SLC: Memory Access Granularity Aware Selective Lossy Compression for GPUs

Sohan Lal, Jan Lucas, Ben Juurlink
Technische Universität Berlin, Germany
{sohan.lal, j.lucas, b.juurlink}@tu-berlin.de

***Abstract*—Memory compression is a promising approach for reducing memory bandwidth requirements and increasing performance, however, memory compression techniques often result in a low effective compression ratio due to large memory access granularity (MAG) exhibited by GPUs. Our analysis of the distribution of compressed blocks shows that a significant percentage of blocks are compressed to a size that is only a few bytes above a multiple of MAG, but a whole burst is fetched from memory. These few extra bytes significantly reduce the compression ratio and the performance gain that otherwise could result from a higher raw compression ratio. To increase the effective compression ratio, we propose a novel MAG aware Selective Lossy Compression (SLC) technique for GPUs. The key idea of SLC is that when lossless compression yields a compressed size with few bytes above a multiple of MAG, we approximate these extra bytes such that the compressed size is a multiple of MAG. This way, SLC mostly retains the quality of a lossless compression and occasionally trades small accuracy for higher performance. We show a speedup of up to 35% normalized to a state-of-the-art lossless compression technique with a low loss in accuracy. Furthermore, average energy consumption and energy-delay-product are reduced by 8.3% and 17.5%, respectively.**

I. Introduction

Memory compression has been demonstrated as a promising alternative to increase memory bandwidth [1], [2], [6], however, memory compression techniques often exhibit a low effective compression ratio. The main reason for the low effective compression ratio is the large memory access granularity (MAG) exhibited by GPUs due to wide bus width and large burst length. For example, MAG of GDDR5/5X/6 is 32B resulting from 32-bit bus width and 8 burst length. MAG is the amount of data read from or written to a memory by a single read or write command. MAG reduces the compression ratio as data can only be fetched in a multiple of MAG but a compressed block is often not a multiple of a MAG. For example, for a compressed size of 36B, we fetch 64B. Thus, a compression ratio that seems close to 4× (3.6×, assuming a typical block size of 128B in current GPUs) is actually only 2×. This leads to a significant difference between the raw and effective compression ratio actually gained by a system. The raw compression ratio is calculated without considering MAG, while the effective compression ratio is calculated by scaling up the compressed size to the nearest multiple of a MAG.

Figure 1 shows the raw and effective compression ratios of BDI [4], FPC [5], C-PACK [1], and E^{2}MC [6] techniques for several memory-bound benchmarks. The geometric mean (GM) of the effective compression ratio of BDI, FPC, C-PACK and E^{2}MC is 22%, 19%, 18%, and 23% less than the GM of the raw compression ratio, respectively. The low effective compression ratio reduces performance benefits, otherwise possible from a higher raw compression ratio.

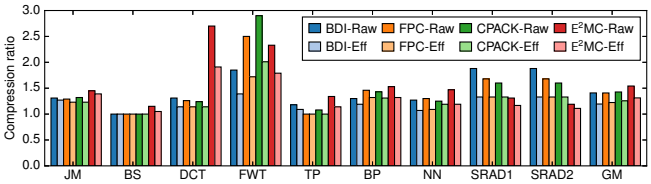


Fig. 1: Raw and effective compression ratio of BDI, FPC, C-PACK and E^{2}MC using MAG of 32B.

Interestingly, our study of the distribution of compressed blocks (presented in Section II-B) shows that a significant percentage of compressed blocks have only a few bytes above a multiple of MAG. With the goal to reduce the compressed size by these extra bytes, we propose a novel MAG aware Selective Lossy Compression (SLC) technique. The key idea of SLC is that when a lossless compression yields a compressed size with a few bytes above MAG, we use lossy compression to approximate these few bytes such that the compressed size is a multiple of MAG. This way, we selectively introduce a small approximation error, however, we significantly increase the compression ratio. Fortunately, there are several GPU applications that are inherently resilient to small error [7], [8].

