Memory Access Granularity Aware Lossless Compression for GPUs

Sohan lal
Technische Universität Hamburg, Germany
sohan.lal@tuhh.de

Manuel Renz, Julian Hartmer, Ben Juurlink
Technische Universität Berlin, Germany
m.renz.1@campus.tu-berlin.de, {j.hartmer, b.juurlink}@tu-berlin.de

***Abstract*—High-bandwidth off-chip memory has played a key role in the success of Graphics Processing Units (GPUs) as an accelerator. However, as memory bandwidth scaling continues to lag behind the computational power, it remains a key bottleneck in computing systems. While memory compression has shown immense potential to increase the effective memory bandwidth by compressed data transfers between on-chip and off-chip memory, the large memory access granularity (MAG) of off-chip memory limits compression techniques from achieving a high effective compression ratio. Unfortunately, state-of-the-art lossless memory compression techniques do not take the large MAG of off-chip memory into account. A recent study has used MAG-aware approximation to increase the effective compression ratio, however, not all applications can tolerate errors, which limits its applicability. We propose extensions and GPU-specific optimizations to adapt a lossless memory compression technique to a MAG size to increase the effective compression ratio and performance gain. Our technique is based on the well-known Base-Delta-Immediate (BDI) compression technique that compresses a memory block to a common base and multiple deltas. We leverage the key observation that deltas often contain enough leading zeros to compress a block to a multiple of MAG without any loss of information. We show that MAG-aware BDI provides, on average, 48% higher effective compression ratio, 10% (up to 27%) higher speedup, and 16% bandwidth reduction compared to normal BDI. While BDI, FPC, and CPACK have a similar compression ratio, MAG-aware BDI outperforms FPC, CPACK, and SLC by 56%, 47%, and 33%, respectively.**

***Index Terms*—GPUs, compression, memory access granularity**

I. Introduction

The computational power of GPUs has rapidly scaled over the last years, almost 28× from the first general-purpose capable GPU, NVIDIA GTX-8800 (Tesla architecture), to NVIDIA Titan V (Volta architecture) [1], [2]. While the off-chip memory bandwidth has also scaled, it is far lagging with respect to computational scaling. Moreover, with the ever-increasing data that need high-capacity storage and fast processing, the pressure on memory bandwidth will be even higher in the future. On the one hand, the large data requires higher memory bandwidth and storage capacity, on the other hand, the large data has redundancy that can be exploited by compression techniques to achieve a higher compression ratio. Therefore, memory compression is being increasingly used to increase the memory bandwidth and/or storage capacity.

While memory compression plays a significant role in meeting the demands of higher memory bandwidth and capacity, memory compression techniques often result in a low effective compression ratio due to the large memory access granularity (MAG) of GDDR/DDR/HBM memories employed in multi-/many-core systems [3]. MAG is the amount of data transferred from/to a memory by a single read or write command. A typical size of MAG is 32 bytes, resulting from 32-bit bus width and a burst length of 8 for GDDR/DDR memories and 128-bit bus width and a burst length of 2 for HBM2. Figure 1 illustrates how a large MAG size leads to a low effective compression ratio. For the compressed size of 36 bytes, we fetch 64 bytes due to the MAG size of 32 bytes. Thus, a compression ratio that seems close to 3.6× is actually only 2×. The large memory access granularity helps to achieve peak memory bandwidth and reduce control overhead, however, it can cause a significant loss (on average about 20%) to the effective compression ratio [3], [4]. The loss occurs because a compressed block can be of any size and not necessarily a multiple of a MAG, while a memory controller only fetches data in the multiple of a MAG. This means a memory controller needs to fetch a whole burst even when only a few bytes are actually required. There can be a significant fraction of blocks that have a few bytes above MAG, leading to a low effective compression ratio. As a result of a large MAG, several state-of-the-art memory compression techniques such as FPC [5], BDI [6], C-PACK [7], E^{2}MC [4], HyComp [8] have a low effective compression ratio [3].

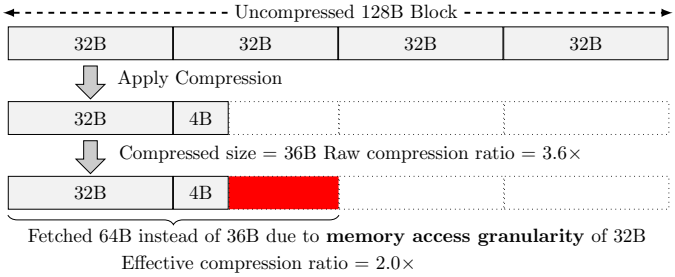


Fig. 1: An example illustrating how a large MAG size (32B) may reduce the effective compression ratio significantly.

Lal et al. [3], [9] propose MAG-aware selective lossy compression (SLC) to approximate the extra bytes above MAG to increase the effective compression ratio. While the SLC technique is effective for error-resilient applications, many applications cannot tolerate any error. Therefore, SLC usability is limited. While a lossless MAG-aware memory compression technique is desirable, we show that it is not possible to

adapt many existing techniques to a MAG size due to the way they perform compression. Our analysis shows that only Base-Delta-Immediate (BDI) [6] compression technique can be adapted to MAG size. The main difference between the BDI and other cache compression techniques such as FPC [5], C-PACK [7], E^{2}MC [4] is that whereas BDI compresses each word of a cache line with the same encoding, the other compression techniques compress each word separately. The same encoding in BDI leads to all potential compressed sizes to be known statically, while encoding each word differently leads to a dynamic compressed size. As the potential compressed sizes are known statically, we propose extensions and GPU-specific optimizations to adapt BDI to be MAG aware.

While BDI is a well-known compression technique with several desired characteristics such as low latency, decent compression ratio, unfortunately, it also suffers from the large MAG. BDI leverages the low dynamic range of values in a block by using a common base and an array of differences as deltas. We analyze the deltas and observe that deltas often contain a few leading zeros that can be leveraged to compress a block to an exact multiple of MAG without any loss of information. MAG-aware BDI provides 48% higher effective compression ratio and 10% (up to 27%) speedup on average.

