2018 IEEE International Symposium on High Performance Computer Architecture

LATTE-CC: Latency Tolerance Aware Adaptive Cache Compression Management for Energy Efficient GPUs

Akhil Arunkumar Shin-Ying Lee Vignesh Soundararajan Carole-Jean Wu
School of Computing, Informatics, & Decision Systems Engineering
Arizona State University
Tempe, AZ 85281
{*akhil.arunkumar,lee.shin-ying,vignesh.soundararajan,carole-jean.wu*}*@asu.edu*

***Abstract*—General-purpose GPU applications are significantly constrained by the efficiency of the memory subsystem and the availability of data cache capacity on GPUs. Cache compression, while is able to expand the effective cache capacity and improve cache efficiency, comes with the cost of increased hit latency. This has constrained the application of cache compression to mostly lower level caches, leaving it unexplored for L1 caches and for GPUs. Directly applying state-of-the-art high performance cache compression schemes on GPUs results in a wide performance variation from -52% to 48%.**

To maximize the performance and energy benefits of cache compression for GPUs, we propose a new compression management scheme, called LATTE-CC. LATTE-CC is designed to exploit the dynamically-varying latency tolerance feature of GPUs. LATTE-CC compresses cache lines based on its prediction of the degree of latency tolerance of GPU streaming multiprocessors and by choosing between three distinct compression modes: no compression, low-latency, and high-capacity. LATTE-CC improves the performance of cache sensitive GPGPU applications by as much as 48.4% and by an average of 19.2%, outperforming the static application of compression algorithms. LATTE-CC also reduces GPU energy consumption by an average of 10%, which is twice as much as that of the state-of-the-art compression scheme.

I. Introduction

The recent advances in numerous computing domains such as artificial intelligence, machine learning, data analytics etc. are propelled by Graphics Processing Units (GPU) based acceleration. The high performance promise of GPUs is delivered through the unique feature of massively parallel fine-grained multithreading. This feature is used to hide pipeline resource contention and long latency memory access delays. However, recent characterization studies [6], [9], [11], [13], [19], [20], [31] have shown that not all long latency memory access delays can be well hidden for many GPU workloads, making the GPU memory subsystem a significant performance bottleneck. In particular, these applications are severely constrained by the peak bandwidth to main memory, interconnection network bandwidth, and limited data cache capacity. Furthermore, recent studies have highlighted the rising cost of data movement energy in processors of different scales [22], [23], [32]. Thus, improving the efficiency of the memory subsystem not only brings performance speedup but also has important energy saving implications.

Compressing cached data is a common way to increase the efficiency of cache hierarchies, which has been studied extensively for chip-multiprocessors (CMPs). Some [1], [18], [40], [41] focused on designing cache architectures for storing compressed cache lines of variable sizes, whereas others [2], [5], [7], [14], [35], [36] proposed high performance compression algorithms. While some researchers have studied the application of compression to GPU register files [26], texture caches [15] and off-chip memory links [24], [25], [34], [43], cache compression remains largely unexplored for GPU data caches.

Despite the overhead of decompression latency, cache compression has the potential to be well-suited to GPUs owing to their unique execution feature—a much higher latency tolerance than CMPs. While GPUs can have a much smaller per-warp cache allocation, if a warp encounters a miss in the cache, it can be readily replaced by executing other warps while the miss is serviced. Thus, GPU execution pipelines are kept busy, increasing resource utilization. However, during a poor latency tolerance phase with fewer ready warps in the GPU pipeline, any additional stall cycles can directly degrade performance. Therefore, to *truly* improve overall GPU application performance, one must take into account the latency tolerance of the architecture to make an informed decision on *what* to optimize in the architecture and *how*. We take the first step towards understanding GPU's latency tolerance, its capabilities and limitations. The goal of this paper is to design a higher performing memory hierarchy in the presence of the GPU latency tolerance and explore the potential benefits of cache compression for GPU's caches: the performance gain and the energy saving potential with reduced long latency stalls and lowered cost of data movement.

Directly employing existing cache compression designs, as we show in the paper, yields sub-optimal performance for GPU caches for a number of reasons. First, the cache compression schemes, that are typically designed to exploit one of two kinds of data value locality (namely, spatial and temporal value locality [39]), do not consistently improve performance across diverse GPGPU applications (Section II-A). Second, while many of the existing com-

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00028

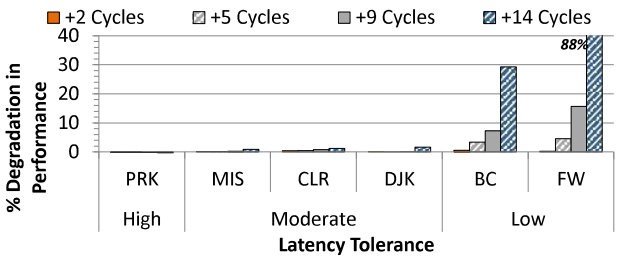

Figure 1: Sensitivity of sample GPU benchmarks to L1 data cache access latency[1].

pression schemes were targeted for CMPs, the performance bottlenecks in the GPU memory system are different. For example, CMP lower level caches are constrained by capacity and can better tolerate the increase in cache hit access times. Thus, last-level caches are often the focus of many cache compression works. In contrast, upper level caches in GPUs are more capacity-constrained and could benefit from cache compression. However, the effect of the associated cache access latency increase is largely unknown for GPU L1 caches, particularly, in the presence of the GPU latency tolerance feature.

Figure 1 shows the performance sensitivity of a few example GPU workloads to different L1 cache hit latencies, indicating that these workloads can tolerate a wide range of hit latencies. For example, `PRK` is insensitive to the increase in L1 cache hit latency whereas, `CLR` and `MIS` are able to tolerate by up to a 9-cycle increase of the hit latency. Other applications such as `BC` and `FW` experience performance degradation with the increase in the hit latency. As we later show in Section II, this latency hiding ability varies over time within an application as well.

Taking advantage of the unique opportunities in GPU architecture, we design an adaptive LATency Tolerance awarE Cache Compression management solution (LATTE-CC). LATTE-CC efficiently navigates the three way trade-off between compression benefit brought by different compression algorithms, their associated decompression latency, and GPU's latency tolerance. LATTE-CC does so by dynamically choosing one of three compression modes — no-compression mode, low-latency compression mode, and high-capacity compression mode, such that it maximizes GPU performance. LATTE-CC assesses the performance benefit brought by different compression modes, while accurately predicting the degree of latency tolerance available to hide the decompression latency incurred at the L1 cache hit path.

For the purposes of this paper, we use Base Delta Immediate (BDI) compression [36] and statistical compression (SC) [5], as low-latency and high-capacity compression modes respectively. The decompression latency of BDI is much lower than that of SC, while the compression ratio for BDI is also typically lower than that of SC for GPGPU applications. Nevertheless LATTE-CC is agnostic to the underlying compression algorithms and can be augmented with other compression hardware as well. We also demonstrate the flexible nature of LATTE-CC by evaluating it with a different compression algorithm, Bit Plane Compression (BPC) [24], as a component policy.

Our evaluation results over a wide range of workloads show that LATTE-CC outperforms Static-BDI and Static-SC compression schemes. LATTE-CC achieves an average of 19.2% (and by up to 48.4%) performance improvement over the baseline uncompressed cache while Static-BDI and Static-SC compression schemes improve performance by 13.7% and -8.2%, for cache sensitive workloads. Furthermore, our detailed energy consumption evaluation shows that LATTE-CC reduces GPU energy consumption by 10% on average which is twice as much as the energy reduction achieved by the next best policy, Static-BDI.

Overall, this paper makes the following contributions:

1) This is the first work that shows a naive application of compression algorithms on GPU's L1 data caches is not an effective option. While compressing caches data increases the effective cache capacity, not all workloads are able to tolerate the increase in the cache hit latency.
2) We demonstrate that the fine-grained latency tolerance estimation design proposed here is essential to take full advantage of cache compression, while hiding the associated increase in hit latencies (Section V-C). This approach allows LATTE-CC to outperform the offline optimal scheme which operates at a coarser, kernel boundary granularity (Kernel-OPT).
3) Finally, we show that an adaptive approach that minimizes miss counts, while is suitable for CMP systems, is sub-optimal for GPUs. The knowledge of GPU's temporally varying latency tolerance, when used intelligently, can be leveraged to achieve significant performance improvement (Section V-D).