Considering that E^{2}MC provides the highest compression ratio, we choose E^{2}MC as the baseline lossless compression for SLC. However, SLC is not limited to E^{2}MC but can also be applied to other techniques. A key challenge of SLC is to find the number of symbols needed to be approximated to decrease the compressed size to a multiple of MAG. We use a tree structure to select the required symbols and call this technique as *Tree-based SLC* (TSLC). For a lossy threshold of 16B, SLC provides a speedup of up to 17% with < 1% average error. In summary, we make the following contributions:

- We quantitatively show that low effective compression ratio due to MAG exists in four state-of-the-art techniques and qualitatively in three more.
- We propose a novel MAG aware Selective Lossy Compression technique for GPUs. We show a significant performance gain with a minimal loss in accuracy.
- This is the first study that highlights the importance of

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 688759.

MAG aware compression by quantitatively studying the distribution of compressed blocks above MAG.
- We implement hardware and show the area and power cost of SLC is only 0.0015% and 0.0008% of GTX580.

This paper is organized as follows. In Section II we further motivate the problem. In Section III we present SLC in detail. Section IV explains the experimental setup and results are presented in Section V. Section VI describes related work and finally, we draw conclusions in Section VII.

II. Motivation

A. Qualitative Analysis of More Compression Techniques

Figure 1 quantitatively shows that four state-of-the-art memory compression techniques suffer due to MAG. There are three other techniques: SC^2 [9], HyComp [10] and BPC [3] that can also be applied for memory compression. SC^2 [9] is a statistical cache compression technique and is similar to E^2MC [6] because both are based on Huffman encoding. The former is proposed for CPUs, while the later is proposed for GPUs. Therefore, SC^2 will suffer due to MAG. HyComp is a hybrid compression method which improves the compression ratio by selecting a suitable compression method based on the specific data-type. HyComp will also suffer from MAG as two (BDI and SC^2) out of the four compression methods that HyComp selectively uses are already shown to suffer. The third method called FP-H divides a floating-point number into three fields and then employs SC^2, that means FP-H will also suffer. BPC stands for bit plane compression that uses transformation to increase the compressibility and then uses either run-length or frequent pattern encoding to compress the transformed data. While transformation increases compressibility, BPC will still suffer from MAG as both the run length and frequent pattern encodings exploit patterns similar to FPC and C-PACK which are already shown to suffer in Figure 1. Therefore, several memory compression techniques suffer from MAG.

B. Distribution of Compressed Blocks at MAG

Figure 2 shows the heat map plot of the distribution of compressed blocks at MAG using E^2MC [6] for different benchmarks (Details in Section IV-B). We assume a MAG of 32B and a block size of 128B, which are typical values in current GPUs. The x-axis shows the number of bytes above a multiple of MAG. 0B on the x-axis means a compressed block size is a multiple of MAG i.e. 32B, 64B, or 96B. For simplification, all blocks with a compressed size $<$ 32B are also included in the 0B origin. 32B on the x-axis represents the percentage of uncompressed blocks. The left y-axis shows the percentage of blocks and the right y-axis shows the number of samples. The number of samples shows the number of times a certain percentage of blocks e.g. 20% are compressed with a certain number of bytes e.g. 4B above a multiple of MAG for all benchmarks. Ideally, for a high effective compression ratio, all blocks should be compressed to 0B above a multiple of MAG. However, we see that there is a significant percentage of blocks that are not compressed to an exact multiple of MAG, but a few bytes above a multiple of MAG. As explained

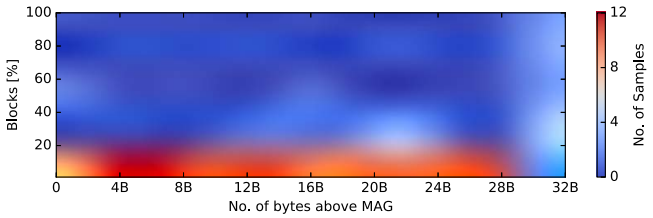


Fig. 2: Heat map showing distribution of compressed blocks.

before, there is no way to just fetch these extra bytes, but we have to fetch a whole 32B burst, causing a low effective compression ratio. Nevertheless, these few extra bytes present an opportunity to achieve a higher effective compression ratio at low accuracy loss by selective approximation.

