte request that the chunks are directly mapped to the banks. This means that the chunk tags for 1st 32 byte are in bank 0 and the last 32 byte are in bank 3. But for a different 32 byte request, it is not guaranteed that the first empty slot will be available to store the 32 bytes. This is done to maintain the non-contiguous nature of the cache line. To be clearer, if the last 2 bits of the chunk tag for a 32 byte access is "11" and if bank 3 for that index is already occupied but bank 2 is empty, the chunk tag will be stored in bank 2. The way in which the possible bank conflicts are handled are discussed later. The same principle is applied for L2 also.

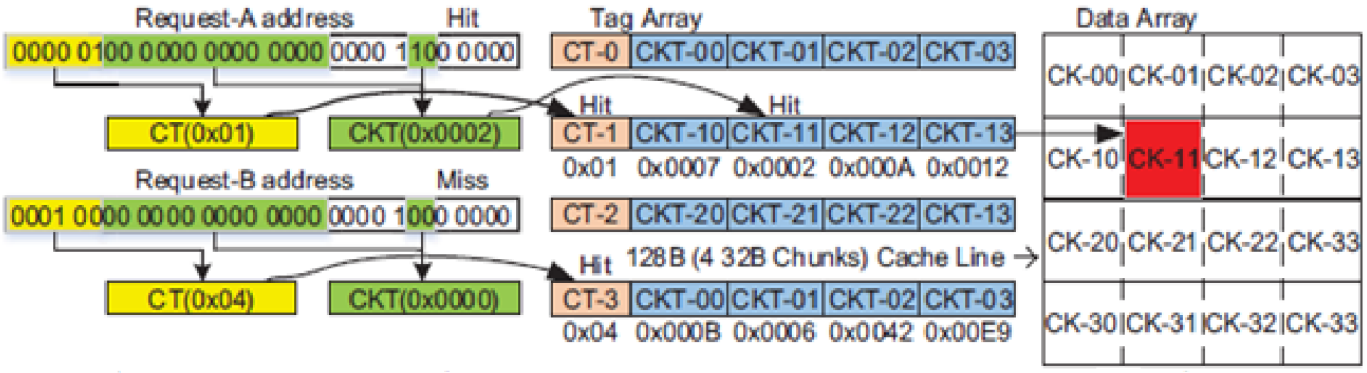

Figure-2: Elastic- Cache: Elastic Cache access example [1].

D. Cache architecture

To obtain all the chunk tags of one set in an access, chunk tag bank conflicts should be avoided. Consequently, we must guarantee that chunk tags of one line are stored in different banks of chunk tag arrays. When L1 cache is configured as 32KB, its associativity is 256 and thus there are 1024 chunks in a set. In the baseline GPU, there is a tag array separated from the on-chip memory. Hence, tags are read from the tag array for the next request while data are read from the data array for the current request. In elastic cache, first we read the common tag from the common tag array, then we read the chunk tag from the correct chunk tag bank in the same cycle to determine hit or miss. Consider the Figure-3 below. When the tag lookup comes to❸, it first goes to check if the common tag matches. If a common tag misses, then the line has to be evicted. If a common tag matches, then the chunk tag is looked up to find a hit or a miss. If a chunk tag miss happens, there is a sector miss which for sake of the report we will refer as "word miss" from now on.

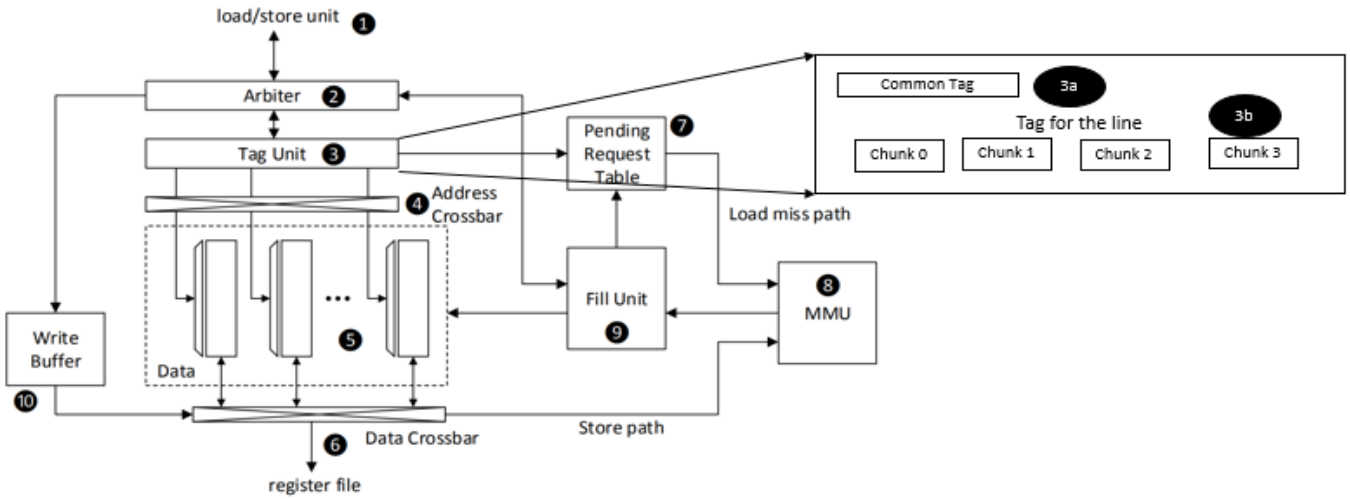

Figure-3: Modified L1-shared memory pipeline [8]