II. Background and Motivation

Prior research have shown that L1 data cache capacity plays a crucial role in the performance of GPGPU applications [6], [9], [31]. A natural approach to increase the effective cache capacity in an energy efficient way is to adopt data compression in the cache hierarchy. For data compression to be beneficial 1) the data used by the applications must be compressible, and 2) the performance benefit given by the effective capacity increase must be greater than the penalty incurred by the increase in cache hit time. In other words, the decompression latency must be partially or entirely hidden from the performance critical path of an application execution. Section II-A first shows a detailed characterization study for data compressibility of GPGPU workloads and evaluates the performance benefit brought

[1]The access latencies shown here correspond to the decompression latencies of four state-of-the-art compression algorithms—BDI, FPC, CPACK+Z, and SC. Workload selection is described in detail in Section IV and Table III.

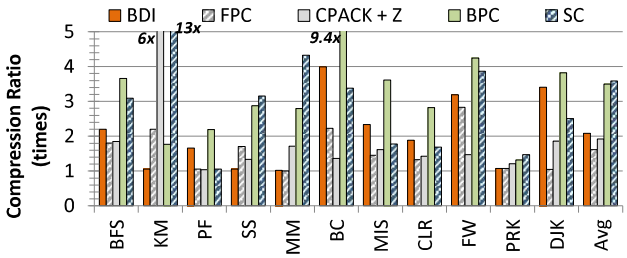

Figure 2: Compression ratio achieved by the state-of-the-art compression algorithms, i.e., BDI [36], FPC [2], CPACK-Z [14], BPC [24], and SC [5].

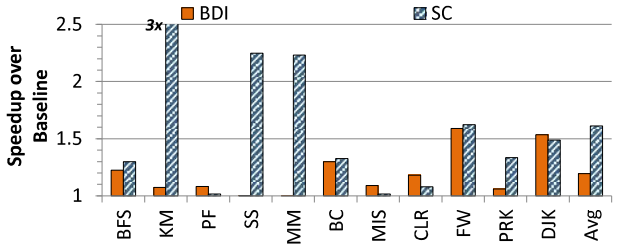

Figure 3: Performance impact of the effective cache capacity increase provided by data compression. The decompression latency is not taken into consideration here. Thus, the performance speedup shown here is the performance upper bound for static application of BDI and SC.

by the effective capacity increase. Section II-B assesses the degree of the decompression latency penalty that can be hidden in these workloads and by the architecture. Lastly, Section II-C motivates the need for adaptive compression designs in GPGPUs.

A. GPU Workload Data Compressibility

The effective capacity increase provided by data compression is a direct function of the data compressibility of applications. This data compressibility is dictated by the data values used and the algorithm itself. Prior work has observed value locality—data accessed by applications often has same or similar values during program execution [3]. Additionally, value locality can be extended to temporal value locality and spatial value locality [39]. Temporal value locality is the phenomenon where a particular data value is accessed repeatedly and spatial value locality is the phenomenon where the data values in adjacent memory locations are similar to each other. Data compression algorithms are designed to exploit the two distinct value locality properties.

The efficiency of a compression algorithm is measured as the achieved compression ratio — the ratio of the original data size and the resulting compressed size. To quantify the data compressibility of the workloads, we evaluate the compression ratio of all cache lines inserted in the L1 data caches with five state-of-the-art cache compression algorithms summarized in Table I: base delta immediate compression (BDI) [36], frequent pattern compression (FPC) [2], dictionary-based compression with zero-block detection (CPACK-Z) [14], [16], bit plane compression (BPC) [24], and huffman-coding based statistical compression (SC) [5]. Algorithms such as BDI, FPC, and BPC perform value compression by compacting identical or similar values within cache lines, exploiting spatial value locality. On the other hand, CPACK-Z and SC exploit temporal value locality by replacing identical values across multiple memory locations with shorter codes.

Figure 2 shows the varying degree of the data compression ratios achieved for a wide range of GPGPU workloads[2]. We observe that almost all applications exhibit a high degree of data compressibility. Applications, such as `BFS`, `BC`, `FW`, and `DJK`, achieve significant cache line size reduction with multiple compression algorithms and show both spatial and temporal localities in their data values. On the other hand, applications, such as `KM`, `SS`, `MM`, and `PRK`, show a significant affinity to the compression algorithms that exploit temporal value locality, whereas `PF` achieves more significant compression ratio with BDI and BPC, compression algorithms that exploits spatial value locality. This is due to the fact that the presence of spatial or temporal value locality in applications depends on the data types that are used [4]. Applications that operate on pointer and integer data typically contain low variance in data bits, thus exhibiting high degree of spatial value locality. On the other hand, high precision floating point data inherently contains high variance in the data bits. Thus, applications that operate on floating point data often have poor spatial value locality but exhibit high temporal value locality. This indicates a need for an adaptive algorithm that can exploit both the spatial and temporal value localities that exists but varies across workloads.

Figure 2 shows that commonly-used cache compression algorithms for CMP caches, i.e., FPC and CPACK+Z, do not achieve high compression ratios compared to BDI, BPC, and SC. Note that SC and BDI compression exploit complementary kinds of value locality and also represent two compression schemes with diverse decompression latencies. Thus, we focus our design and analysis with the combination of BDI and SC for the purpose of GPU's memory hierarchy optimization. Since there are a few workloads that prefer BPC compression in particular, we will study the inclusion of BPC compression in LATTE-CC later in Section V-E

Next, we characterize the expected performance gain that can be attained due to the increase in L1 data cache capacity. To isolate the performance improvement potential from the decompression latency penalty, we increase the cache capacity by employing static compression while assuming a zero decompression latency. Figure 3 shows that significant performance improvement can be achieved for a majority of the workloads. This serves as the performance upper bound for the workloads under the static applications of BDI and SC, respectively.

[2]Workload selection is described in detail in Section IV and Table III.

Table I: Comparison between the state-of-the-art cache compression algorithms.

Algorithm	Decomp. Lat.	Value Locality	Compressibility	Complex.
Base Delta Immediate (BDI) [36]	2	Spatial	Higher	Low
Frequent Pattern Compression (FPC) [2]	5	Spatial	Low	High
Cache Packer + Zero Value Compression (CPACK + Z) [14], [16]	8	Both	Low	High
Bit Plane Compression (BPC) [24]	11	Spatial	High	Moderate
Statistical Compression (SC) [5]	14	Temporal	Highest	High

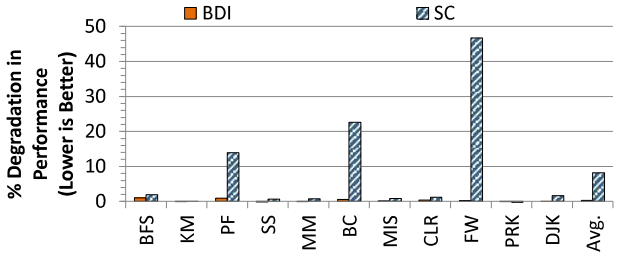

Figure 4: Performance degradation with increase in cache hit latency due to decompression. The cache capacity increase is not taken into consideration here.

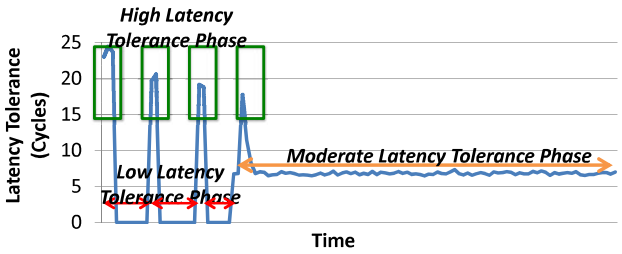

Figure 5: GPU latency tolerance characterization for SS GPGPU benchmark.