III. Selective Lossy Compression

A. Overview of a System with SLC Components

Figure 3 shows an overview of a system with the main components of SLC. The compressor, decompressor, and metadata cache (MDC) are integrated into the memory controller (MC). The memory controller needs to fetch only the required number of bursts for every compressed block to save bandwidth. As the number of bursts varies from 1 to 4, we store 2 bits in MDC similar to previous work [2], [6]. Data transfer to and from DRAM is in compressed form with (de)compression taking place in the MC. The data is stored in compressed format in the DRAM, however, the goal is not to increase the effective capacity but to increase the effective off-chip memory bandwidth similar to [2], [6]. Hence, a compressed block is still allocated the same size, although it may require less space. Moreover, we decompress the required number of symbols to recover the original block and the extra data that is fetched due to MAG is meaningless and not interpreted.

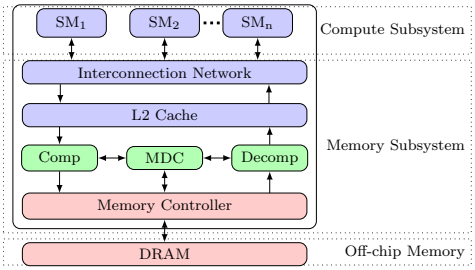


Fig. 3: Overview of a system with compression components.

B. SLC Architecture

Figure 4 shows an overview of the SLC technique. Basically, SLC is a budget-based compression technique which allows selection between different compression modes depending upon *comp size*, *bit budget*, *extra bits*, and a *threshold*. The *bit budget* is a multiple of MAG i.e. 32B, 64B, 96B, or 128B. When the *comp size* of a block is more than its uncompressed size, the block is always stored uncompressed and the *bit budget* is 128B. Since it is not possible to fetch less than 32B from memory, we also use lossless compression when the *comp size* is less than 32B and in this case, the *bit budget*

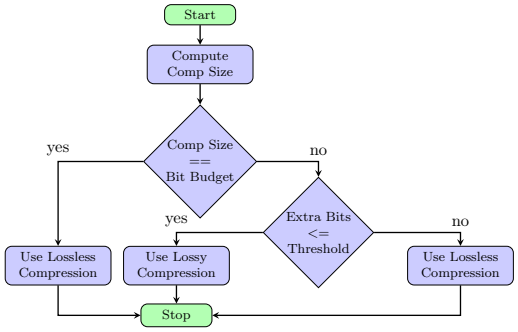


Fig. 4: Overview of selective lossy compression.

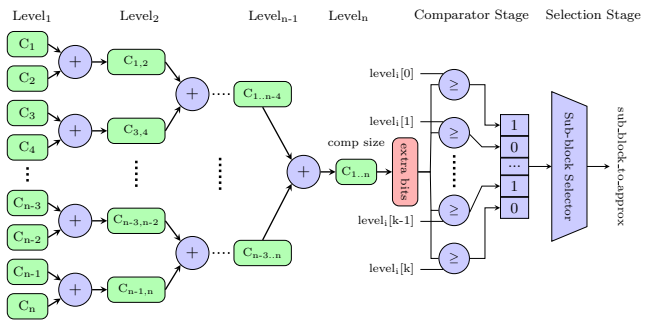


Fig. 5: Tree-based SLC.

is also 32B. The *extra bits* are the number of bits above the *bit budget* and the *threshold* is the number of bits defined by a user that can be safely approximated.

Once we know the *comp size*, we check if it is equal to *bit budget*. We use lossless compression when the *comp size* is equal to *bit budget*. When the *comp size* is not equal to *bit budget*, we use lossy compression if the *extra bits* $\leqslant$ *threshold* and lossless compression if the *extra bits* $>$ *threshold*. Thus, SLC mostly retains the quality of a lossless compression and smartly trades small accuracy to achieve the desired compression and higher performance.

We know how many bytes (*extra bits*) are above a MAG, but the problem is that these *extra bits* are codewords and not symbols. The challenge here is to find the number of symbols that need to be approximated to decrease the compressed size by *extra bits*. We determine the number of symbols needed to be approximated using a tree structure and then only approximate these symbols. We call this technique Tree-based SLC (TSLC). We first describe how we compute *comp size*, *bit budget*, *extra bits*, and *threshold* in Section III-C and then explain TSLC in Section III-D.