As L1 is already having 4 banks even for a coalesced 128 byte contiguous access, it eventually gets broken into 32 byte interleaved accesses across 4 banks. For such an access, each 32 byte sector is directly mapped to the correct chunk bank. Meaning if the bank ID is '11' from the address, then the request will be forwarded to bank 3 directly after common tag matches for chunk tag lookup. For non-contiguous access we use open linear addressing hashing to avoid bank conflicts. Consider Figure-4 and Figure-5 given below. There are two back to back accesses (access 1 and access 2) to the same cache line, (❶ and ❹), both of them matched in common tag, (❷,❺).Let's say the last 2 bits of chunk tags of the accesses are "11" and "10" respectively and at that time all the chunks are valid. So access 1 will first check chunk 3 (❸) while access 2 will check chunk 2 (❺). If it's a match, then it's a hit otherwise the access checks the next chunk. Let's say access 2 didn't match in chunk 2. Hence, chunk 3 (❻) is checked after which it rolls back to check chunk 0 (❼) and chunk 1 (❽). If none of them matches, then it's a word miss, and a 32 byte sector will be selected to evict later. By using open linear addressed hashing, we are avoiding possible bank conflicts in simultaneous accesses and extract possible parallelism. The same common tag and chunk tag lookup principles are applied to L2 cache also.