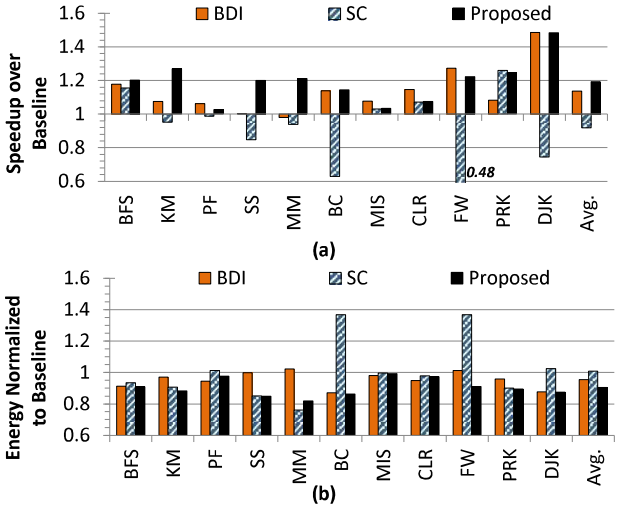

Figure 6: (a) Potential performance impact and (b) potential energy impact when BDI and SC are directly applied, and when an adaptive technique like LATTE-CC is applied

B. Latency Tolerance of GPUs

GPUs group a number of parallel threads and execute them simultaneously in single instruction multiple thread (SIMT) fashion. This group of threads that execute simultaneously is called a warp. GPUs are able to hide the stall latency from a warp with useful instruction execution from another warp through fast context-switching. By taking advantage of this latency hiding feature, we expect to see a part of or all of the decompression latency to be overlapped with the execution of other available warps in the GPU pipeline.

The availability of this latency hiding feature depends mostly on two factors. Firstly, the regularity in an application's memory access behavior influences the available latency tolerance. For instance, depending on the underlying warp scheduling policy, all warps in a GPU application could experience long latency memory access stalls at the same time. This results in low latency tolerance. Second, the GPU application might be characterized by varying amount of warp-level parallelism, possibly due to branch divergence.

To quantify the available latency tolerance in GPGPU workloads we measure the performance degradation caused by the decompression latencies of BDI and SC compression algorithms[3]. From Figure 4, we can see that some applications are highly sensitive to the decompression latency, while others are not. Applications, such as FW and BC, undergo significant performance degradation (47% and 22%, respectively), whereas PRK is able to tolerate the 14-cycle decompression latency of SC without experiencing any performance degradation.

C. Adaptive Compression in GPUs

Besides the varying degree and different forms of data value locality, and the varying degree of latency tolerance across different workloads, we also observe that the latency hiding ability of GPUs varies over time. This motivates us to delve deeper into investigating the temporal characteristics of GPU latency tolerance. We use the number of available warps in a GPU Streaming Multiprocessor (SM) as a proxy for the degree of latency tolerance and examine the time-varying latency tolerance for SS in Figure 5 as an example. The x-axis represents the application execution over time while the y-axis plots the latency tolerance. The latency tolerance represents the number of latency cycles that can be hidden by the GPU, described in detail in Section III-B2. We can see that SS goes through phases of varying degrees of latency tolerance, which dictates whether the decompression latency can be hidden or not. Therefore, exploiting the temporal variation in latency tolerance is critical to maximizing performance.

We characterize the performance improvement and energy reduction when BDI and SC are directly applied to the L1

[3]The decompression latencies are detailed in Section IV

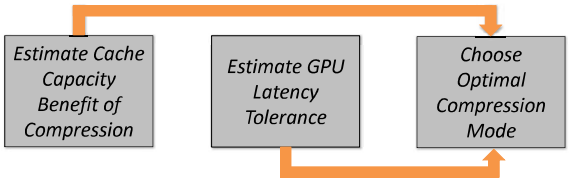

Figure 7: A conceptual overview of LATTE-CC.

data caches, taking into account both the capacity benefit and latency increase. Figure 6(a) shows the performance speedup for the GPGPU workloads under BDI (the first bars) and SC (the second bars). There is a significant variation from +48% to -52% in performance when a static cache compression method is applied. Similarly, a significant variation can be seen in the energy consumption (1.36x to 0.76x) when a static compression method is applied (Figure 6(b)). This is a compound effect of the performance gain from the increased cache capacity, the latency penalty from decompression and the temporal variations of latency tolerance. With a design that is able to exploit the variations of latency tolerance (the third bars) by switching between the available compression modes, additional performance and energy savings can be achieved. This is particularly significant for KM, SS, and MM.

Therefore, to achieve consistent high performance speedup and energy reduction, it is necessary to adopt a compression algorithm that achieves a higher compression ratio at the cost of longer decompression latency, during the execution phases of high latency tolerance. Similarly, it is also important to revert to a compression algorithm that incurs lower decompression latency at the cost of achieving potentially lower compression ratio during the execution phases of low latency tolerance. Finally, when compression brings no added benefit, it might even be necessary to switch off the compression feature. With this insight, we design LATTE-CC, an adaptive technique that learns the runtime latency tolerance of GPU workloads, estimates the performance benefit of different compression methods, and determines the best cache compression operation mode to maximize performance.

III. Latency Tolerance Aware Cache Compression Management

We propose an adaptive compression management approach, LATency Tolerance AwarE Cache Compression or LATTE-CC for the L1 data caches of GPUs. The key component of LATTE-CC is the design of an adaptive algorithm that dynamically predicts the best compression operating mode among the three choices: no compression (baseline), low-latency, and high-capacity modes, at runtime. The low-latency mode implements the BDI compression algorithm that exploits spatial value locality while the high-performance mode implements the SC compression algorithm which exploits temporal value locality available in applications. LATTE-CC is agnostic to the underlying compression algorithms and can be implemented with different compression hardware as well.

The dynamic compression mode selection is designed based on the performance trade-off of three important factors: cache capacity benefit brought by compression, decompression latency overhead, and the extent of GPU latency tolerance. Depending on the application locality and data value characteristics, the different compression modes result in different degrees of performance improvement. On the other hand, depending on the dynamically varying latency hiding ability of the pipeline, a varying degree of the decompression latency can be hidden. Thus, LATTE-CC is designed to adopt the compression decision to maximize the net performance improvement (Section III-A) by estimating both the benefit of cache capacity increase offered by the different compression modes (Section III-B1) and the dynamically varying latency tolerance of the GPU (Section III-B2). Figure 7 shows a conceptual overview of LATTE-CC's design and Figure 8 illustrates the LATTE-CC architecture and its three major components: the adaptive compression mode prediction algorithm, the cache capacity benefit estimator, and the latency tolerance estimator, in the context of a GPU.

A. Minimizing $AMAT_{GPU}$ for Optimal Compression Mode Selection

We use the average memory access time (AMAT) as a metric to combine the performance effects of cache capacity increase and decompression latency in the presence of GPU's latency tolerance. An application receives more performance gain from using one compression algorithm (Compr1) than using a different compression algorithm (Compr2) if $AMAT_{Compr1}$ is less than $AMAT_{Compr2}$.

Conventionally, $AMAT$ is given by

$$AMAT = \frac{total\_hit\_latency + total\_miss\_latency}{N_{hits} + N_{misses}} \quad (1)$$

where,

$$total\_hit\_latency = (N_{hits} * hit\_latency)$$
$$total\_miss\_latency = (N_{misses} * miss\_latency)$$

However, in the context of GPUs, the average memory access time that is experienced by the pipeline also depends on the GPU pipeline's latency hiding ability or latency tolerance. Therefore, $AMAT_{GPU}$ for a GPU should be expressed as

$$AMAT_{GPU} = \frac{total\_hit\_latency_{GPU} + total\_miss\_latency}{N_{hits} + N_{misses}} \quad (2)$$

where,

$$total\_hit\_latency_{GPU} = N_{hits} * min[(hit\_latency - latency\_tolerance), 0]$$
$$latency\_tolerance = \text{latency tolerance of GPU}$$
$$total\_miss\_latency = N_{misses} * miss\_latency$$

We utilize this notion of $AMAT_{GPU}$ to dynamically determine the better operating compression mode. In other

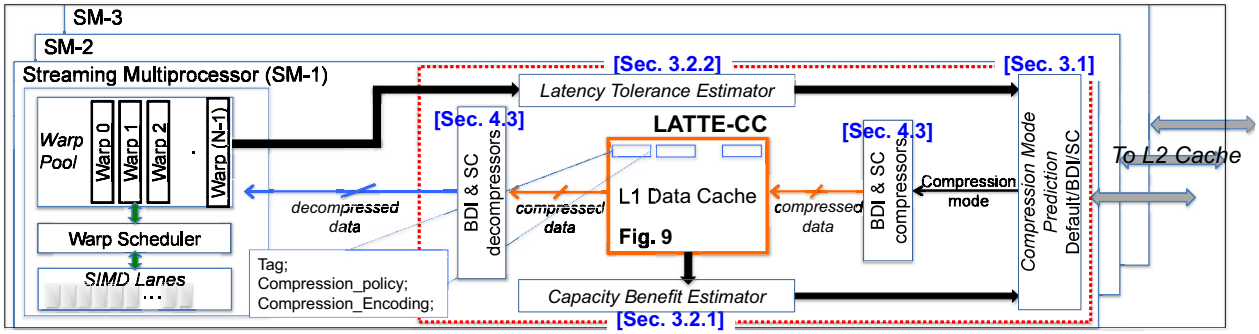

Figure 8: Block diagram of the modern GPU architecture with the LATTE-CC components.

words, LATTE-CC estimates the $AMAT_{GPU}$ for the different compression modes periodically and chooses the compression mode that minimizes the average memory access time experienced by the application.