C. Compressed Block Size, Bit Budget, and Extra Bits

To use SLC, the first thing that we require is the compressed block size (*comp size*) that would result if only lossless compression is used. However, we cannot wait for a lossless compression to compress a block and then decide which compression mode to use as compression incurs long latency. Although GPUs can hide compression latency, too much increase can also degrade their performance [6]. Fortunately, we only need the *comp size* and not the compressed block to choose a compression mode and the *comp size* can be easily calculated by just adding all code lengths [6]. RTL synthesis details to obtain the *comp size* are described in Section IV-A.

Once the *comp size* is known, *bit budget* of a block can be computed. The *bit budget* is the closest multiple of MAG $\leqslant$ *comp size*. The possible values of bit budget are 32B, 64B, 96B, or 128B. The *extra bits* are simply calculated by subtracting the *bit budget* from *comp size*.

D. Tree-based SLC

We use a parallel tree adder to add all code lengths of a block and small additional logic to select the symbols for approximation as shown in Figure 5. The last node of the tree contains the *comp size* that is used to find the *bit budget* and *extra bits*. We use intermediate sums of the code lengths to select the symbols for approximation as explained below.

When the lossy compression mode is selected, the *extra bits* are compared with the intermediate sums at all levels in parallel as shown in Figure 5. The output of comparisons is written to a bit vector. It may happen that we do not find any sub-block with compressed size (intermediate sum) $\geq$ *extra bits* at some levels. The output of the comparison stage is all zeros for these levels. In the sub-block selection stage, priority encoders are used to output the indices of the first sub-block with compressed size $\geq$ *extra bits* for each level of the tree. Finally, a sub-block ($sub\_block\_to\_approx$) with compressed size $\geq$ *extra bits* from the lowest level ($approx\_level$) is selected for approximation as at this level we need to approximate the fewest symbols. As the $sub\_block\_to\_approx$ is selected in parallel, the latency is fixed regardless of the approximated level. The latency overhead is described in Section IV. Once $sub\_block\_to\_approx$ is selected, the start symbol for approximation is obtained by: $sub\_block\_to\_approx \times 2^{approx\_level}$.

E. Value Similarity-based Prediction

In TSLC, we simply truncate selected symbols during compression that guarantees the desired compression, however, truncation may cause a high error. Considering that we only need to predict few symbols and adjacent threads in GPUs have high-value similarity [7], [11], we decided to use a simple value similarity-based prediction to reduce the error. While decoding an approximated block, we use the value of the first non-truncated symbol of the block as the predicted value for truncated symbols. In terms of decompressor hardware change, we only need to generate the index of the predicted value. While there are exact value predictors [12], [13] with trade-offs in terms of accuracy, complexity, and storage, we opt for the very simple prediction scheme due to its negligible hardware cost and reasonable accuracy for our use case.

F. TSLC Optimization

TSLC may approximate significantly more bits than needed due to coarse intermediate sums at middle levels. This can happen as a node at level *l+1* has a sum of $2\times$ the nodes at level *l* as shown in Figure 5 and when we cannot find a sub-block with compressed size $\geq$ *extra bits* at level *l*, we move to *l+1* and it may be the case that the largest sub-block at level

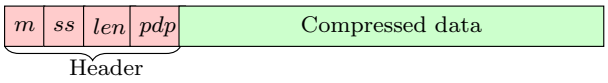


Fig. 6: Structure of a compressed block.

l is only a few bits less than the *extra bits*. The experiments show that a significant unneeded approximation may happen at the middle levels (3 and 4). The high unneeded approximation does not happen at lower levels (< 3) as the intermediate sums are smaller and it also does not occur at higher levels (> 4) because we can mostly find a sub-block to approximate at the middle levels. To reduce the unneeded approximation, we further optimize TSLC by adding a few extra nodes at middle levels. We add 8 and 4 extra nodes to have less coarse sums at levels 3 and 4, respectively, which originally have 16 and 8 nodes. We can further optimize by having even fine-grained sums, however, that will require more hardware resources.

G. Structure of a Compressed Block

Figure 6 shows the structure of a compressed block which consists of a header and compressed data. The header is needed to decompress a block. The header consists of 1-bit (m) to indicate the compression mode (lossless or lossy), 6-bit to store the index (ss) of the first approximated symbol, 4-bit to store the number of approximated symbols (len), and 3 parallel decoding pointers (pdp) for 4 parallel decoding ways (PDWs). Experiments show that the maximum number of approximated symbols is 16, thus we need 4-bit to store the len. Each pdp consists of N bits, where 2^N is the block size in bytes. No header is needed for uncompressed block as in the baseline [6].