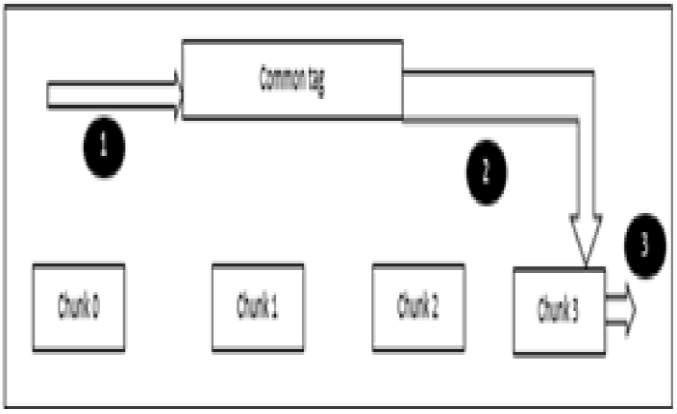

Figure-4: Example cache chunk hit access-1

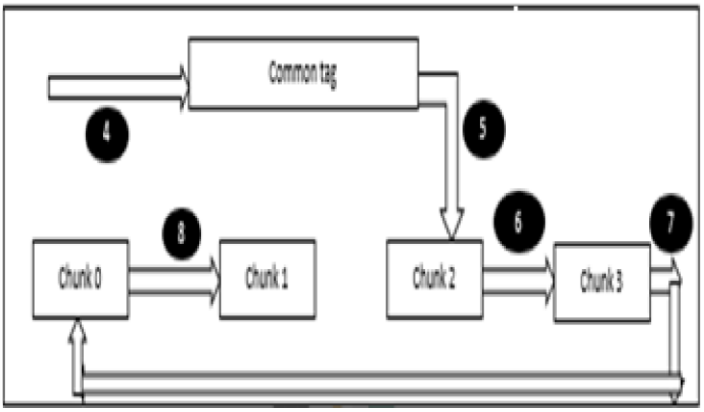

Figure-5: Example cache chunk miss access-2

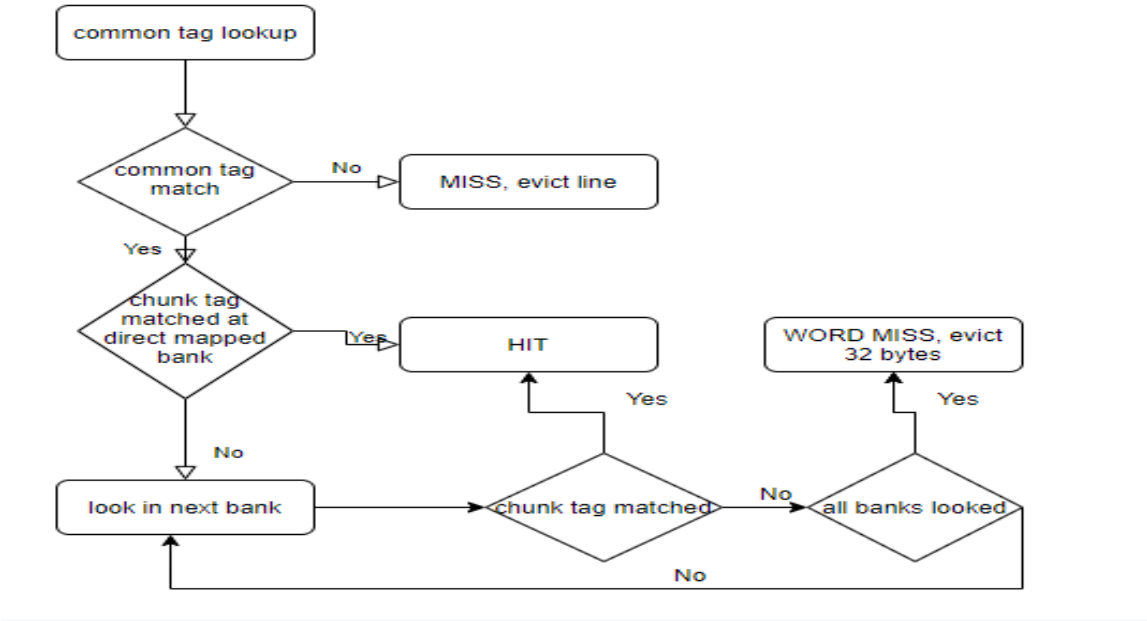

Figure-6: Access flow chart of chunk tags and data in Elastic-Cache

E. Replacement policy and coherence between line and sectors

Elastic-Cache 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 least recently used (LRU) maintained for all 128- byte cache lines per set, which is the same as the baseline cache. We have two possible eviction scenarios. Let us discuss the scenario when a miss occurs for a 32-byte request. Firstly, when the 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 from that cache line. To handle word misses, we maintain the timing info of each chunk as well and evict the LRU candidate. Secondly, 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 empty chunk slot of the cache line. After choosing the eviction candidate, we update both the LRU cache-line state of a set and the LRU chunk state of a cache line for each 32-byte request and 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 orchestrating the updates across all the partial chunks we take a simple approach. Whenever a partial miss is encountered, it is handled as a word miss on that chunk and the available valid chunks are returned to the requested instruction in case of reads. The chunks which are missing are fetched from the lower level and then provided to the requesting instruction. Correctness is still maintained because only after all the requested data have been provided, the dependent scoreboard registers are released. Warps proceed in lockstep. Unless the missed chunk is provided the warps will not proceed thus maintaining correctness. Even for writes, the correct write policies for different levels of memory and for different types of accesses (global or local) maintain correctness by default in case of partial misses. As

our minimum granularity is 32 bytes, partial misses are handled in a hassle freeway. Lastly, we still support partial read and write hits of 32- byte chunks to 128-byte cache lines brought by misses of 128-byte requests.

4. L2 cache lockdown

We thought it would be interesting to study the impact of L2 cache locking features for Volta architecture. [2] studies the same for GTX280 but given that the GPUs are moving towards more general purpose applications and the need for parallelism increases, modern day GPUs support larger L2 caches. GTX280 supported 768KB of L2 cache in total [2] and QV100 supports around 6MB of L2 with 3x more associativity [5] and 4.5MB in TITANV with 3x associativity. So, we implemented primarily 4 types of L2 cache locking to check its impact, and for TITANV, its impact was bare minimum. But we will discuss the implementation. The basic idea is to not evict the most accessed address from L2 cache. This has been implemented on top of baseline L2 implementation in GPGPUSIM.

A. Externally setting the locked addresses by profiling

We first profile the addresses which are accessed often (above a certain threshold) in the benchmark. During eviction if such an address becomes the eviction candidate, we don't evict it but evict the next best candidate. If all the ways are locked, then the access is bypassed to DRAM and its valid response to be sent to the shader core is directly passed to interconnect.

B. Profiling locked addresses on flight

While running the benchmark we store the addresses and keep a count of the number of times it is accessed and if it crosses a threshold then we push it into a locked list of addresses making it invalid for eviction. Then, we follow the same principle as discussed in section A.

C. Profiling locked addresses on flight with dynamically adjusting the access count

The profiling of addresses is similar to section B but after every access, we reduce the access count of all the addresses whose access count has gone above threshold. In this way, if an address goes into the locked list and never accessed again for a long time, in every cycle, its access count is reduced by 1 until it comes below the threshold value thus making it a suitable candidate for eviction again.

D. Relaxed locking

For the above three algorithms, if all the ways are locked, then access and response bypasses the L2 cache. But in relaxed locking if all the ways are locked, then we evict the way whose difference of access count with the threshold is minimum. For example, in a 4-way L2 cache, if all 4-ways are inside locked regions with respective access counts being 110,114,115,119 and the locking threshold being 100, we evict the way with access count 110 when needed.