In summary, we make the following main contributions:

- We qualitatively analyze the feasibility of MAG-aware adaption of several state-of-the-art lossless compression techniques and show that only BDI can be adapted.
- We analyze BDI-compressed blocks and observe that 46% of the compressed blocks have deltas with a few leading zeros, presenting an opportunity to further shorten the deltas by a few bits.
- We propose MAG-aware BDI that leverages the leading zeros in the deltas to compress a block to a multiple of MAG without any loss of information. We further show opportunities for GPU-specific optimizations.
- We show that MAG-aware BDI improves the effective compression ratio by 48%, speedup by 10% (up to 27%), and bandwidth by 16% compared to BDI compression. MAG-aware BDI not only outperforms state-of-the-art techniques by up to 56% higher compression ratio, several state-of-the-art techniques also benefit from the adaption of BDI to a MAG size.
- We further show that MAG-aware BDI provides performance gain for different MAG sizes.

This paper is organized as follows. In Section II, we discuss the related work. In Section III, we provide the necessary background on BDI. In Section IV, we explain MAG-aware BDI. The experimental methodology and experimental results are presented in Section V and Section VI, respectively. Finally, we draw conclusions in Section VII.

II. Related Work

Recent research has shown that compression can be used for general-purpose GPU workloads [5], [10]–[13]. Sathish et al. [10] use Cache-Packer (C-PACK) [7], a dictionary-based compression technique to compress data transferred through memory I/O links. Lee et al. [11] use Base-Delta-Immediate (BDI) compression [6] to compress the register file. Vijaykumar et al. [14] use underutilized resources to create assist warps that can be used to accelerate the work of regular warps. To illustrate a use case, authors utilize assist warps for compression using BDI. Kim et al. [13] propose Bit-plane Compression (BPC) that first uses transformations to increase the compressibility and then uses either run-length or Frequent Pattern Compression (FPC) [5] to compress the transformed data. While BDI, FPC, C-PACK techniques can decompress with a few cycles, their compression ratio is typically between 1.5 - 2.0×. Compared to these techniques, E^{2}MC [4] and Statistical Cache Compression (SC^2) [15] have a higher latency but also a higher compression ratio. Arunkumar et al. [16] propose LATTE-CC, a compression management scheme that chooses between no compression, low-latency compression (BDI), and high-capacity compression (SC^2) based on predicted latency tolerance of a cache line. Tavana et al. [17] propose an adaptive compression scheme that uses either FPC, CPACK, or BDI to reduce inter-GPU traffic.

A problem with lossless memory compression techniques is that they have a low effective compression ratio due to the large memory access granularity (MAG) [3], [4]. To address the issue, Lal et al. [3], [9], [18] propose a MAG-aware selective lossy compression (SLC) technique, which approximates bytes above the MAG to increase the effective compression ratio. While the SLC technique is effective for error-resilient applications, it is infeasible for many applications. We propose a MAG-aware lossless memory compression technique. Our technique is based on the well-known BDI compression and leverages the leading zeros in deltas to compress a block to an exact multiple of MAG without any loss of information. To the best of our knowledge, no prior technique exists that implements MAG-aware lossless compression for GPUs. BPC [13] is evaluated using a Hybrid Memory Cube (HMC) that uses packets based memory accesses, unlike DDR. On the one hand, packets provide a small 16-byte MAG, on the other hand, they have header and tail overhead which limits the benefits of packets based memory. Moreover, a higher effective compression that BPC gains from a small MAG of HMC does not apply to DDR/HBM type memories which are more common. Furthermore, 16-byte MAG increases the effective compression ratio compared to 32-byte MAG but does not solve the problem fully (see Section VI).

For CPUs, there is more work on cache compression [8], [19] and increasing the main memory capacity [20]. Yang et al. [19] develop a compression scheme for an L1 cache that exploits frequently accessed values. Arelakis et al. [8] propose hybrid cache compression (HyComp) technique for CPUs that selects a suitable compression method based on the data type. The main challenge of using compression to increase capacity is to find the starting address of a compressed block in the main memory. Pekhimenko et al. [20] propose a framework called linearly compressed pages to compute the starting address. Hong et al. [21] propose a framework called Attaché that mitigates metadata accesses to off-chip memory.

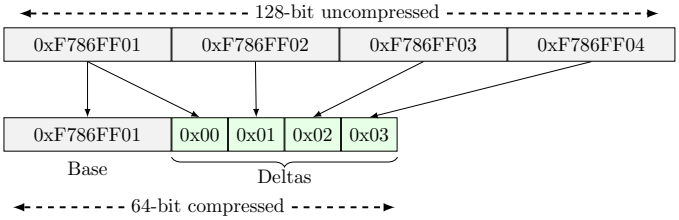


Fig. 2: An example of BDI technique with one base.

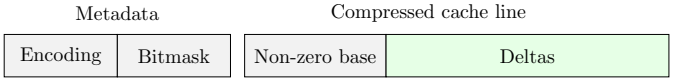


Fig. 3: Structure of a BDI-compressed cache line.

III. Background

Base-Delta-Immediate (BDI) [6] compression is based on the observation that values in a cache line have a high-value similarity and low dynamic range i.e., the relative difference between the values is small. A cache line is represented using a common base and small differences as deltas. Instead of a single base, multiple bases can be used to increase the compression ratio. Using multiple base values, BDI compression scheme can compress data with different dynamic ranges. The authors show that two base values work the best.

Figure 2 shows a simple example of BDI compression using a single base. Although, the uncompressed values are large, the relative difference is very small. The first value is chosen as the base and the differences are stored as deltas. As deltas require less space, we can achieve significant compression. Determining the optimal number of bases is complicated. A typical implementation of BDI uses two base values: an implicit zero base and a non-zero value from the input, usually the first value that cannot be compressed with the zero base. The simple selection of the non-zero base decreases the compression ratio only marginally, while reducing the hardware complexity [6]. The zero-base helps to compress the narrow values, while the non-zero base can compress the wide values. This simple selection of base values allows parallel decompression by just adding deltas to base values, resulting in a very low decompression latency. The use of implicit zero as the second base also means that it does not require additional storage overhead. However, the compression phase needs to store a bitmask, one bit per value of a cache line, to indicate which of the two bases is used. The delta size is the same for all deltas in a cache line. BDI compression uses different base values (8, 4, and 2 bytes) and different delta sizes (4, 2, and 1 byte) for compression. A cache line is compressed in parallel with all combinations of bases and deltas and the smallest possible compressed size is chosen.