B. Dynamic Estimation of $AMAT_{GPU}$

LATTE-CC is designed with the goal of capturing the dynamic application phase behavior. To accomplish this, LATTE-CC uses dynamic profiling to estimate $AMAT_{GPU}$ for the different compression modes periodically. LATTE-CC breaks down the application execution into multiple smaller periods that consists of learning and adaptive phases, each comprised of one or more *Experimental Phases (EPs)*.

1) Estimating Performance Improvement From Increased Effective Capacity: LATTE-CC uses the learning phase of each period to estimate the cache capacity benefit brought by different compression modes. This is done using a modified set sampling-based dynamic profiling method [37] as shown in Figure 9. During the learning phase EPs, LATTE-CC operates a small number of cache sets of the L1 data cache as the dedicated sets for the Default, BDI, and SC compression modes. Then, in the following EPs in the adaptive phase, the dedicated sets are operated as the follower sets (to minimize set sampling overhead) which always apply the winning compression mode.

In order to estimate the performance of the different compression modes, we implement two counters for each mode—one counts the number of cache line insertions of a compression mode ($N_{miss,mode_i}$) and the other counts the number of cache hits ($N_{hit,mode_i}$). These counters are incremented only on accesses to the corresponding dedicated sets. That is, during the learning phase EPs, $N_{miss,mode_{default/BDI/SC}}$ and $N_{hit,mode_{default/BDI/SC}}$ are incremented. Note that the benefit of compression might manifest later in time relative to the insertion time. Therefore, we allow $N_{hit,mode_{default/BDI/SC}}$ to continue its update on hits in the dedicated sets during one subsequent EP following the learning phase EPs. Since cache line reuses exhibit generational behavior, we do not expect to see many more cache hits beyond the EP following the learning phase. Thus, if the learning phase spans one EP, then $N_{hit,mode_i}$ is designed to count the number of hits during the first and second EPs of each period.

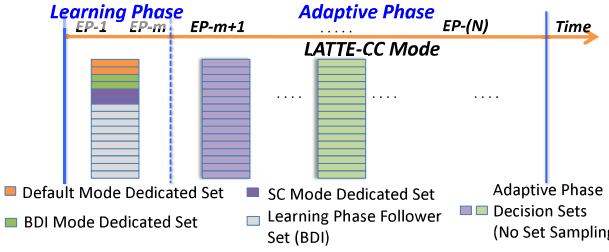

Figure 9: Modified set sampling in the LATTE-CC data caches.

Following the end of the learning phase, LATTE-CC is able to estimate the cache performance under the three operating compression modes using the dynamically measured hit and insertion counts from the dedicated sets.

2) Estimating Performance Penalty From Increased Hit Latency: In addition to the key element of exploiting data compressibility in GPU workloads, another important design question to address is whether and how the increase in cache hit latency can be overlapped with useful instruction execution in the GPU pipeline. An effective cache compression design has to take into account the time-varying degree of latency tolerance in GPU SMs such that the hit latency due to decompression can be hidden as much as possible.

We estimate the effective hit latency experienced by a compressed cache line as the sum of the decompression latency and the amount of time the line waits for service from the decompression unit (in a decompression queue). Therefore, the effective hit latency is

$$effective\_hit\_latency = decompression\_latency * (queue\_insertion\_pos + 1) \quad (3)$$

where,

$$queue\_insertion\_pos = the\ insertion\ position\ of\ the\ line\ in\ the\ decompression\ queue$$

To determine whether or not the $effective\_hit\_latency$ can be hidden, LATTE-CC uses the number of available warps as a proxy for the degree of pipeline latency tolerance. For example, when a compressed cache line receives a hit, the additional decompression latency is incurred for the de-compressor to provide the data to the requesting warp.

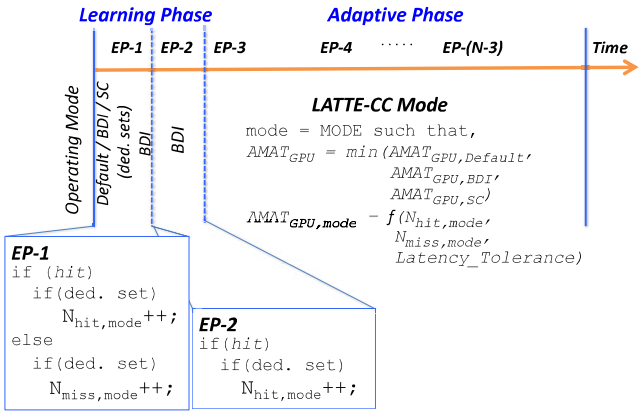

Figure 10: A temporal representation of LATTE-CC.

If there are other ready warps available for execution, the decompression latency becomes hidden and is overlapped with the execution of other warps.

In a round-robin warp scheduler, the degree of latency tolerance can be simply estimated as the number of available warps in the warp scheduler as the scheduler executes one instruction from each available warp before switching to the next. We utilize a more advanced Greedy-Then-Oldest (GTO) scheduler [38] in LATTE-CC. In schedulers similar to the state-of-the-art GTO scheduler, the scheduler tries to execute as many instructions as possible from each available warp before switching to the next. In such a scenario, the degree of latency tolerance can be estimated as follows:

$$latency\_tolerance = average\_warp\_available * average\_execution\_cycles\_per\_schedule \quad (4)$$

C. Putting it all Together

Figure 10 shows the LATTE-CC execution over time. LATTE-CC goes through a number of learning and adaptive phases to adapt to the run-time phase behavior in an application. LATTE-CC learns and predicts the operating mode that results in better cache hit performance in the learning phase. Furthermore, LATTE-CC continuously estimates the latency tolerance of the GPU pipeline in each EP of the adaptive phase. Finally, LATTE-CC chooses the optimal compression mode that maximizes the cache hit performance subject to the current degree of latency tolerance in each EP. By doing so, LATTE-CC always chooses the operating mode for each EP that results in the lowest $AMAT_{GPU}$.

IV. Experimental Setup

A. Simulation Methodology

We model LATTE-CC with GPGPU-Sim (version 3.2.2), a cycle-level GPU simulator [8]. The details of the simulated baseline system are given in Table II. This setup is similar to the baseline configurations used in other recent works [27], [29], [33], [38]. We implement BDI and SC compressors/decompressors, and a compressed data cache in GPGPU-Sim. The compressed cache is provisioned with four times the tag blocks and allows data to be stored in 32B sub blocks. This compressed cache organization is a simple modification to the existing data cache and similar cache organizations have been used in prior works [4], [18].

Table II: Baseline system configurations.

Parameter	Value(s)
Num. of SMs	*15*
Max. # of Warps per SM	*48*
Max. # of Blocks per SM	*8*
# of Schedulers per SM	*2*
# of Registers per SM	*32768*
Shared Memory	*48KB*
L1 Data Cache	*16KB per SM (128B lines/4-ways)*
L1 Inst Cache	*2KB per SM (128B lines/4-ways)*
L2 Cache	*768KB unified cache (128B lines/8-ways/12-banks)*
Min. L2 Access Latency	120 cycles
Min. DRAM access Latency	220 cycles
Warp Size (SIMD Width)	*32 threads*
Warp Scheduler	GTO

To analyze the energy consumption of LATTE-CC and other compression methods, we use a modified version of GPUWattch [28] that is augmented with the BDI and SC compressor and decompressor power models.