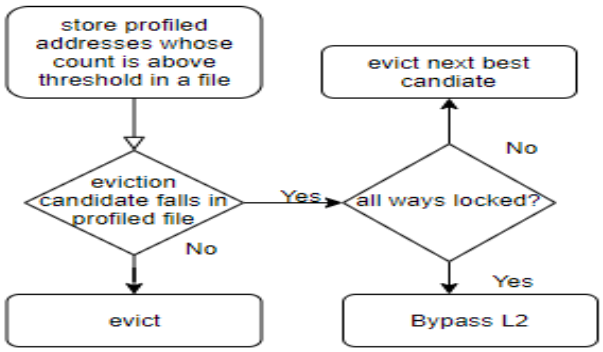

Figure-7: Locking address by external profiling

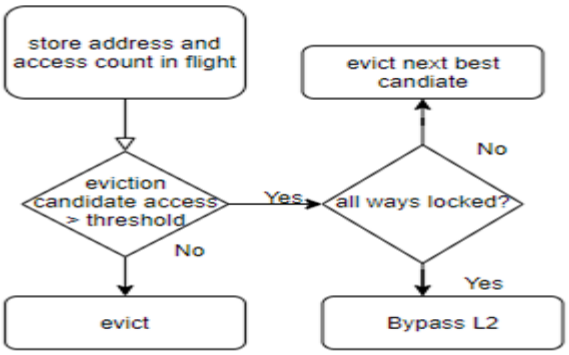

Figure-8: Locking address in-flight

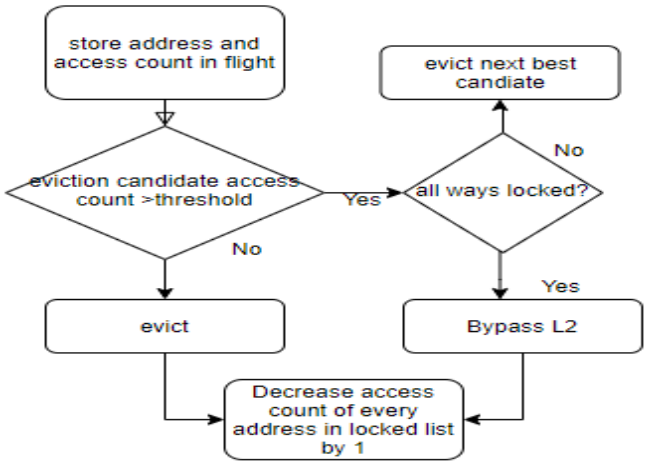

Figure-9: Locking address by adjusting access count

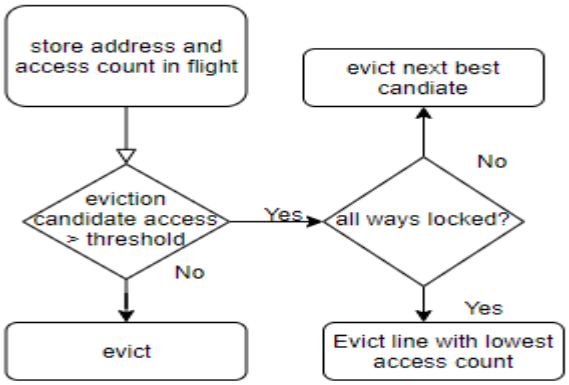

Figure-10: Relaxed lockdown

5. Methodology

We evaluated 11 benchmarks from rodinia benchmark suite. The benchmarks were back-propagation (back-prop), breadth-first-search (BFS), Needleman-wunsch (NW), nearest-neighbor (NN), path-finder (PF), heart-wall (HW), hotspot (HS), kmeans (KM), LU decomposition (LUD), speckle reducing anisotropic diffusion (SRAD), streaming-clusters (SC). We used GPGPUSIM version 4.0 [7]. We tested our implementation on TITANV, QV100, TITANX architectures. The first two configurations are used for runs on Volta architecture and the last configuration is used for runs on Pascal architecture. We analyzed our results for various cache configurations including adaptive configuration wherever supported, varied associativity and various cluster sizes.

6. Results

A. Performance

As stated before, we tried the elastic configuration for both L1 and L2 and compared the results with sector cache implementation. As shown in figure 11, it is seen that implementing both L1 and L2 as elastic cache degrades performance for all the benchmarks. Because L2 is larger in size and shared across multiple SIMT cores, bandwidth will only be properly utilized when accesses occur from L2 to DRAM in a coalesced manner.

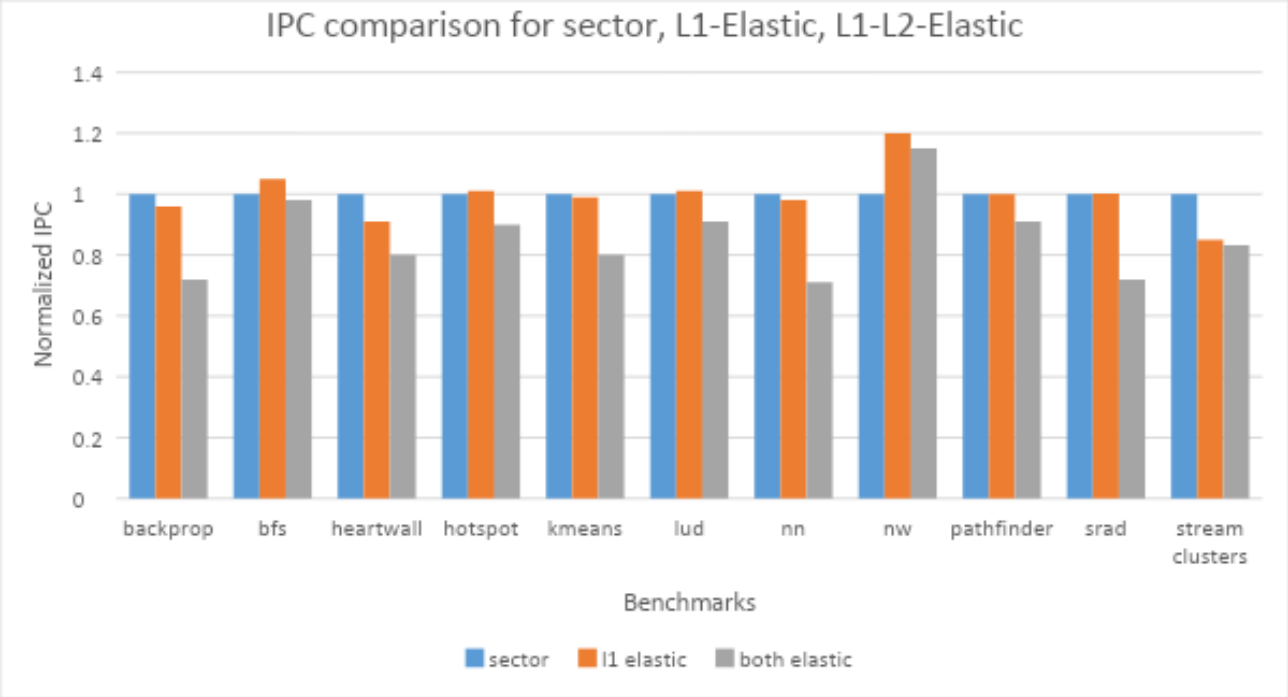

Figure-11: IPC Comparison between sector, L1-Elastic and L2-Elastic

If we only make L1 as elastic cache and keep L2 as sector cache then we can see irregular applications (applications having divergent memory access patterns) like BFS and NW getting good speedups. The speedup of NW is 1.2x and the speedup of BFS is 1.05x the sector cache implementation. Speedup of BFS is less than NW because the kernel execution times are less for a total of 8 kernels and after each kernel execution, L1 cache is flushed. NW is one of the most divergent applications and having very sparse memory-density in terms of access pattern, so it benefits the most. We can see in figure-11 that even when both L1 and L2 are elastic, NW gives performance improvement of 1.15x over sector cache implementation. For other applications with regular access patterns, elastic L1 does not degrade performance much. Backprop, heartwall, kmeans, NN, stream clusters experience a slight degradation of performance for L1 elastic and L2 sector implementation with an average degradation of 6.2%. Other regular applications like hotspot, LUD, pathfinder, SRAD gain slightly in performance by an average of 1%. NW gains by 20%, BFS gains by 5%. Figure-11 is plotted under adaptive cache configuration that's why L1 cache size could be anywhere between 32KB to 128KB based on need, and is fully-set associative. So, when cache size is large, meaning the application is using very less shared memory, the effect of elastic cache won't be prominent for some applications as increasing the cache lines allows more addresses to be present in cache, thus incurring lesser misses which is again augmented by the streaming accesses which theoretically allows infinite misses in flight, thus reducing the effectiveness of elastic cache. But still for NW, which reconfigures L1 to 120KB, elastic cache gives 20% speedup.