TABLE I: Frequency, area, and power of SLC.

Compressor			Decompresor		
Freq (GHz)	Area (mm^2)	Power (mW)	Freq (GHz)	Area (mm^2)	Power (mW)
1.43	0.00830	1.620	0.80	0.00030	0.210

H. Hardware Implementation and Overhead of SLC

To estimate the frequency, area and power overhead of SLC, we wrote RTL and synthesized the design using Synopsis design compiler version K-2015.06-SP4 at 32 nm node. Table I shows the frequency and additional hardware overhead of extending E^{2}MC with TSLC. We only present synthesis results for the optimized TSLC with prediction as the differences are insignificant. The area and power overhead of TSLC is only 0.0015% and 0.0008% of GTX580. Moreover, TSLC only adds 5.6% of the area of E^{2}MC. Therefore, in terms of hardware overhead, SLC is feasible and very cheap.

IV. Experimental Methodology

A. Simulator

We modify gpgpu-sim [14] to integrate BDI, FPC, C-PACK, E^{2}MC, and SLC and configure it to simulate a GPU similar to GTX580. Table II shows the configuration summary. For the baseline lossless compression E^{2}MC, we use 16-bit symbols, 4 PDWs and an online sampling size of 20 million instructions as they provide the best results [6]. It takes 46 cycles to compress and 20 cycles to decompress a block by E^{2}MC. For a block size of 128B and 16-bit symbols, a total of 64 code lengths need to be read from the compressor table and added to know the compressed block size. RTL synthesis shows that all code lengths can be fetched in about 12 cycles at 1002 MHz and it requires another 2 cycles to add them and select a sub-block for approximation. Thus, TSLC needs 60 cycles to compress a block. Due to very simple additional decompression logic, a block in TSLC can be decompressed in the same number of cycles as in E^{2}MC. For estimating the energy consumption, GPUSimPow [15] is modified with the RTL synthesis based power models of E^{2}MC and TSLC.

TABLE II: Baseline simulator configuration.

#SMs	16	L1 $ size/SM	16 KB
SM freq (MHz)	822	L2 $ size	768 KB
Max #Threads/SM	1536	#Registers/SM	32 K
Max CTA size	512	Shared memory/SM	48 KB
Memory type	GDDR5	# Memory controllers	6
Memory clock	1002 MHz	Memory bandwidth	192.4 GB/s
Bus width	32-bit	Burst length	8

B. Benchmarks

We include memory-bound and amenable to approximation benchmarks shown in Table III. We use speedup and application specific error metrics to trade-off performance and accuracy in accord with [8], [11], [16]. We use mean relative error (MRE) for applications which produce numeric outputs and Normalized Root Mean Square Error (NRMSE) which process images or belong to a signal processing domain. JM finds the intersection between triangles and we use miss rate to report the fraction of incorrect decisions.

TABLE III: Benchmarks used for experimental evaluation.

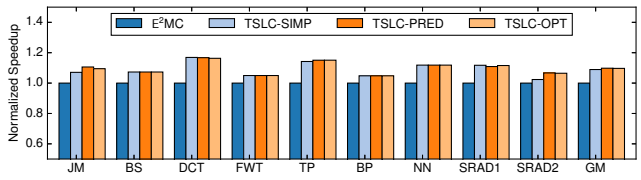
Name	Short Description	Input	Error Metric	#AR
JM	Intersection of tri. [17]	400 K tri. pairs	Miss rate	6
BS	Options pricing [18]	4 M options	MRE	4
DCT	Discrete trans. [18]	1024×1024 img.	Image diff.	2
FWT	Fast walsh trans. [18]	8 M elements	NRMSE	2
TP	Matrix transpose [18]	1024×1024	NRMSE	2
BP	Perceptron train. [19]	64 K elements	MRE	6
NN	Nearest neighbors [19]	20 M records	MRE	2
SRAD1	Anisotropic diff. [19]	1024×1024 img.	Image diff.	8
SRAD2	Anisotropic diff. [19]	1024×1024 img.	Image diff.	6