B. Benchmarks

We use a wide variety of GPU workloads taken from Pannotia [11], Rodinia [12], Mars [19], and NVIDIA SDK [30] benchmark suites to evaluate LATTE-CC and compare its performance with other designs. These workloads represent important computing domains such as web document clustering, web search, medical imaging, data mining, social network and graph analysis, financial modeling, and scientific simulations. We classify the applications into two categories based on their sensitivity to data cache capacity. We classify a workload as cache insensitive (C-InSens) if it experiences less than 20% performance speedup in the presence of a 4x larger data cache and as cache sensitive (C-Sens) if it experiences more than 20% performance speedup. The workloads and their input sets are summarized in Table III. We simulate each benchmark for 1 billion instructions or to completion, whichever is earlier. Similar methodology is used in recent GPGPU works [6], [10], [29].

C. Implementation Details

1) BDI Compressor/Decompressor Details: We model a 2/2-cycle compression/decompression latency, and 0.192/0.056 nJ compression/decompression access energy for BDI [4]. The BDI compression algorithm chooses a 2, 4, or 8B *base*, divides the cache line into blocks of size equal to *base*, and represents each block as a *delta* which is the difference of the value of the block from the *base*. This results in 10 possible encoding combinations of different *base* and *delta* as follows: (1) All zero; (2) *base* = 8B, *delta* = 0 (all blocks are same); (3) *base* = 8B, *delta* = 1, 2, or

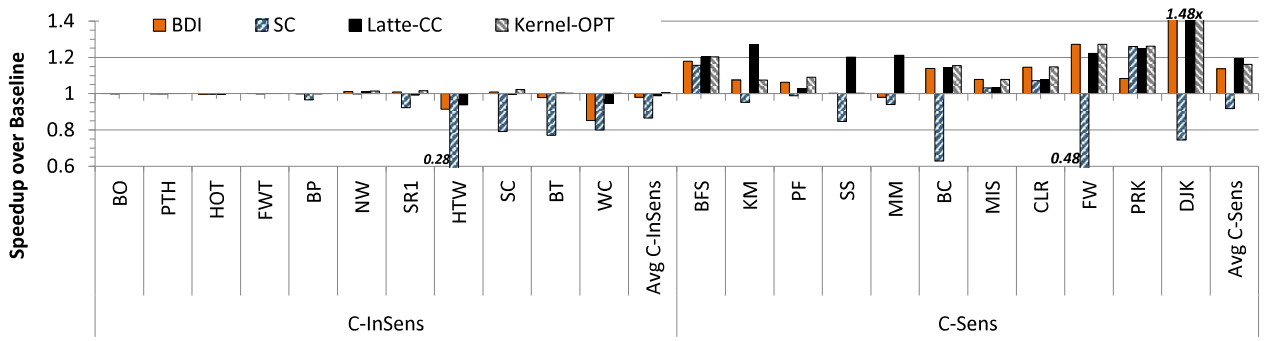

Figure 11: Performance improvement with LATTE-CC.

Table III: Benchmarks and input sets.

Abbr	Application	Input	Cat.
BO	Binomial Options [30]	512 Options	C-InSens
PTH	Path Finder [12]	100k nodes	
HOT	Hotspot [12]	512x512 nodes	
FWT	Fast Walsh Trans. [30]	32k samples	
BP	Back Propagation [12]	65536 nodes	
NW	Needleman-Wunsh [12]	1024x1024 nodes	
SR1	SRAD1 [12]	502x458 nodes	
HTW	Heartwall [13]	656x744 AVI	
SC	Streamcluster [12]	32x4096 nodes	
BT	B+Tree [12]	1M nodes	
WC	Word Count [19]	86kB text file	
BFS	Breadth First Search [12]	65536 nodes	C-Sens
KM	K-Means [12]	494020 objects	
PF	Particle Filter [12]	28x128x10 nodes	
SS	Similarity Score [19]	1024x256 points	
MM	Matrix Multiplication [19]	1024x1024	
BC	Betweenness Centrality [11]	1K (V), 128K (E)	
MIS	Maximal Ind. Set [11]	ecology	
CLR	Graph Coloring [11]	ecology	
FW	Floyd Warshall [11]	256(V), 16K (E)	
PRK	Pagerank (SPMV) [11]	Co-Author DBLP	
DJK	Dijkstra-ELL [11]	USA road NY	

4B; (4) *base* = 4B, *delta* = 0 (all blocks are same); *base* = 4B, *delta* = 1 or 2B; (5) *base* = 2B, *delta* = 0; (6) *base* = 2B, *delta* = 1B. These encodings are stored in the 4 bit *compression_enc* field within each tag block.

2) SC Compressor/Decompressor Details: We model a 6/14-cycle compression/decompression latency and 0.42/0.336 nJ compression/decompression access energy for SC [4]. The SC compression algorithm uses a Huffman coding based compression technique [21] to compress the cache lines. Huffman coding based compression assigns variable length codes to data values based on the probability of their occurrence. A shorter code is applied to a value which has a higher probability of occurrence. In order to generate Huffman codes for compression, a value-frequency table (VFT), that holds the data value and the frequency of it's use needs to be built.

To exploit the generational behavior of cache accesses in GPU applications, we revise the SC compression algorithm such that, a 1024-entry VFT with 12-bit counters, is built during the first EP of the first period, and is re-built during the final EP of each period. Therefore, during each period of EPs, SC (and LATTE-CC) uses a newly generated set of codes to perform SC compression. SC (and LATTE-CC) invalidates older cache lines when a new period starts.

SC utilizes a table of code-words in the compressor, and a lookup table for decompression (DeLUT). The hardware overhead is 5.5KB for VFT, 7KB for the compressor, and 3KB for the DeLUT. This translates to a total of 6.45% of the total data cache capacity (15.5KB/(16KB/SM*15 SMs)) for the GPU. Note, the bandwidth requirement of the SC decompressor is determined largely by the L1 data cache hit rate, which is typically in the range of 40% - 50% for GPGPUs. This places a relatively low bandwidth demand on the decompressor as compared to CMPs, whose L1 data cache hit rates are typically greater than 90%.

3) LATTE-CC Parameters: As detailed in Section III-B, LATTE-CC divides application execution into multiple Experimental Phases (EPs), each EP being 256 L1 cache accesses long. We empirically set 10 EPs to form a "period" consisting of learning (1-EP long) and adaptive (9-EPs long) phases. During the learning phase, we use four dedicated sets per compression mode. We utilize two additional bits per tag block to store the *compression_policy* information for each cache line. In total, LATTE-CC incurs a small microarchitectural overhead of 134 bytes to enable adaptive compression between multiple modes.

Based on our characterization, we observe that the write policy employed for GPU L1 caches has negligible impact on performance. Therefore, we model L1 caches as write-evict caches. This allows us the choice of not having to potentially evict other cache lines on write hits.

V. Evaluation Results

A. Overall Performance and Energy Impact

Overall, LATTE-CC improves GPU performance by an average of 19.2% (by as much as 48.4%) and reduces L1 data cache misses by 24.6% compared to the baseline uncompressed cache for the C-Sens category workloads. While Static-BDI achieves 13.6% speedup and 19.2% reduction in L1 data cache misses, Static-SC incurs a performance degradation of 8.2% despite achieving an impressive 28.7%

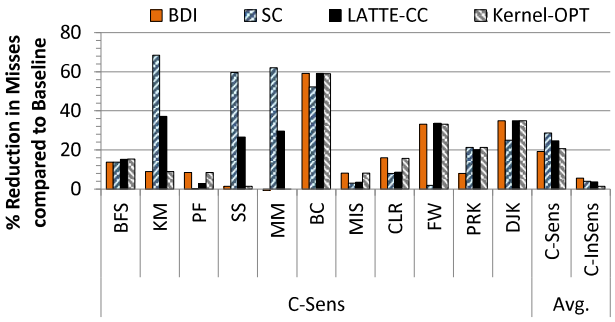

Figure 12: L1 cache miss reduction with LATTE-CC.