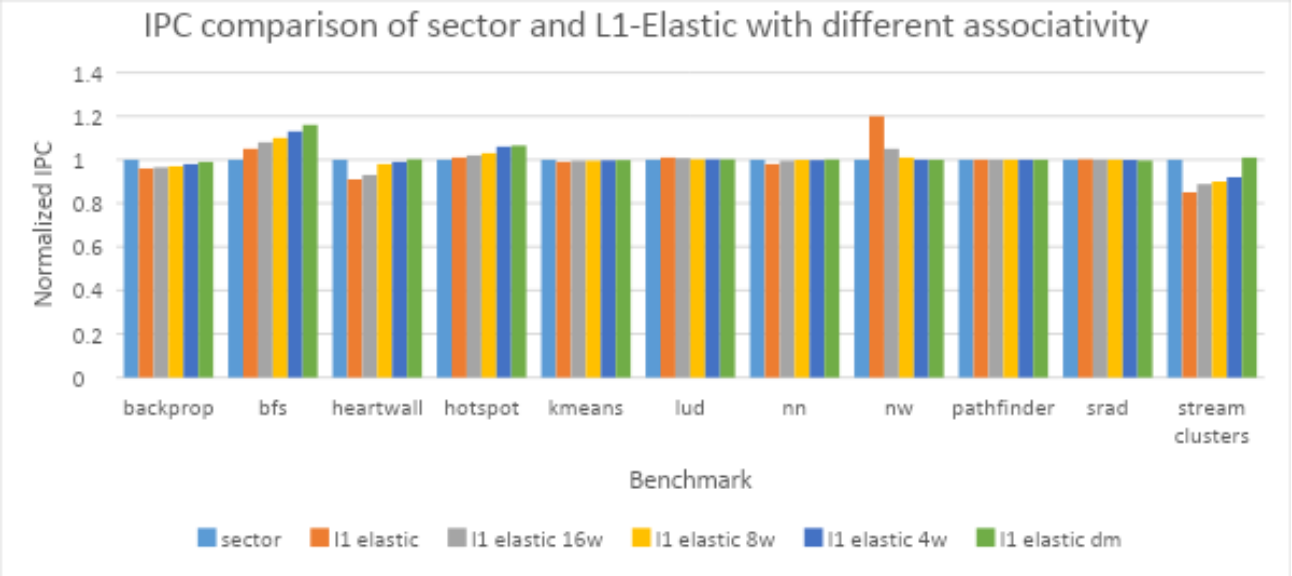

Figure-12: IPC Comparison of sector, L1-Elastic with different associativity

In figure-12, performance analysis was done by disabling adaptive cache configuration and fixing L1 cache size to 32KB and varying the associativity. We saw that sector cache performance doesn't change while varying the associativity of L1 cache but elastic cache performance changes. We see NW performs best in fully-associative cache. BFS gains substantially when L1 associativity is reduced and a maximum speedup of 1.16x is obtained in direct-mapped configuration. Some of the applications which had performance degradation in fully-associative cache, gain performance when associativity reduces and even outperform sector cache implementation. Heartwall, hotspot, nearest neighbor, stream-cluster gains performance improvement in direct-mapped configuration. Stream-clusters gain performance by 18% from fully-associative configuration. NW, SRAD, LUD degrades with reduced associativity.

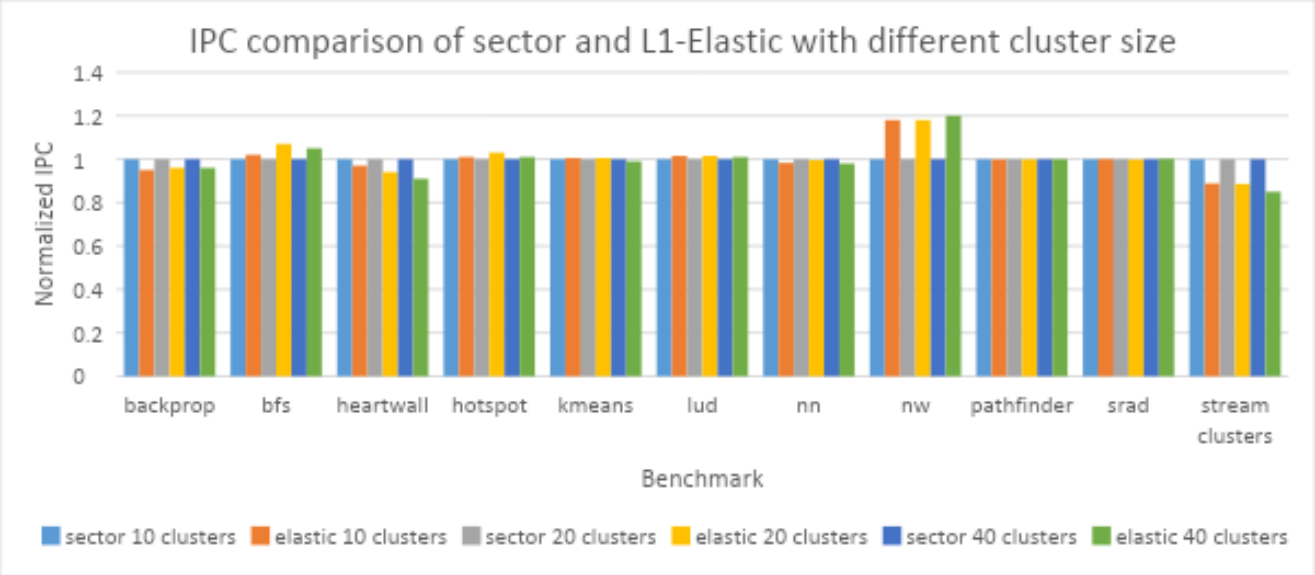

Figure-13: IPC Comparison of sector and L1-Elastic with different cluster size

Figure-13 plots the performance when the number of clusters changes, but the total number of cores are still the same. For 10 clusters, each cluster has 8 cores, for 20 clusters each cluster has 4 cores, for 40 clusters each core has 2 cores. L1 cache in this analysis is fully associative here. NW gives best performance for 40 clusters and BFS gives best performance for 20 clusters.