Figure 3 shows the structure of a BDI-compressed cache line, which consists of metadata and the compressed cache line. The compressed cache line stores a non-zero base and deltas. The metadata consists of the encoding bits and bitmask, which help to decompress the cache line. The encoding bits determine the base and delta size used for compression. The bitmask distinguishes between the implicit zero and non-zero base. BDI stores the encoding bits in a tag and uses them to know the number of segments used for a cache line. We store the encoding bits in the metadata cache and use them to fetch the required number of memory bursts from the off-chip memory as explained in Section IV-F.

TABLE I: Qualitative analysis of state-of-the-art lossless compression techniques for MAG-aware adaption.

Technique	Granularity level	Compressed words	Compressed size	MAG-aware possibility
FPC	Word	Variable codes	Dynamic	No
C-PACK	Word	Variable codes	Dynamic	No
E^2MC	Word	Variable codes	Dynamic	No
SC^2	Word	Variable codes	Dynamic	No
HyComp	Mixed	Mixed	Mixed	Partial
BPC	Word	Variable codes	Dynamic	No
BDI	**Block**	**Fixed delta**	**Static**	**Yes**

IV. MAG-aware Lossless Compression

In this section, we explain our MAG-aware extensions to BDI compression. First, we qualitatively analyze state-of-the-art compression techniques, showing only BDI can be adapted. We then show opportunities to increase the effective compression ratio by leveraging leading zeros in deltas.

A. Qualitative Analysis for MAG-aware Adaption

Lal et al. [3] propose a MAG-aware approximation technique, which selectively approximates bytes above a MAG to increase the effective compression ratio. The approximation is effective for error-resilient applications, however, it cannot be applied to applications that cannot tolerate any errors. While a lossless MAG-aware technique has broader usability, unfortunately, it is not possible to adapt several state-of-the-art lossless memory compression techniques to a MAG size.

Table I shows a summary of the qualitative analysis of several state-of-the-art lossless memory compression techniques, investigating whether they can be made MAG-aware without approximation. The table shows only BDI can be adapted to a MAG size without approximation. What makes BDI adaptable to a MAG size is the granularity at which compression is performed. While BDI performs compression at a block granularity, other techniques such as FPC [5], C-PACK [7], E^2MC [4] use word-level compression. More specifically, in BDI compression, all words of a block are compressed to a fixed size delta, which makes BDI adaptable to a MAG size. Although, there are different delta sizes for different encodings, a block is compressed using only one encoding, which means the same delta size is used for all words of a block. As all delta sizes are known beforehand, all potential compressed block sizes are also known statically. However, the delta sizes used in BDI lead to a compressed size that is not a multiple of a MAG, but by reducing a delta size, a block can be compressed to a multiple of MAG. We need to explore if the delta size can be reduced.

The other techniques listed in Table I compress a cache line at word granularity i.e., all words of a block are compressed separately, potentially with different encodings. For example,

TABLE II: State-of-the-art compression techniques that benefit from MAG-aware adaption of BDI.

Technique	Reason
HyComp	Selects **BDI** or SC^2 based on data type
Latte-CC	Uses **BDI** or SC^2 based on latency tolerance
CABA [14]	Assist warps utilize **BDI** as a use case
Tavana et al. [17]	Uses **BDI**, FPC, or CPACK based on sampling

FPC uses different prefix codewords for each word (frequent-pattern) of a block. Although, the prefixes have fixed length, the encoded patterns can be of different sizes such as 4-, 8-bit sign-extended, or uncompressed word. Moreover, the number of times each pattern can occur in a cache line is not known statically, leading to a compressed size that is known only at run-time. This is unlike BDI where delta size is fixed for all words of a block. C-PACK also exploits fixed frequently occurring static patterns like FPC and uses a dynamic dictionary to adapt to other frequently appearing words. Not only the different patterns in a block can get variable codewords, the frequency of each pattern is also unknown. Therefore, FPC and C-PACK cannot be adapted to a MAG size like BDI. Similarly, E^2MC also uses variable codewords and cannot be adapted to MAG size.

The three other techniques shown in the table: SC^2 [15], HyComp [8] and BPC [13] can also be applied for memory compression. SC^2 [15] is a statistical cache compression technique and is similar to E^2MC [4] because both are based on Huffman encoding. HyComp is a hybrid compression that selects a suitable compression method based on the data-type. As HyComp uses either BDI or SC^2 for compression, the compression granularity is mixed. The blocks compressed using BDI have block granularity, while blocks compressed using SC^2 have word granularity. Therefore, HyComp can be partially adapted to a MAG size and will benefit from our proposal. BPC uses transformations to increase the compressibility and then uses either run-length or frequent pattern encoding to compress the transformed data. While transformations increases compressibility, BPC also cannot be adapted as both the run length and frequent pattern encodings exploit patterns similar to FPC and C-PACK, which are not adaptable.

Due to several desired characteristics of BDI such as low latency, low area overhead, it is an important component of several state-of-the-art compression techniques such as HyComp, Latte-CC. Therefore, adopting BDI to a MAG size also directly benefits these techniques. Table II shows a list of compression techniques that utilize BDI and reasons why adopting BDI to a MAG size also benefits these techniques.

B. Motivation

BDI compression provides a high raw compression ratio but it is not optimized towards the coarse-grained memory access granularity (MAG) exhibited by multi-/many-core systems [3], [4]. The large MAG causes a significant difference between the raw and the effective compression ratio, the latter significantly lower than the former. Figure 4 shows an example of how a large MAG can lead to a low effective compression ratio.

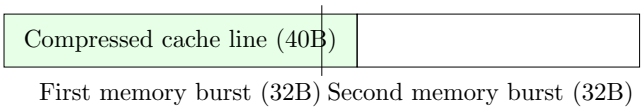


Fig. 4: 128B cache line compressed by BDI using 4-byte base and 1-byte delta (Base4B-Δ1B encoding). The compressed size is 40 bytes including non-zero base and 32-bit bitmask.

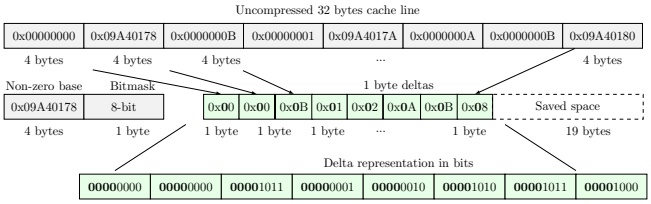


Fig. 5: A representational example of BDI compression with leading zeros. The bold bits are leading zeros.