C. Safe to Approximate Loads and Approximation Threshold

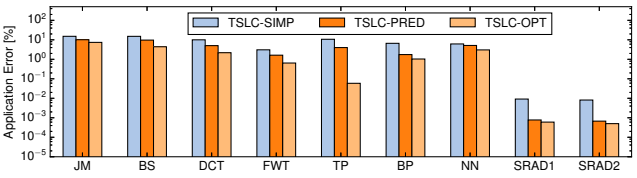
The research has shown that safety is a semantic property of a program [20] and to identify a safe-to-approximate load or region, it requires programming language support. Therefore, it is a common practice that a programmer annotates the loads or code regions [2], [11], [16]. Similar to previous work, SLC also requires annotations, however, instead of burdening a programmer with the task of identifying individual loads, we opt for a model that is much easier to use and cheaper to implement. In our model, a programmer specifies if a memory region is safe to approximate using an extended cudaMalloc() as shown below.

```
cudaMalloc(void** devPtr, size_t size,
           bool safeToApprox, size_t threshold)
```

The address returned by the extended cudaMalloc() and size of the memory allocation is used to determine if a load is safe to approximate or not. We implement extended cudaMalloc() in

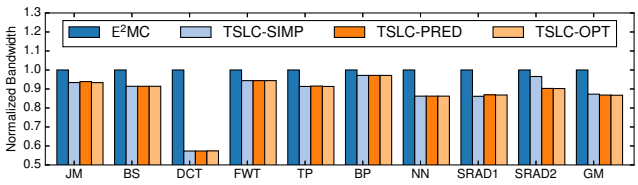


(a) Speedup

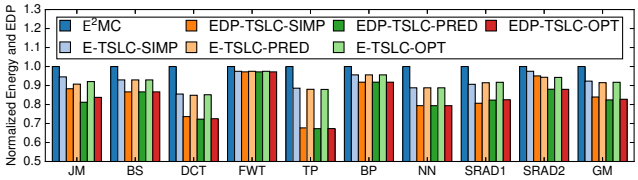


(b) Error

Fig. 7: Speedup and error for SLC.



(a) Bandwidth reduction



(b) Energy and EDP reduction

Fig. 8: Bandwidth, energy, and EDP reduction for SLC.

gpgpu-sim. Table III lists the number of approximated memory regions (AR). We exclude memory regions which can cause a catastrophic failure such as segmentation fault. Likewise, a programmer needs to specify a lossy threshold that satisfies the target output quality and maximizes the benefits.

V. Experimental Results

We present results for three variations of TSLC using a lossy threshold of 16B and 32B MAG. The first variation is called TSLC-SIMP, where we simply truncate the selected symbols during compression and replace them with zero values during decompression. The second variation is called TSLC-PRED, where we use value similarity-based prediction during decompression. The third variation is called TSLC-OPT, where in addition to prediction, we further optimize TSLC-PRED by adding a few extra nodes.

A. Speedup

Figure 7 shows the speedup and error for TSLC normalized to E^{2}MC [6]. Figure 7a shows that all three variations of TSLC provide significant speedups compared to E^{2}MC. The maximum speedup is about 17% for *DCT* and minimum speedup is about 5% for *FWT* and *BP*. The geometric mean (GM) of the speedups is 9%, 9.8%, 9.7% for TSLC-SIMP, TSLC-PRED, and TSLC-OPT, respectively. There is only a slight variation in the average speedup of the three schemes which is expected because all of them roughly approximate the same number of blocks by the same amount. However, a slight variation in speedup occurs due to the differences in the way the three schemes perform (de)compression. All the three schemes truncate during compression, however, TSLC-OPT may truncate slightly less number of symbols. During decompression, TSLC-SIMP uses zeros for approximated symbols, TSLC-PRED and TSLC-OPT employ prediction, but the number of predicted symbols may differ. Thus, due to the above-mentioned differences, a decompressed block may differ from one scheme to another. Because of the differences in the decompressed blocks, their further compressibility and the blocks which depend on them may differ and hence we see slight variations in the speedups of the three schemes.