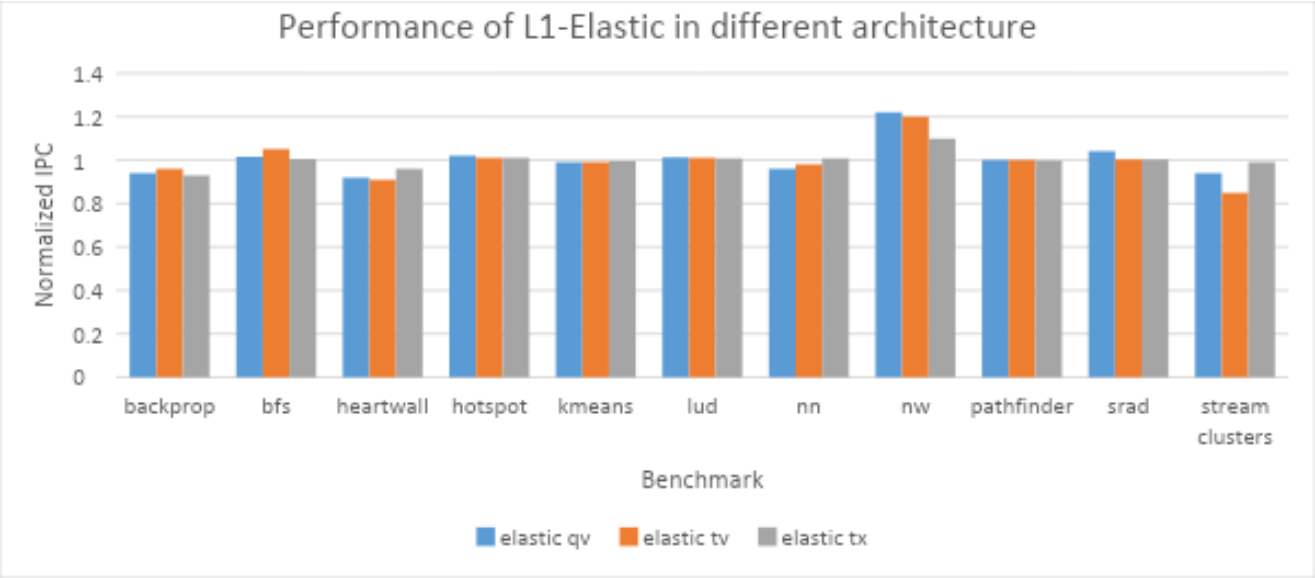

Figure-14: IPC Comparison of L1-Elastic in different architectures (QV100, TITANV, TITANX)

We analyzed our implementation in QV100 and TITANX also under default cache configuration. QV100 has 80 clusters, each cluster has 1 core and L1 cache size is still 32KB and supports adaptive cache configuration and TITANX supports Pascal architecture, L1 cache has 2 banks and each cache bank is 24KB making a total of 48KB and is unified with texture cache [4]. TITANX doesn't support adaptive cache configuration with shared memory [4]. For TITANX shared memory is of fixed size, 96KB [4]. Figure-14 analyzes the performance of elastic cache implementation for these 3 architectures. NW performs best with L1 elastic cache in QV100 by outperforming TITANV by 10%. BFS and NW degrades in performance in TITANX. Stream-cluster improves in performance by around 11% in QV100 and by 16% in TITANX. NN gains a little speedup in TITANX with elastic cache implementation whereas it sees a little slow down in TITANV and QV100.

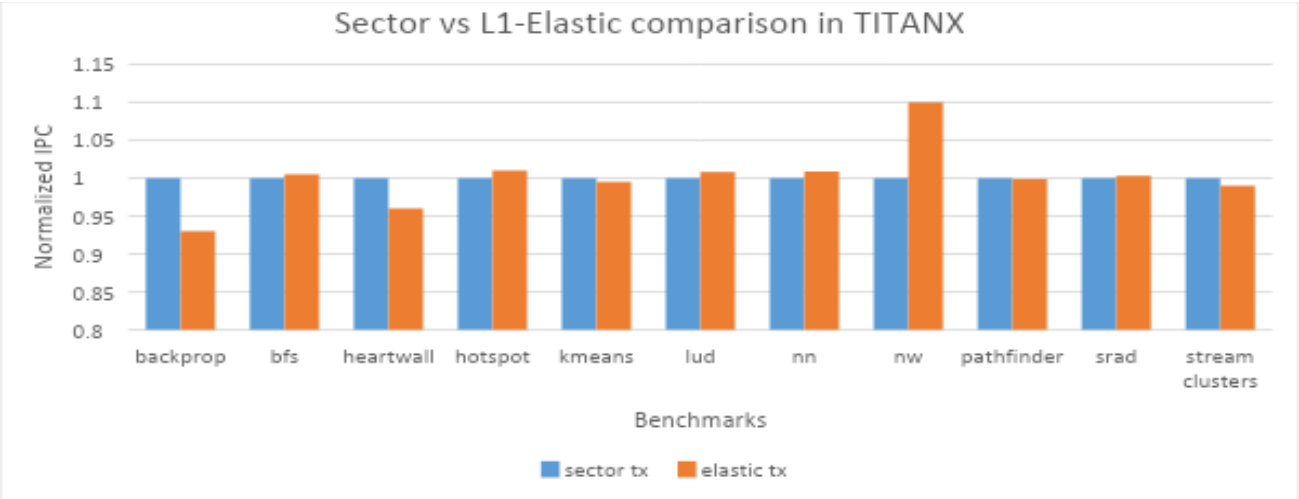

Figure-15: Sector vs L1-Elastic Comparison in TITANX

From TITANX and TITANV performance of elastic cache it gives an indication that elastic-cache will perform well in newer architectures like Turing, also has similar unified cache configurations [6].