Assuming a block size of 128 bytes, 4-byte base, and 1-byte deltas (Base4B-Δ1B encoding), BDI will compress the block to 40 bytes, which consists of a 4-byte non-zero base, 32-bit bitmask, and 32 deltas of 1 byte each. With a MAG of 32 bytes, this results in two memory bursts (32 bytes each), fetching a total of 64 bytes instead of 40 bytes from the off-chip memory. While the raw compression ratio is 3.2$\times$ (128/40), the effective compression ratio is only 2$\times$ (128/64). The situation is similar with other encodings, for example, for the 4-byte base and 2-byte deltas (Base4B-Δ2B encoding), the raw compression ratio will be 1.78$\times$ (128/72), while the effective compression ratio will only be 1.33$\times$ (128/96).

The compressed size has 8 bytes above the MAG which requires an extra burst of 32 bytes, causing a low effective compression ratio. These 8 bytes could also be stored in a metadata cache and only deltas could be stored in a cache line that would result in a compressed size, which is a multiple of a MAG. However, this is not a practical solution as it will not only explode the total metadata size but will also increase the metadata that needs to be stored in the metadata cache. As a result, only a few block addresses can be stored in a block of a metadata cache as explained in Section IV-F. This leads to a high miss rate, which will require two accesses to DRAM and defeats the purpose of compression to save memory bandwidth. Instead of storing 8 bytes in a metadata cache, we investigate the possibilities of further reducing the delta size to accommodate these 8 bytes so that the resulting compressed size is a multiple of a MAG. As BDI exploits the narrow dynamic range to store the deltas in fewer bits, there is a possibility that the deltas are even narrower than 2 bytes and 1 byte. Fortunately, to reduce a compressed size by 8 bytes, we only need to reduce the delta size by 2 bits.

Figure 5 shows a representational example of BDI compression with leading zeros. While the principle applies for larger cache lines as well, a 32-byte cache line is used for illustration. BDI compresses the cache line with 4-byte base and 1-byte deltas, however, as the values are so similar, 4 bits per delta would actually be sufficient. The 4 leading zeros

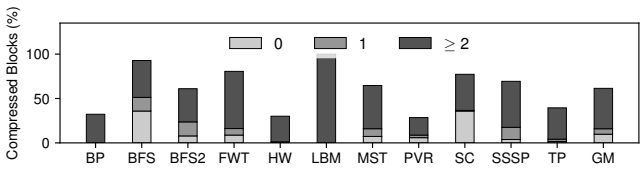


Fig. 6: Number of leading zeros in BDI-compressed blocks.

could be removed without any loss of information.

Our initial investigation shows that deltas in BDI-compressed blocks contain leading zeros. Therefore, we further investigate how often these leading zeros occur in BDI-compressed blocks in real-world applications. Figure 6 shows the number of leading zeros in BDI-compressed blocks. We use Base4B-Δ1B and Base4B-Δ2B encodings for the reasons explained in Section IV-C. As we are interested in the number of leading zeros and not in the delta size itself, we do not distinguish between delta sizes. Moreover, as we need to save 2 leading bits, the number of leading zeros are broken down accordingly in Figure 6. The figure shows that 46% of the BDI-compressed blocks have two or more leading zeros per delta. The 6% of the compressed blocks have only one leading zero per delta and only 10% of the compressed blocks have no leading zeros. The remaining 38% of blocks are uncompressed. As a significant percentage of compressed blocks contain 2 or more leading zeros, we leverage the opportunity to further increase the effective compression ratio by compressing blocks to a multiple of a MAG.

C. Base and Delta Sizes for GPU Workloads

The number of base and delta sizes determine the number of metadata bits as well as compressor and decompressor units. By using fewer metadata bits, the miss rate of a metadata cache can be kept low because more block addresses can be cached. By using a smaller number of compressor units, the area overhead and power consumption can be reduced. Thus, we analyze suitable base and delta sizes, as there might be opportunities for GPU-specific optimizations.

BDI uses multiple compressor units for base sizes of 8 bytes, 4 bytes, and 2 bytes in combination with delta sizes of 4 bytes, 2 bytes, and 1 byte. Each compressor unit independently performs compression in parallel and the base and delta size, which yields the highest compression ratio is chosen. This approach makes sense for CPU workloads because they process diverse data. As GPU workloads are different, they do not require all base sizes offered by BDI.

Kim et al. [13] study dominant data types in GPU workloads and show that 92% of the data consist of integers or floats. Atoofian et al. [22] study data-type specific cache compression for GPUs, showing 8-byte base is rarely used. Our experiments also confirm that the inclusion of 8- and 2-byte base sizes increase the compression ratio on average by 1.5%. This is expected because GPU workloads mostly operate on integer and floating-point data [13], which are both 4 bytes. Therefore, we only use the 4-byte base for MAG-aware BDI compression.

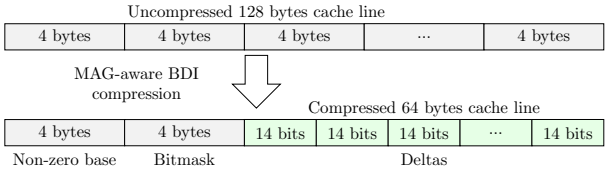


Fig. 7: Basic idea of MAG-aware BDI compression.

TABLE III: MAG-aware BDI encodings. Compressed sizes are given for a 128-byte cache line.

Name	Base (Bytes)	Δ (Bits)	Compressed size (Bytes)	Encoding (Bits)
Base4B-Δ6b	4	6	32	00
Base4B-Δ14b	4	14	64	01
Base4B-Δ22b	4	22	96	10
Uncompressed	-	-	128	11

D. MAG-aware BDI

As shown in Section IV-B, 46% BDI-compressed blocks contain two or more leading zeros. We reduce the delta size by 2 bits to make the compressed size an exact multiple of a MAG. Figure 7 illustrates the idea of MAG-aware BDI using 14-bit deltas and a 4-byte base. By choosing a delta size of 14 bits, 8 bytes can be saved and the cache line can be compressed to 64 bytes, which is exactly 2$\times$ the MAG size. This improves the effective compression ratio from 1.33$\times$ (128/96) to 2$\times$ (128/64). Thus, we replace Base4B-Δ2B encoding with Base4B-Δ14b in MAG-aware BDI. Similarly, MAG-aware BDI replaces Base4B-Δ1B encoding with Base4B-Δ6b, a delta size of 6 bits is used. In addition to adapting Base4B-Δ1B and Base4-Δ2B encodings, we introduce a third delta size of 22 bits. The 22-bit deltas enable a compressed size of 96 bytes, which BDI would have left uncompressed.