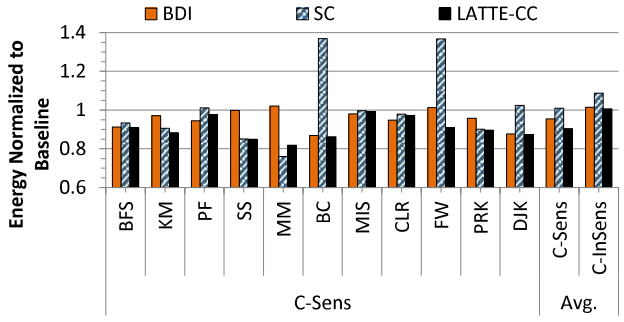

Figure 13: GPU energy consumption comparison.

reduction in cache misses for these workloads. For C-InSens category workloads, LATTE-CC and Static-BDI result in negligible performance change as these workloads are not sensitive to the additional cache capacity brought by compression. On the other hand, the decompression latency penalty incurred by Static-SC results in a significant performance degradation for many applications (e.g. HTW, SC, BT), leading to 13.4% performance degradation on average for C-InSens workloads. Figures 11 and 12 show the application performance speedup and the L1 miss reduction comparison for Static-BDI, Static-SC, and LATTE-CC compression designs.

Furthermore, as we see from Figure 2, some C-Sens workloads favor BDI compression (e.g. BC, FW, DJK), while others favor SC (e.g. PRK, KM). Additionally, although applications such as KM, SS, and MM achieve higher compression ratio and lower cache miss rate (Figure 12) with SC, Static-SC is unable to translate this into performance improvement due to the high degree of unhidden decompression cost.

One of LATTE-CC's key design feature is to predict and adopt the best performing cache compression mode while taking into account the degree of GPU's latency tolerance, dynamically. By doing so, it captures the diverse and time-varying behavior of the workloads. That is, for workloads such as BC, FW, DJK and others, LATTE-CC is able to get the performance benefits of BDI compression while realizing the increased capacity benefits of the SC compression for workloads such as KM, MM and others. This results in LATTE-CC achieving superior performance across diverse application behaviors with a robust 19% average speedup and 24.6% reduction in misses compared to the baseline.

Energy Saving: We observe that LATTE-CC is able to achieve significant energy savings compared to the baseline. LATTE-CC's energy impact comes from the following sources: reduction in application execution time, reduction in data movement in the cache hierarchy, overhead associated with compression and decompression operations, and reduction in the L2 cache energy due to reduced accesses. We take all these factors into consideration.

Figure 13 shows the energy consumption of the GPU, for the different compression schemes. For C-Sens workloads, LATTE-CC reduces the energy consumption by 10% while

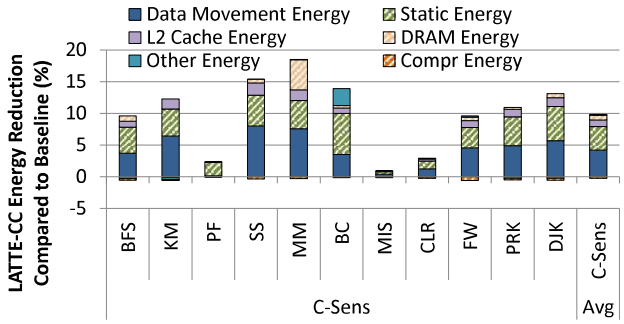

Figure 14: Breakdown of GPU Energy Reduction achieved by LATTE-CC.

Static-BDI does so by 5%. Static-SC on the other hand does not provide any energy savings on average. Among the C-InSens workloads, while LATTE-CC and Static-BDI do not alter the energy consumption considerably, Static-SC increases the energy consumption by 8.7% on average and by much as 53% for HTW workload.

Next, we take a closer look to understand the sources of energy reduction achieved by LATTE-CC. Figure 14 shows the breakdown of the energy reduction achieved by LATTE-CC for C-Sens workloads. We find that the reduction in data movement and static energy make up the bulk of the energy savings, providing 4.2% and 3.7% GPU energy reduction on average. Finally, we observe that the cost of compression and decompression energies is $< 0.25\%$ of the total GPU energy on average. *The energy analysis highlights the effectiveness of data compression in reducing the energy consumption in addition to GPU performance improvement.*

B. Comparing LATTE-CC with an Offline Optimal Policy

We also compare LATTE-CC to an oracular compression policy, Kernel-OPT. Kernel-OPT uses oracle knowledge from the end of each kernel of the application[4] to choose the compression mode that gives the lowest execution time for that kernel[5]. That is, while Kernel-OPT performs adaptive compression at a coarse kernel boundary granularity, LATTE-CC performs adaptive compression at a finer granularity

[4]Note that a kernel is the block of parallel execution running on the GPU which consists of multiple LATTE-CC learning and adaptive phases.

[5]Though such a policy cannot be implemented in hardware, it serves as a reference point for our studies.

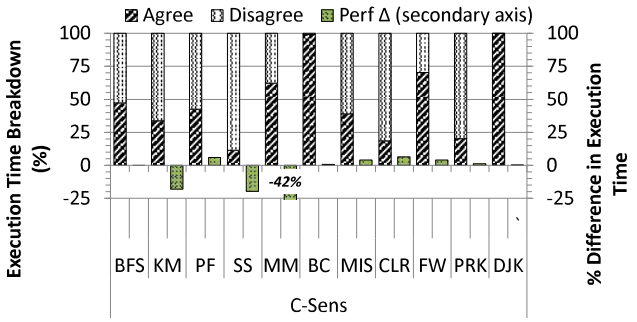

Figure 15: Comparison of LATTE-CC's compression mode decision with decision given by Kernel-OPT. (*Perf* Δ: Execution time difference between LATTE-CC and Kernel-OPT. Negative value means LATTE-CC performs better than Kernel-OPT.)

within each kernel. As seen in Figure 11, LATTE-CC is able to perform slightly better than Kernel-OPT, achieving 3% higher speedup on average and 4% greater miss reduction for the C-Sens workloads.

To compare LATTE-CC's compression mode decisions with those suggested by Kernel-OPT, we measure the fraction of execution time where LATTE-CC's prediction agrees with Kernel-OPT's (shown in Figure 15). The x-axis shows the different benchmarks and the primary y-axis shows the fraction of the total application execution time where LATTE-CC's decision agrees/disagrees with the decision given by Kernel-OPT. This execution breakdown is shown in the first column for each application. The secondary y-axis represents the performance gap between Kernel-OPT and LATTE-CC for the different benchmarks. This is shown in the second column for each application.

For applications such as BC and DJK, the compression mode decision of LATTE-CC is highly correlated with the decision given by Kernel-OPT. However, for others, the compression mode selected by LATTE-CC is different from the decision given by Kernel-OPT. This results in some lost opportunity for performance improvement in applications such as PF, CLR and PRK. This lost opportunity is shown by the *Perf* Δ bar in Figure 15. It is important to note that LATTE-CC is not designed to necessarily agree with Kernel-OPT as it operates at a much finer granularity than Kernel-OPT. This fine-grained runtime adaptation is particularly important for applications whose best compression operating mode changes over time. For such applications, LATTE-CC is able to achieve performance improvement that is significantly greater than what is suggested by Kernel-OPT (by as much as an additional 42%). KM, SS, and MM are applications that particularly benefit from the fine-grained adaptation as seen from the corresponding *Perf* Δ bars of Figure 15. Next, we examine the performance of SS in more detail.

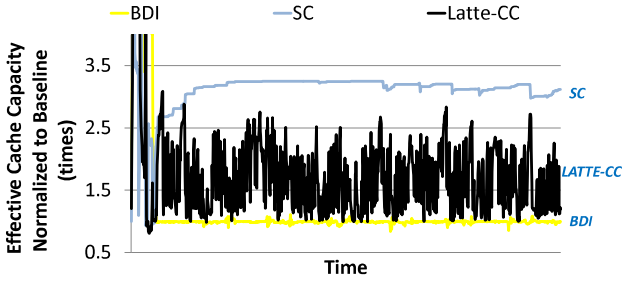

Figure 16: Effective cache capacity variation over time for Similarity Score (SS) application.