Figure-15 separately plots performance of different benchmarks for TITANX as it runs on Pascal architecture. NW gains a speedup of 10% over sector cache, BFS gains speedup of 5%. Stream-cluster gains performance over TITANV performance but for all other benchmarks performance improvement or slowdown almost remain the same.

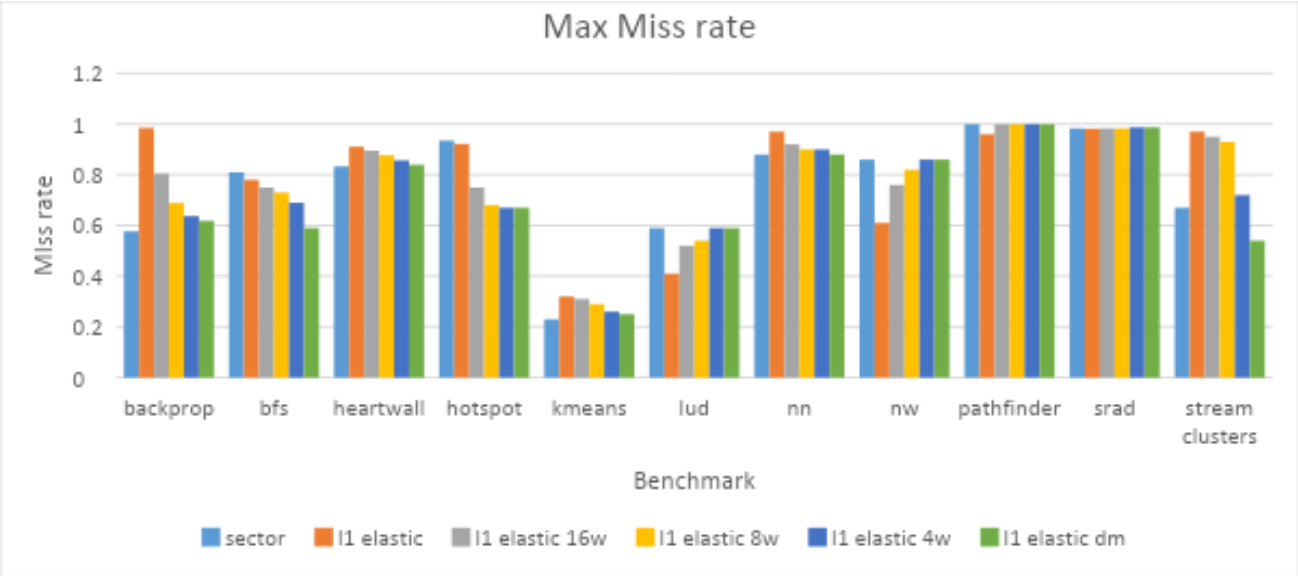

Figure-16: Max Miss rate

B. Miss rate

Figure-16 plots miss rate for different benchmarks in TITANV when L1 cache is not adaptively configured and the associativity is varied. NW has the lowest miss rate compared to sector cache when cache is fully associative. Its miss rate reduces by 17.4%. BFS has reduced miss-rate by 28% when the cache is direct-mapped and only 3% when it is fully associative. It can be seen from figure-16 that there are applications which have higher miss rate than sector, when associativity is high and it reduces and outperforms sector cache when associativity reduces, which is congruent with our earlier observations. It also gives us an idea that even when the cache is adaptive, the degree of adaptation can be extended towards associativity as well. For now, L1 cache in GPGPUSIM for Volta architecture is fully associative, supporting streaming accesses and depending on the shared memory size its associativity changes from 256 but it remains a fully-associative cache. If based on the application, we can change a cache from fully-associative to 4-way or direct mapped in flight, then it will help in overall performance for lots of applications. It is going to take extra hardware to correctly mask proper address bits for proper indexing but if such tradeoffs increase performance, it is recommended to go for such novel implementations to analyze performance.

C. Locked L2 cache results

As discussed above, we implemented L2 cache locking using various algorithms and various degrees of lockdown. But we found that except for NW it is not showing any performance improvement or degradation. We had implemented it for TITANV architecture where there are 24 memory partition units and 2 sub partitions in each of those. Each sub partition contains 96KB

of L2 cache and in total 4.5MB in total in the GPU. NW gains by 2% over sector cache implementation only when the locking threshold is low as seen in figure-17 over sector cache and relaxed cache locking implementations. If we increase the threshold to a higher value, then we don't get any performance variation. L2 cache is 24 way in TITANV, so it is clear that there are enough ways to evict a line even if a locked line is encountered. The rare corner case of the need for L2 bypassing when all ways are locked is not encountered. Hence, it is quite evident that as L2 cache size increases with every architecture, cache locking will not improve performance and will take extra hardware to implement the bypass logic. Also incorporating L1 elastic cache with locked L2 cache will not result in improved performance over L1 elastic and L2 sector cache implementation.