Table III shows a summary of base and delta sizes, compressed sizes, and encoding bits. The compressed sizes are shown for a 128-byte cache line, including a 32-bit bitmask and 32-bit base. The raw and effective compressed sizes are the same for MAG-aware BDI. The encoding bits uniquely identify which delta size is used to compress a block. While 6-, 14-, and 22-bit delta sizes are chosen for 32-byte MAG and 128-byte block sizes, the approach is applicable in general. When a delta size does not match the sizes listed in Table III, a cache line is left uncompressed.

$$\#delta\_bits = floor((comp\_size - header\_size)/\#deltas) \quad (1)$$

Equation 1 shows a simple mathematical expression to calculate the number of delta bits ($\#delta\_bits$) for any combination of block and MAG sizes. $comp\_size$ is a multiple of a MAG size less than a block size. For example, for the block size of 128 bytes and the MAG size of 32 bytes, the possible values of $comp\_size$ are 32, 64, and 96 bytes, while for the MAG size of 16 bytes, the possible values of $comp\_size$ are 16, 32, 48, 64, 80, 96, and 112 bytes. $header\_size$ is the sum of a non-zero base and a bitmask. $comp\_size$ and $header\_size$ are in bits. Table III shows non-zero base of 4

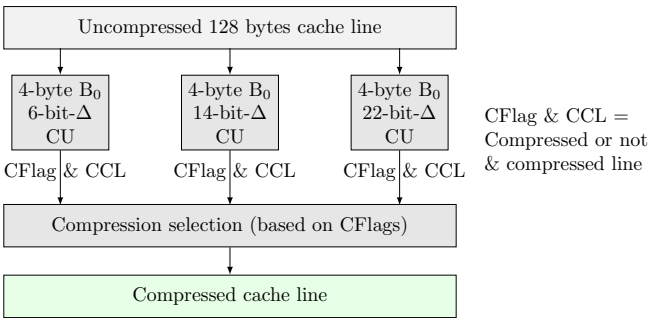


Fig. 8: Overview of a compressor design for MAG-aware BDI with one unit for each base and delta configuration.

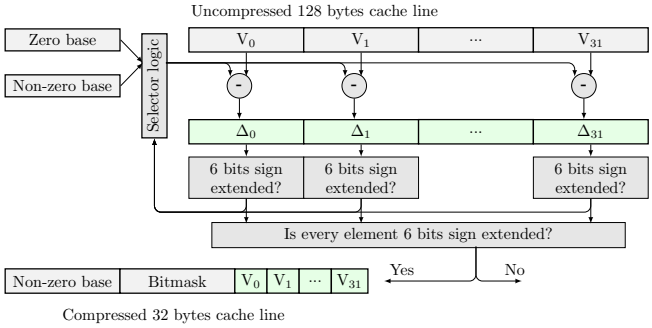


Fig. 9: Compressor unit for Base4B-Δ6b encoding.

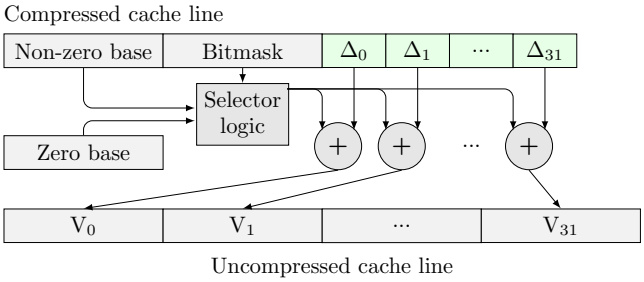


Fig. 10: Decompressor design.

bytes, but other values such as 8, and 2 bytes may also be used to support double- and low-precision operations for machine learning applications. The bitmask depends on the non-zero base and the block size. For example, for 4-byte base and 128-byte block size, the bitmask contains 32 bits, while for an 8-byte base, the bitmask will contain 16 bits. $\#deltas$ are the number of deltas stored in a compressed cache line and are equal to the number of bitmask bits.

MAG-aware BDI uses the delta sizes shown in Table III and does not use the delta sizes of BDI even when a cache line cannot be compressed. For example, when a cache line cannot be compressed using 14-bit deltas but it can be compressed with 16-bit deltas, MAG-aware BDI uses 22-bit deltas instead of 16-bit deltas. This is because in both cases, three bursts of 32 bytes will be read from the off-chip memory, resulting in the effective compression ratio of 1.33×. Moreover, when MAG-aware BDI cannot compress a cache line with 22-bit deltas, it stores the cache line uncompressed, which saves the decompression latency overhead. While BDI can compress a cache line with 24-bit deltas, there will be no gain as four 32-byte bursts will be fetched from the memory. In addition, BDI also needs to perform decompression.

E. Compressor and Decompressor Design

Figure 8 shows an overview of the compressor design for MAG-aware BDI, which in principle is similar to BDI. The main difference is the less number of compressor units due to the reduced base and delta configurations needed for GPUs. While MAG-aware BDI only requires three compressor units, BDI used six compressor units. The compressor consists of three distinct compressor units (CUs), one each for the encoding listed in Table III. As in BDI, each CU takes a cache line as an input and outputs whether or not the cache line can be compressed by this CU. If the cache line can be compressed by a unit, it outputs the compressed cache line as well. The compression selection is made depending on which CUs are able to compress. The 6-bit deltas are preferred over 14-bit deltas, which are preferred over 22-bit deltas. If neither of the units can compress, the cache line is stored uncompressed. The compressor units operate in parallel, reducing the compression latency to basically the time of a vector subtraction.