C. An Illustrating Application Example: Similarity Score (SS)

SS presents an interesting case. SS is a memory-intensive application whose performance is mainly restricted by the efficiency of the data cache. SS achieves a modest 0.3% performance improvement with Static-BDI, -15.3% with Static-SC and 20% with LATTE-CC. To investigate the performance of SS further, we measure the effective cache capacity relative to the baseline cache in Figure 16. The effective cache capacity is calculated as the integral sum of the uncompressed size of all valid compressed cache lines. Over time, Static-BDI consistently offers very small capacity benefit to SS. This is because BDI is unable to compress the data values used by SS significantly (Figure 2). On the other hand, SC achieves a much higher compression ratio of 3.2x, resulting in the highest effective cache capacity increase for SS (Figure 16) among the available compression choices. In the hypothetical scenario where this cache capacity increase could be leveraged without the performance penalty of hit latency, SC compression could introduce significant performance improvement (Figure 3). However when the increased hit latency due to decompression is taken into account, the performance of SS is significantly *degraded*. This is not only due to the high decompression latency of SC compression, but also the resulting resource contention at the decompressor caused by increased cache hits. The decompression cost outweighs the benefit brought by the capacity improvement.

To address the aforementioned shortcoming in the static schemes and to fully exploit the potential benefit brought by SC's high compression ratio, LATTE-CC dynamically assesses the degree of latency tolerance in the GPU pipeline and switches among the three compression modes, depending on the degree of latency tolerance. Over its execution period, SS goes through phases of high, moderate, and low latency tolerance (Figure 5). LATTE-CC is able to take advantage of the high and medium latency tolerance phases to choose SC compression during the periods of high data locality. This enables LATTE-CC to opportunistically achieve higher cache capacity when it is most beneficial. As shown in Figure 16, LATTE-CC's effective cache capacity hovers between 1-2X. As LATTE-CC chooses the compression mode

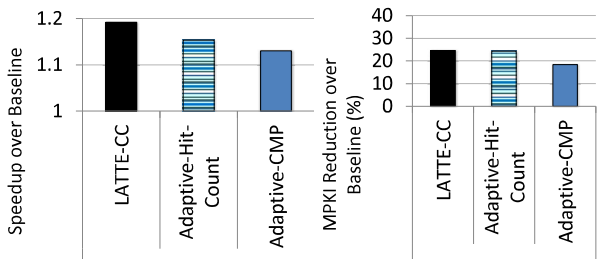

Figure 17: Performance comparison of LATTE-CC, Adaptive-Hit-Count, and Adaptive-CMP [1] policies for C-Sens workloads.

whose decompression penalty can be hidden, it minimizes decompressor resource contention and therefore results in significant net performance gain. This is due to the fine-grained adaptive compression mode selection of LATTE-CC. Therefore, LATTE-CC is able to achieve 20% performance improvement, significantly higher than Static-BDI and Static-SC compression. LATTE-CC also results in 26.6% decrease in L1 cache misses whereas Static-BDI, and Static-SC achieve 1.4%, and 59.6% miss reduction, respectively.

LATTE-CC's performance is also much higher than that of Kernel-OPT. By operating at a much coarser, kernel boundary granularity, Kernel-OPT loses the opportunity to take advantage of the runtime changes in latency tolerance within the kernel execution. This results in Kernel-OPT achieving only 0.3% performance improvement over the baseline. We observe a similar behavior with `KM` and `MM` which experience 26.9%, and 21.2% performance improvement under LATTE-CC. This performance improvement is significantly larger than that with Static-BDI, Static-SC, and Kernel-OPT for these workloads. *These applications highlight the advantage of LATTE-CC's fine-grained adaptive compression mode selection feature for GPUs.*

D. Benefits of Latency Tolerance Awareness

We next show that the optimization goal conventionally used for CMP caches— *the higher the cache hit rate, the better the performance is*—does not hold true for GPU compressed data caches. By accounting for the runtime latency tolerance of GPUs, LATTE-CC is able to achieve higher performance by sacrificing some cache hits in the process.

To illustrate the benefit of the latency tolerance awareness feature of LATTE-CC, we compare LATTE-CC to two other adaptive policies. First, we implement an adaptive policy that is purely based on the hit counts of the different compression modes—*Adaptive-Hit-Count*. The Adaptive-Hit-Count policy is based on the modified set sampling policy described in Section III-B1 without taking into account the decompression latency or runtime latency tolerance variation in the GPU pipeline. Second, we compare LATTE-CC to an adaptive compression management method proposed for CMPs [1] that takes into account the effect of decompression latency but not the latency tolerance of GPUs. We refer to this policy as *Adaptive-CMP*.

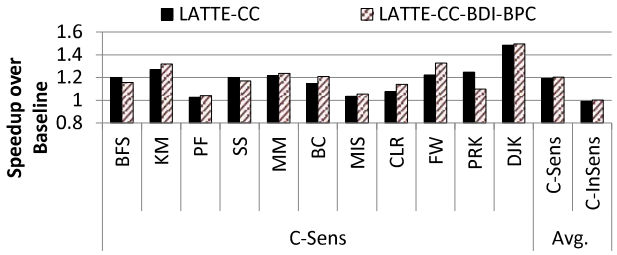

Figure 18: LATTE-CC performance with an alternative underlying compression algorithm.

Figure 17 shows the performance comparison of LATTE-CC in comparison with Adaptive-Hit-Count, and Adaptive-CMP policies. We can observe that although Adaptive-Hit-Count reduces misses by an average of 24.3% which is similar to that of LATTE-CC's miss reduction, this reduction in misses is not translated to performance improvement entirely. The *Adaptive-Hit-Count* policy experiences lower performance improvement compared to LATTE-CC, improving performance by only 15% over baseline. Similarly, Adaptive-CMP policy that is not aware of the GPU latency tolerance and hence performs sub-optimally compared to LATTE-CC. It achieves only 13% speedup over the baseline.

The performance of *Adaptive-Hit-Count* and *Adaptive-CMP* policies highlights two important aspects of GPU data cache designs—(1) Designs aimed to minimize miss counts, typically targeting CMP systems, are not always the best choice for GPUs. (2) The knowledge of GPU's time-varying latency tolerance is crucial and can be leveraged to achieve additional performance improvements.

E. Flexibility of LATTE-CC Design

Sensitivity to cache size: While data cache sizes of GPUs have been increasing over time, the working set sizes and bandwidth demands of GPGPU applications are also expected to increase. In such cases, cache compression will continue to be beneficial. To evaluate the effect of larger cache sizes, we evaluate LATTE-CC with the 48KB/16KB L1D cache/shared memory configuration that is typically available in NVIDIA GPUs. In this configuration, LATTE-CC is able to achieve an average of 6% performance improvement for cache sensitive applications while BDI improves performance by 3%.

Sensitivity to underlying compression algorithm: Thus far, we have focused LATTE-CC's design and evaluation, having BDI, and SC as component compression algorithms. This combination of algorithms offers qualitative diversity in terms of the kind of value locality exploited and the decompression latency incurred. However, it is important to note that LATTE-CC's adaptive algorithm design is agnostic to the underlying compression algorithms.

From Figure 2, we observe that there are a few workloads such as `PF`, `MIS`, `CLR`, and `FW`, that show affinity to BPC. In fact, we see that on average, BPC achieves a

similar compression ratio as SC. Therefore, BPC can be a plausible alternative to SC compression. Figure 18 shows the performance of LATTE-CC when it adaptively chooses between no-compression, BDI, and BPC compression modes (LATTE-CC-BDI-BPC). On average, we see that LATTE-CC-BDI-BPC performs similarly as LATTE-CC. This is reasonable considering that on average, BPC achieves a similar compression ratio (3.5x) as SC (3.6x), and its decompression latency (11 cycles) is also comparable to that of SC (14 cycles). Furthermore, we see that LATTE-CC-BDI-BPC performs better than LATTE-CC for workloads that show affinity to BPC compression i.e. `PF`, `MIS`, `CLR` and `FW`.

LATTE-CC is a flexible compression management design that can adapt to and maximize the performance advantages in the presence of larger caches and different underlying compression algorithms.

VI. Related Work

Compressed cache designs have been studied extensively for CMPs. Owing to the increase in hit latencies, they are typically not employed on the upper level of caches. This is the first work that carefully exploits the latency tolerating ability of GPUs to adaptively compress GPGPU L1 caches.

A. Data Compression in CMPs

Data compression for CMPs can be broadly categorized as compressed cache architectures [5], [18], [40], [41], compression algorithms [2], [14], [16], [36], cache replacement for compressed caches [7], [35], and main memory compression [17]. While these prior works all address the various design aspects of compressed caches for CMPs, the two closest related works that focus on adaptive compression techniques are [1], [4].