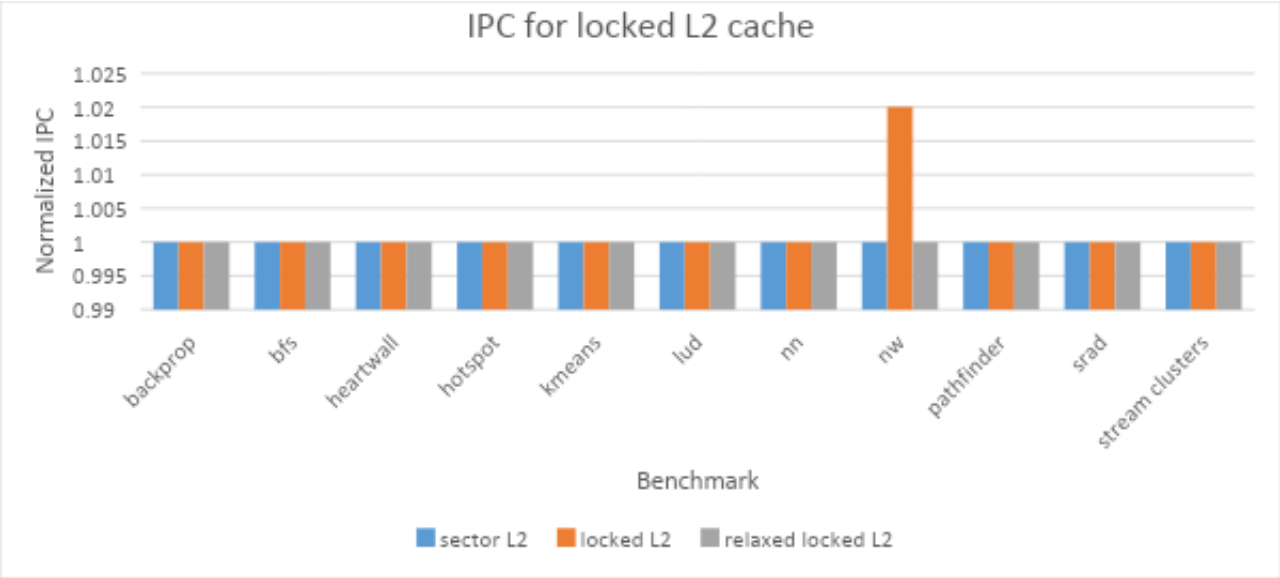

Figure-17: IPC for locked L2 cache

7. Conclusion and Future work

It has been shown that L1 elastic cache supports both coarse- and fine-grained operation with irregular applications seeing performance improvement and regular application seeing almost constant performance. It is also evident that elastic cache performance changes when associativity changes. So, if adaptive associativity can be incorporated, then depending on the application maximum performance can be extracted. As of now our implementation additionally supports adaptive dynamic associativity only for the set of benchmarks discussed in earlier sections allowing L1 to be configured as direct mapped or fully associative cache in flight to maximize performance gain of elastic L1 over sectored L1. [1] Shows elastic cache implementation for Fermi where caches are 16KB by default with 4-way associativity. For Volta, cache size is 32 KB and supports streaming accesses and hence it can handle a lot of misses in flight [5]. Hence for such reasons, the performance benefit of elastic cache is not that high in Volta architecture as compared to Fermi, but nevertheless it gives performance benefit and it's up to the designer to analyze the cost of the trade-off while deciding upon the cache architecture. But again, as discussed above, if adaptive dynamic associativity is incorporated along with compiler support to identify divergent applications like NW, BFS, then when need arises, associativity of the cache can be changed from full associativity to direct mapped or any desired associativity based on application requirement thus making performance of elastic cache scalable for any application so that

they can extract maximum benefit out of the elastic nature of L1 data cache. This will result in maximum performance and justify the hardware overhead required for implementing elastic L1. Implementing L2 cache in elastic configuration is a bad idea as it incurs a lot of area overhead with not so much performance gain in NW and performance degradation in other benchmarks. L2 cache locking is not a good idea in modern day architectures as when L2 cache sizes increase, locking doesn't improve or degrade performance much while unnecessarily adding area overhead. It would be interesting to analyze how to add compiler support to identify memory access patterns for applications beforehand and control the associativity of the elastic cache. Future work on improving the performance of L2 elastic cache should also take a look at the resulting impact in the performance of the DRAM memory controller when it is servicing continuous divergent memory access inside the same channel and also its impact on the DRAM timing parameters like tRCD, tRAS, tRC, etc.

References

1. Li, B. ; Sun, J. ; Annavaram, M. ; Kim, N.S. Proceedings - 2017 IEEE 31st International Parallel and Distributed Processing Symposium, IPDPS 2017, 30 June 2017, pp.82-91, Elastic-Cache: GPU Cache Architecture for Efficient Fine- and Coarse-Grained Cache-Line Management.
2. Picchi, J; Zhang, W Conference Proceedings - IEEE SOUTHEASTCON, 24 June 2015, Vol.2015-(June), Impact of L2 Cache Locking on GPU Performance.
3. Chen Zhao ; Fei Wang ; Zhen Lin ; Huiyang Zhou ; Nanning Zheng, 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), December 2016, pp.908-915 , Selectively GPU Cache Bypassing for Un-Coalesced Loads.
4. NVIDIA Tesla P100 GPU architecture, [online], available at: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf
5. NVIDIA Tesla V100 GPU architecture, [online], available at: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
6. NVIDIA Turing GPU architecture, [online], available at: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
7. Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, Tor M. Aamodt, Analysing CUDA Workloads Using a Detailed GPU Simulator, ISPASS , Boston, MA, April 19-21, 2009.
8. Tor M. Aamodt, Wilson Wai Lun Fung, Timothy G. Rogers, General-Purpose Graphics Processor Architecture.

YouTube presentation link: https://www.youtube.com/watch?v=i-0bw2jNxqI