Figure 9 describes the organization of Base4B-Δ6b compressor unit (CU), which is similar to the one proposed by [6]. The CU operates in two steps. First, all values are subtracted from the zero base and a 6-bit sign-extension check is performed. Depending on the result of the sign-extension check, the corresponding bit is either set or reset in the bitmask. The first value for which the sign-extension check is false, i.e, the first value that cannot be compressed with the zero base, is set to be the non-zero base. Second, the values that cannot be compressed with the zero base are subtracted from the non-zero base. Again, a 6-bit sign-extension check is performed. When all deltas are 6-bit sign-extended, the cache line can be compressed. If the sign-extension check after the second step returns false, even for a single value, then the CU cannot compress this cache line.

Figure 10 provides an overview of the decompressor design. To decompress a block, the decompressor takes the non-zero base, the bitmask, and an array of deltas as inputs. A bitmask is stored to differentiate the zero and non-zero base. To restore a cache line, the deltas are simply added to the corresponding base with the help of the bitmask. As the values can be computed in parallel with a SIMD-style vector adder, the decompression essentially takes as long as vector addition.

F. System Overview with Compression

MAG-aware BDI aims to increase the effective off-chip memory bandwidth and not its capacity, similar to the previous work [4], [10], [12]. The write requests are compressed in a memory controller before writing to off-chip memory, however, no compaction is performed in the off-chip memory. A small on-chip metadata cache is updated with the compressed block size to retrieve it later for a read request. So, the read/write from the off-chip memory is in a compressed form, while the compression is transparent to other components. The

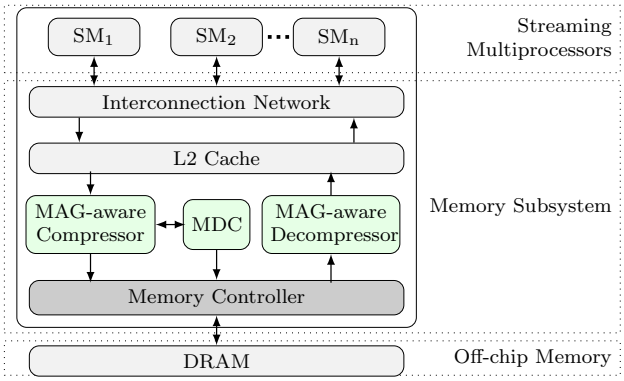


Fig. 11: System overview with MAG-aware compression.

TABLE IV: The main configuration parameters for simulator.

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

compressor, decompressor, and metadata cache (MDC) are integrated into the memory controller as shown in Figure 11. As 2-bit encodings (Table III) can determine a compressed block size, we store these 2 bits in a MDC to know the number of 32-byte bursts to be fetched from the off-chip memory.

For a 32-bit, 4GB DRAM with a block size of 128 bytes, we need 8MB of DRAM for storing the metadata. Like previous work [10], [12], we cache the most recently used metadata in a cache. A metadata cache of 16KB with a 4-way set associativity has a hit-rate of 99%. The average miss rate is less than 1% with 16KB metadata cache for all benchmarks except LBM, which has a 3% miss rate.

The main reason why a small metadata cache provides a high hit-rate is that it can store the metadata for a large number of blocks. For example, a 128-byte cache block can store the metadata of 512 blocks. Thus, a 16KB metadata cache per memory controller can store the metadata of 64K blocks, which covers 8MB of off-chip memory, and for 6 memory controllers, this amounts to 48MB. When we also store the 32-bit bitmask and 4-byte non-zero base in the metadata cache, not only the metadata size will explode to 264MB, which is a significant fraction (6.5%) of memory, but we can cache the metadata of only 0.24MB of off-chip memory, in contrast to 8MB with only 2 encoding bits. We also experimented by storing 32-bit bitmask and 4-byte non-zero base in the metadata cache. The miss-rate increased to 31% and 26% for 16KB and 32KB metadata cache, while the speedup was 2% and 1% lower than the uncompressed baseline.

V. Experimental Methodology

A. Simulator

We modify gpgpu-sim [23] to integrate FPC, CPACK, SLC, BDI, and MAG-aware BDI techniques. Table IV shows the main parameters used for simulations. We use 1 cycle (de)compression latency for BDI and MAG-aware BDI [6]. We use compression and decompression latency of 6 and 10 cycles for FPC and 12 and 13 cycles for CPACK. SLC uses 60 cycles for compression and 20 cycles for decompression [3]. The metadata cache has a latency of 1 cycle and the additional latency of accessing the off-chip memory in case of a miss. SLC builds on E^{2}MC lossless compression using 16-bit symbols, 4 ways parallel decoding, and an online sampling of 20 million instructions [3]. We use a block size of 128 bytes and MAG size of 32 bytes, unless explicitly specified.

TABLE V: Benchmarks used for experimental evaluation.

Name	Abbreviation	Domain	Origin
backprop	BP	Machine learning	Rodinia
bfs	BFS	Graph analytics	Rodinia
bfs	BFS2	Graph analytics	Lonestar
fastWalshTran	FWT	Speech processing	CUDA SDK
heartwall	HW	Image processing	Rodinia
lbm	LBM	Fluid dynamics	Parboil
mst	MST	Graph analytics	Lonestar
PageViewRank	PVR	Data analytics	MARS
scan	SC	Data analytics	CUDA SDK
sssp	SSSP	Graph analytics	Lonestar
transpose	TP	Data analytics	CUDA SDK

B. Hardware Overhead

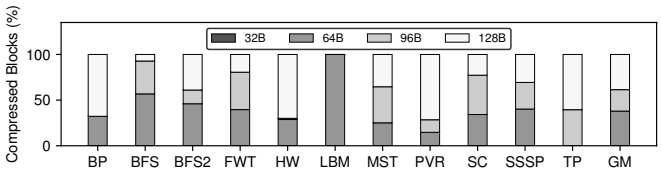
As MAG-aware BDI for GPUs only needs three compressor units in contrast to six for BDI, it reduces area and power overhead. The main additional overhead is from the use of a metadata cache. We use CACTI 6.5 [24] to estimate the area and power cost of metadata cache for a 32 nm technology node. A 16 KB metadata cache increases area and power by 0.005 mm^2 and 0.1 mW, which is 0.0009% and 0.00004% of GTX580, respectively. Furthermore, a read consumes about 0.0014 nJ, while a write takes 0.00097 nJ. While state-of-the-art SLC also has a low area (0.0015%) and power (0.0008%) overhead [3], it builds on top of E^{2}MC, which has a relatively higher area (5.8%) and power (1.5%) overhead [4].