Alameldeen et al. [1] proposed a method to adaptively compress or not compress individual cache lines in a cache set. They consider the effect of decompression latency by noting that cache compression will be beneficial only if the performance benefit gained by compression offsets the decompression penalty that is incurred. However since their technique was proposed for CMP caches, it doesn't consider the impact of GPU latency tolerance.

More recently, Arelakis et al. [4] proposed a method to adaptively use one of several compression methods tailored to the data types of data being compressed. They develop heuristics to predict and identify different data types in hardware and choose a compression method that is known to yield the maximum compression ratio for a given data type. While their work estimates the benefit of cache compression that can be attributed to the increased cache capacity, they do not take into account the effect of increased hit latency or the latency tolerance that is available in GPUs. A direct application of such techniques would lead to sub-optimal performance improvement for reasons similar to those detailed in Section II-B.

B. Data Compression for GPU Memory

While this is the first work that explores the possibility of compressing L1 data caches in GPUs, data compression has been employed in GPU register files and off chip interconnect in GPUs. With the goal of reducing GPU register file power consumption, Lee et al. [26] proposed a method to compress the GPU register file. They observe that data held in registers exhibit low dynamic range for GPGPU applications. With this insight they use BDI [36] compression algorithm to compress the GPU register file and design the supporting compressed register file microarchitecture.

Pekhimenko et al. [34] proposed a toggle aware compression technique to reduce energy consumption while transferring compressed data, across the GPU interconnect. They noted that compression typically reduces the redundancy in bits and thereby increasing the amount of randomness that is seen per bit and thus leading to significant additional energy consumption due to bit-toggles. Vijaykumar et al. [43] propose to utilize idle cycles to compress the interconnect traffic with the help of assist warps. Another recent work by Sathish et al. [42] proposed a lossy technique to compress the traffic on the off-chip interconnect of GPGPU systems.

Kim et al. [24] propose bit plane compression, a compression algorithm tailored for GPUs that achieves high compression ratio by employing data transformation techniques to enhance and exploit spatial value locality in cache lines. They utilize BPC to compress the interconnect traffic on GPUs. Similarly, Lal et al. [25] propose to compress the interconnect traffic on GPUs using huffman compression, similar to SC compression [5] employed in this work. Compressing the interconnect traffic reduces bandwidth consumption significantly, and could result in significant performance improvement and energy reduction due to reduced congestion on the interconnect. The benefit provided by such designs is orthogonal to the benefit provided by LATTE-CC.

VII. Conclusion

This paper performs a detailed performance investigation for quantifying the impact of the GPU latency tolerance feature on GPGPU performance. By leveraging the latency tolerance of GPUs, we design a new adaptive latency tolerance aware cache compression management technique for GPGPU L1 data caches. LATTE-CC assesses the tradeoff between the capacity benefit given by multiple compression schemes that exploit different kinds of value locality and the performance penalty introduced by the corresponding decompression latencies. It then adaptively applies the best-performing compression mode during application execution. By doing so, the decompression penalty is overlapped with the runtime latency tolerance of the GPU. LATTE-CC can accurately predict the latency tolerance variation over application phases and from applications to applications. By operating at a finer-grainularity in time, LATTE-CC is demonstrated to perform better than an oracular scheme

(Kernel-OPT) which applies the static oracle compression decision at the application kernel boundary. Overall, LATTE-CC improves GPU performance by an average of 19.2% and reduces L1 misses by an average of 24.6% for a wide range of cache-sensitive GPGPU applications, resulting in a 10% reduction in overall GPU energy consumption.

ACKNOWLEDGEMENTS

The authors would like to thank the paper shepherd and the anonymous reviewers for their insightful feedback. The authors would also like to thank Michael Sullivan (NVIDIA) and the authors of Bit Plane Compression for providing the source code of the BPC algorithm. This work is supported in part by the National Science Foundation (Grant #CCF-1618039).

REFERENCES

[1] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proc. of the 31st Annual International Symposium on Computer Architecture*, 2004.

[2] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for L2 caches," April 2004.

[3] A. Arelakis and P. Stenstrom, "A case for a value-aware cache," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 1–4, Jan 2014.

[4] A. Arelakis, F. Dahlgren, and P. Stenstrom, "HyComp: a hybrid cache compression method for selection of data-type-specific compression methods," in *Proc. of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[5] A. Arelakis and P. Stenstrom, "SC2: A statistical compression cache scheme," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[6] A. Arunkumar, S.-Y. Lee, and C.-J. Wu, "ID-cache: instruction and memory divergence based cache management for GPUs," in *Proc. of the 2016 IEEE International Symposium on Workload Characterization*, 2016.

[7] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim, "ECM: Effective capacity maximizer for high-performance compressed caching," in *Proc. of IEEE 19th International Symposium on High Performance Computer Architecture,*, 2013.

[8] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proc. of the 2009 IEEE International Symposium on Analysis of Systems and Software*, 2009.

[9] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *Proc. of the 2012 IEEE International Symposium on Workload Characterization*, 2012.

[10] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia, "Managing DRAM latency divergence in irregular GPGPU applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC14*, 2014.

[11] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular GPGPU graph applications," in *Proc. of 2013 IEEE International Symposium on Workload Characterization*, 2013.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. of the 2009 IEEE International Symposium on Workload Characterization*, 2009.

[13] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Proc. of the 2010 IEEE International Symposium on Workload Characterization*, 2010.

[14] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas, "C-Pack: a high-performance microprocessor cache compression algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1196–1208, Aug 2010.

[15] M. Doggett, "Texture caches," *IEEE Micro*, vol. 32, no. 3, pp. 136–141, May 2012.

[16] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proc. of the 23rd international conference on Supercomputing*, 2009.

[17] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proc. of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[18] E. Hallnor and S. Reinhardt, "A unified compressed memory hierarchy," in *Proc. of 11th International Symposium on High-Performance Computer Architecture*, 2005.

[19] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[20] J. Hestness, S. W. Keckler, and D. A. Wood, "A comparitive analysis of microarchitecture effects on CPU and GPU memory system behavior," in *Proc. of the 2014 IEEE International Symposium on Workload Characterization*, 2014.

[21] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. of the IRE*, 1952.

[22] S. W. Keckler, "GPU computing and the road to extreme-scale parallel systems," in *Proc. of the 2011 IEEE International Symposium on Workload Characterization*, 2011.

[23] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *Proc. of the 2013 IEEE International Symposium on Workload Characterization*, 2013.

[25] S. Lal, J. Lucas, and B. Juurlink, "E2MC: Entropy encoding based memory compression for gpus," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[26] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient GPUs through register compression," in *Proc. of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[27] S.-Y. Lee, A. Arunkumar, and C. J. Wu, "CAWA: coordinated warp scheduling and cache prioritization for critical warp acceleration of GPGPU workloads," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[28] 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*, 2013.

[29] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-based cache allocation in throughput processors," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[30] NVIDIA, "CUDA C/C++ SDK code samples v4.0," 2011. [Online]. Available: http://docs.nvidia.com/cuda/cuda-samples

[31] M. A. O'Neil and M. Burtscher, "Microarchitectural performance characterization of irregular GPU kernels," in *Proc. of the 2014 IEEE International Symposium on Workload Characterization*, 2014.

[32] D. Pandiyan and C. J. Wu, "Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms," in *Proc. of the 2014 IEEE International Symposium on Workload Characterization*, 2014.

[33] G. Pekhimenko, E. Bolotin, M. O'Connor, O. Mutlu, T. Mowry, and S. Keckler, "Toggle-aware bandwidth compression for GPUs," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2015.

[34] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A case for toggle-aware compression for GPU systems," in *Proc. of the 22nd IEEE International Symposium on High Performance Computer Architecture*, 2016.

[35] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting compressed block size as an indicator of future reuse," in *Proc. of the IEEE 19th International Symposium on High Performance Computer Architecture*, 2015.

[24] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[36] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[37] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-aware cache replacement," in *Proc. of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[38] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proc. of the 45th IEEE/ACM International Symposium on Microarchitecture*, 2012.

[39] S. Sardashti, A. Arelakis, P. Stenström, and D. A. Wood, *A Primer on Compression in the Memory Hierarchy*, ser. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2015.

[40] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *Proc. of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[41] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *Proc. of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[42] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory I/O link compression for improving performance of GPGPU workloads," in *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[43] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps," in *Proc. of the 42nd Annual International Symposium on Computer Architecture*, 2015.