TSLC-SIMP has the highest error due to truncation. The error reduces significantly for TSLC-PRED which shows that the value similarity-based prediction works quite well, commensurating with previous work [11]. The error reduces further for TSLC-OPT because it reduces the unneeded approximation by adding a few extra nodes. The error is $< 3\%$ except for *JM* (7.3%) and *BS* (4.4%) when TSLC-OPT is used and for TP (0.05%), SRAD1 (0.001%), SRAD2 (0.001%) is extremely low. The reason for slightly high error (miss rate) for JM is that it finds an intersection between two triangles and outputs a boolean that may flip due to approximation. In general, the error is much lower than 10% that is treated as an acceptable error in many related works [8], [11]. In addition to application-specific error, we also calculated the mean relative error for individual benchmark which enables averaging the error across all benchmarks. For TSLC-OPT, the geometric mean of the mean relative error for all benchmarks is 0.99%.

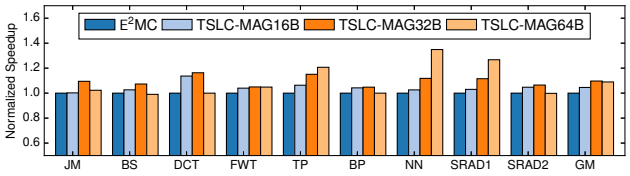
B. Bandwidth and Energy Reduction

Figure 8 shows the reduction in off-chip memory bandwidth, energy, and energy-delay-product (EDP) normalized to E^{2}MC. The reduction results from the increased effective compression ratio due to SLC. The geometric mean (GM) of the reduction in memory bandwidth is about 14% for all three variations of TSLC. The reduction in bandwidth, which is a reciprocal of the compression ratio, results from saving 32B bursts. The results show a relatively higher decrease in bandwidth for some benchmarks such as *DCT*, however, there is not that much increase in speedup. This is because some loads can be more performance critical than others [11].

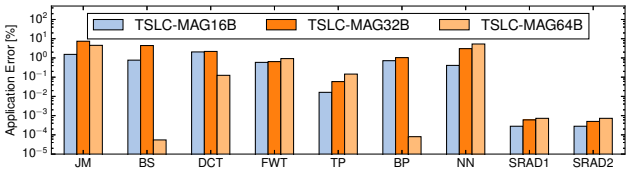
The GM of the energy and EDP reduction is about 8.3% and 17.5%. SLC reduces the energy consumption by reducing the off-chip memory bandwidth and execution time.

C. SLC Sensitivity to MAG

The geometric mean (GM) of the effective compression ratio is 1.41, 1.31, 1.16 for E^{2}MC with MAG of 16B, 32B, and 64B, respectively. The effective compression ratio decreases with the increase in MAG because the number of points decreases where a block can have compression ratio > 1.0.



(a) Speedup



(b) Error

Fig. 9: Speedup and error for SLC with different MAGs.

The GM of the raw compression ratio (1.54) is the same for different MAGs because it does not depend on MAG.

Figure 9 shows the speedup and error for TSLC-OPT with MAG of 16B, 32B, and 64B. We set the lossy threshold to half of the corresponding MAG for this experiment as one threshold across different MAGs is not suitable. For example, a lossy threshold of 16B for MAG of 16B will always approximate blocks to a lower multiple of a MAG. Moreover, a 16B threshold might be small for a 64B MAG.

The GM of the speedup is 5%, 9.7%, and 9% for MAG of 16B, 32B, and 64B, respectively. There are a few interesting observations. First, we observe that SLC provides speedup across different MAGs. Second, we observe large variations in the speedup for 64B MAG compared to 16B and 32B. We have a high speedup of 35% for *NN*, 27% for *SRAD1*, 21% for *TP*, while there is no speedup for *BS, DCT, BP* for 64B MAG. This is mainly because there is only one point ($\leq$ 64B) where the effective compression ratio can be $>$ 1.0 for MAG of 64B, while there are multiple such points for MAG of 32B and 16B. For example, for *NN, SRAD1, TP*, the losslessly compressed size of most of the blocks is above 64B and SLC exploits that to increase the performance gain. In contrast, the compressed size of most of the blocks is already below 64B for *DCT, BP* and above 96B for *BS*, thus, there is no opportunity for SLC to improve the performance further. Like speedup, we have higher variations in the error for 64B MAG, for example, relatively higher error for *NN* (5.2%) and *TP* (0.14%).