C. Benchmarks

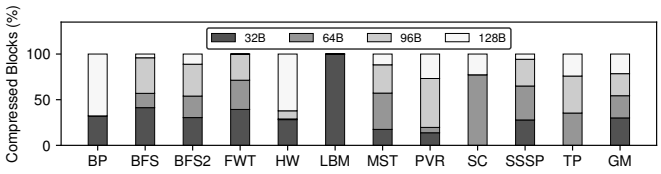
Table V shows the benchmarks used for experimental evaluation. We include memory-bound benchmarks from MARS [25], Rodinia [26], Lonestar [27], Parboil [28] and CUDA SDK [29], covering a diverse range of domains. We include graph benchmarks such as breadth-first search (BFS, both from Rodinia and Lonestar), single-source shortest paths (SSSP), and minimum-spanning tree (MST) as these benchmarks cannot tolerate any approximation errors. Moreover, graph benchmarks have a low dynamic range data which makes them a good use case for BDI compression.

VI. Experimental Results

We compare MAG-aware BDI with BDI, SLC, FPC, and CPACK. We present raw and effective compression ratios. The raw compression ratio is calculated without considering a MAG size, while the effective compression ratio is calculated by scaling up a compressed size to the nearest multiple of a MAG. Lal et al. [3] present results for three variations

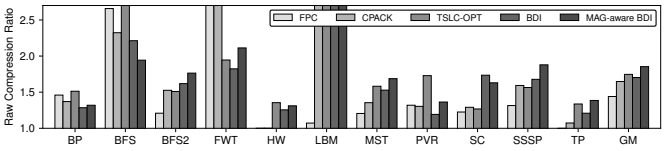


(a) Distribution of compressed block sizes for BDI.

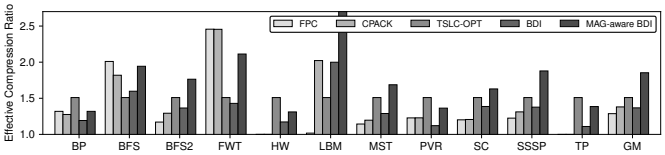


(b) Distribution of compressed block sizes for MAG-aware BDI.

Fig. 12: Distribution of compressed block sizes.



(a) Raw compression ratio.



(b) Effective compression ratio.

Fig. 13: Raw and effective compression ratio.

of SLC using a lossy threshold of 16 bytes and 32 bytes MAG. The first variation called, TSLC-SIMP, uses truncation for the approximation. TSLC-PRED is the second variation and uses prediction to reconstruct approximated symbols. The third variation is called TSLC-OPT and uses extra nodes to further reduce error. The three variations have almost the same speedup as well as compression and only differ in error [3]. Therefore, we show compression ratio and speedup for TSLC-OPT and approximation error for all three variations.

A. Distribution of Compressed Blocks

Figure 12 shows the distribution of compressed block sizes. We show the effective compressed block size because memory is accessed at this granularity. A compressed size of 128 bytes means that the block is stored uncompressed. There are several interesting observations about the results.

First, both BDI and MAG-aware BDI compress a large fraction of blocks for most of the benchmarks. Not only MAG-aware BDI compresses more blocks compared to BDI, but also the distribution of compressed blocks is different. On average, MAG-aware BDI is able to compress 79% of the blocks, while BDI compresses 62%. MAG-aware BDI compresses more blocks due to the additional delta size of 22 bits. This allows MAG-aware BDI to compress blocks with delta sizes 17 to 22 bits that were previously left uncompressed by BDI. *PVR, TP*, and *BFS* gain the most from this optimization because they frequently access blocks with a higher dynamic range.

Second, we observe that MAG-aware BDI decreases the effective compressed size of BDI-compressed blocks by removing leading zeros. Figure 12 shows that on average 30%, 24.5%, 24.5% of the blocks are compressed to 32 bytes, 64 bytes, and 96 bytes, respectively, by MAG-aware BDI, while 0%, 38%, 24% of the blocks are compressed to 32 bytes, 64 bytes, and 96 bytes, respectively, by BDI. While MAG-aware BDI compresses 30% of the blocks to 32 bytes, BDI cannot compress blocks to 32 bytes because compressed size cannot be smaller than 40 bytes as 1 byte is the smallest delta size. For *BP* and *LBM*, MAG-aware BDI compresses blocks to 32 bytes that BDI compresses to 64 bytes. Recall that we are discussing the effective compressed size, which is always a multiple of a MAG. For *BP*, a large fraction of blocks is initially set to zero and MAG-aware BDI compresses them to 32 bytes. *LBM* contains repeating data patterns for which the deltas are zero. The compressed blocks of 32 bytes for *BFS, BFS2, HW, MST*, and *SSSP* are in the range of 17%-41%. As described before, these benchmarks include many blocks with a low dynamic range. *SC* and *TP* contain no 32 bytes compressed blocks due to a higher dynamic range. MAG-aware BDI optimization that reduces a compressed block size from 96 bytes to 64 bytes is mostly visible in *FWT, MST, SC, SSSP*, and *TP* benchmarks.

BFS, BFS2, FWT, LBM, MST, and *SSSP* benefit the most from the memory compression, with $\geq$ 78% of compressed blocks for MAG-aware BDI and $\geq$ 61% of compressed blocks for BDI. We also observe that graph processing benchmarks *BFS, BFS2, MST*, and *SSSP* have a high percentage of compressed blocks. On the one hand, the graph processing benchmarks cannot tolerate any approximation errors and require exact computation, on the other hand, they operate on integer data with a low dynamic range, which makes them highly compressible by a MAG-aware BDI technique. The benchmarks *TP, SC*, and *PVR* have a medium fraction of compressed blocks. *BP* and *HW* benefit the least as they have a large fraction of uncompressed blocks because they mostly operate on floating-point data. Therefore, both BDI and MAG-aware cannot compress most of *BP* and *HW* blocks.

B. Raw and Effective Compression Ratio

Figure 13 shows the raw and effective compression ratio of FPC, CPACK, TSLC-OPT, BDI, and MAG-aware BDI. The three variations of SLC have almost the same compression ratio, therefore, we only show the compression ratio for TSLC-OPT. The average raw compression ratio for FPC, CPACK, TSLC-OPT, BDI, and MAG-aware BDI is 1.44, 1.65, 1.75, 1.70, and 1.85, respectively. The compression ratios of the benchmarks correlate with the distribution as shown in Figure 12a and Figure 12b. Benchmarks with a low dynamic range data have a higher compression ratio, while benchmarks with a high dynamic range have a lower compression ratio.