VI. Related Work

To the best of our knowledge, there is no work directly related to MAG aware approximation, however, approximation has been used for GPUs at software, hardware, and hybrid levels [7], [11], [16]. Samadi et al. [7] used static compilers to automatically generate different versions of GPU kernels with different aggressiveness and depending on the target quality, a run-time system selects the appropriate version. NPU [8] uses program transformation to select and train a neural network to mimic a region of imperative code. Yazdanbakhsh et al. [11] proposed rollback-free value prediction scheme to reduce long memory latency and memory bandwidth. Sathish et al. [2] proposed lossless and lossy memory I/O link compression. Warp Approximation [16] executes only a single representative thread and uses its value to approximate intra-warp values.

VII. Conclusions

Memory compression techniques often exhibit low effective compression ratio due to large MAG. We studied the distribution of compressed blocks and observed that a significant percentage of blocks are compressed to a size that is only a few bytes above a multiple of MAG, but a whole burst is fetched from memory, leading to low effective compression ratio. We proposed a novel MAG aware Selective Lossy Compression (SLC) technique for GPUs. SLC appropriately selects between lossless and lossy compression modes, mostly retains the quality of a lossless compression and intelligently trades small accuracy for higher performance. SLC provides a speedup of up to 35% and 17% normalized to a state-of-the-art lossless compression technique for 64B MAG and 32B MAG, respectively. For a lossy threshold of 16B, the average loss in accuracy is $<$ 1% and average energy consumption and EDP are reduced by 8.3% and 17.5%, respectively. We also implemented hardware and showed that the area and power overhead of SLC is feasible and cheap.

References

[1] Chen *et al.*, "C-Pack: A High-Performance Microprocessor Cache Compression Algorithm," *IEEE Transactions on VLSI Systems*, 2010.

[2] Sathish *et al.*, "Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads," in *Proc. 21st PACT'12*.

[3] Kim *et al.*, "Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures," in *Proc. 43rd ISCA*, 2016.

[4] Pekhimenko *et al.*, "Base-Delta-Immediate Compression: Practical Data Compression for On-chip Caches," in *Proc. 21st PACT*, 2012.

[5] Alameldeen *et al.*, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," in *Report, UWM*, 2014.

[6] Lal *et al.*, "E^{2}MC: Entropy Encoding Based Memory Compression for GPUs," in *Proc. 31st IPDPS*, 2017.

[7] Samadi *et al.*, "SAGE: Self-tuning Approximation for Graphics Engines," in *Proc. 46th MICRO*, 2013.

[8] Esmaeilzadeh *et al.*, "Neural Acceleration for General-Purpose Approximate Programs," in *Proc. 45th MICRO*, 2012.

[9] Arelakis *et al.*, "SC^2: A Statistical Compression Cache Scheme," in *Proc. 41st ISCA*, 2014.

[10] Arelakis *et al.*, "HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods," in *Proc. 48th MICRO*, 2015.

[11] Yazdanbakhsh *et al.*, "RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads," *ACM TACO*, 2016.

[12] Perais *et al.*, "Practical Data Value Speculation for Future High-end Processors," in *Proc. 20th HPCA*, 2014.

[13] Sazeides *et al.*, "The Predictability of Data Values," in *Proc. 30th MICRO*, 1997.

[14] Bakhoda *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. ISPASS*, 2009.

[15] Lucas *et al.*, "Why a Single Chip Causes Massive Power Bills - GPUSimPow: A GPGPU Power Simulator," in *Proc. ISPASS*, 2013.

[16] Wong *et al.*, "Approximating Warps with Intra-warp Operand Value Similarity," in *Proc. HPCA*, 2016.

[17] Yazdanbakhsh *et al.*, "AxBench: A Multiplatform Benchmark Suite for Approximate Computing," *IEEE Design & Test*, 2017.

[18] NVIDIA, "CUDA," http://developer.nvidia.com/object/gpucomputing.html.

[19] Shuai *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. IISWC*, 2009.

[20] Park *et al.*, "FlexJava: Language Support for Safe and Modular Approximate Programming," in *Proc. 10th ESEC/FSE*, 2015.