While CPACK, BDI, and TSLC-OPT have raw compression ratios in the same range, MAG-aware BDI provides a higher compression ratio. MAG-aware BDI also provides a higher raw compression ratio than BDI for all benchmarks except *BFS and SC*. MAG-aware BDI increases the raw compression

ratio for most of the benchmarks because it saves 8 bytes for each compressed block if MAG-aware optimization can be applied. In addition, the additional delta size of 22 bits for MAG-aware BDI also leads to a higher raw compression ratio compared to BDI. The raw compression ratio for *BFS and SC* is higher for BDI for the following reason. *BFS* and *SC* both access a large percentage of compressed blocks for which 1-byte and 2-byte deltas of the BDI are fully occupied. In other words, a large fraction of the compressed blocks have none or only one leading zero in BDI's deltas (see Figure 6). Therefore, BDI compresses these blocks to 40 bytes and 72 bytes, respectively. MAG-aware BDI cannot compress these blocks to a smaller multiple of MAG (32 and 64 bytes) because they do not have enough leading zeros. Instead, MAG-aware BDI compresses these blocks to 64 bytes and 96 bytes, leading to a higher raw compressed size. As many compressed blocks of *SC and BFS* fall into this category, the raw compression ratio is higher for BDI. However, recall that due to the MAG, the amount of data fetched from a DRAM is 64/96 bytes in both the cases as evidenced by the higher effective compression ratio for *BFS and SC* with MAG-aware BDI.

Figure 13b shows the effective compression ratio for FPC, CPACK, TSLC-OPT, BDI, and MAG-aware BDI. The average effective compression ratio for MAG-aware BDI is 1.85, while it is only 1.29 for FPC, 1.38 for CPACK, 1.52 for TSLC-OPT, and 1.37 for BDI. We see that the effective compression ratio is in the same range for FPC, CPACK, and BDI, however, MAG-aware BDI improved the effective compression ratio by 56%, 47%, 33%, 48% over FPC, CPACK, TSLC-OPT, and BDI, respectively. While TSLC-OPT provides a 15% higher effective compression ratio than BDI, it has a much higher area and power overhead because it is based on E^2MC which has a higher complexity [3], [4]. Moreover, MAG-aware BDI also outperforms TSLC-OPT by 33%. Furthermore, as SLC uses MAG-aware approximation, it also results in the loss of accuracy which is presented in Section VI-D.

The effective compression ratio of BDI and MAG-aware BDI varies across benchmarks and also correlates with the distribution shown in Figure 12. Figure 13b shows that MAG-aware BDI provides a higher effective compression ratio than BDI. The effective compression ratio for BDI is about 40% lower than the raw compression ratio because BDI-compressed blocks are not a multiple of a MAG. On the contrary, MAG-aware BDI compresses blocks to exact multiples of a MAG. Therefore, the raw and the effective compression ratio are the same. This observation is also true for *BFS* and *SC*. FPC and CPACK have higher compression ratios than MAG-aware BDI for benchmarks *BFS* and *FWT* because these benchmarks exhibit patterns that are better exploited by FPC and CPACK. There are some missing bars for *HW* and *TP* as corresponding techniques are unable to provide any compression.

C. Speedup

Figure 14 shows the speedup of FPC, CPACK, TSLC-OPT, BDI, and MAG-aware BDI compared to an uncompressed baseline. The average speedup for FPC, CPACK, TSLC-OPT, BDI, and MAG-aware BDI is 13%, 17%, 20%, 18%, and 28%, respectively. The speedup of MAG-aware BDI is higher or equal to BDI for all benchmarks. On average, the speedup of MAG-aware BDI is 10% higher compared to BDI and 15% higher compared to FPC. TSLC-OPT has only 2% higher speedup compared to BDI, but 8% lower than MAG-aware BDI. While TSLC-OPT has a 15% higher effective compression ratio than BDI, it has only a 2% higher speedup compared to BDI because SLC has a much higher decompression latency. The increase in the speedup is due to the increase in the effective compression ratio, which in turn increases the effective DRAM bandwidth. MAG-aware BDI achieves a maximum speedup of up to 27% compared to BDI. While FPC has a slightly higher effective compression ratio than MAG-aware BDI for *BFS*, the latter has a higher speedup. This is because MAG-aware BDI has a lower latency.

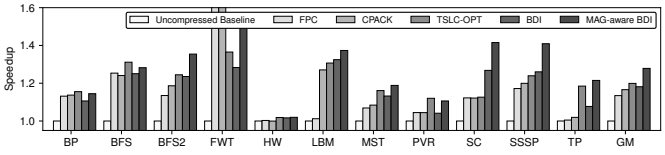


Fig. 14: Speedup of FPC, CPACK, TSLC-OPT, BDI, and MAG-aware BDI.

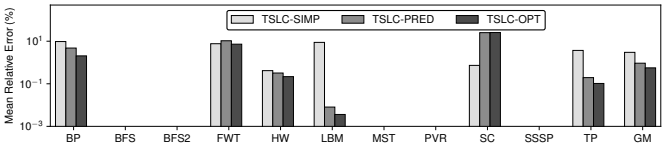


Fig. 15: Approximation error for SLC. There are no bars for some benchmarks as they are not safe to approximate.

Besides leading zeros, the deltas may also contain leading ones. However, to also include leading ones to reduce the deltas size, we need to sacrifice 1-bit of the deltas for the sign bit. On the one hand, including leading ones may increase the compression ratio. On the other hand, they may also decrease the compression ratio as deltas now must fit in one bit less. When we also include leading ones, there is a 1.3% decrease in the compression ratio and a 2.4% decrease in the speedup.

D. Approximation Error due to SLC

Figure 15 shows the mean relative error introduced by the three different variations of SLC. The geometric mean of the error is 3.0%, 0.92%, and 0.56% for TSLC-SIMP, TSLC-PRED, and TSLC-OPT, respectively. The error reduces from TSLC-SIMP to TSLC-OPT, which is the most optimized version. However, there are a few interesting observations. First, for *SC* benchmark, the error increases from 0.74% to almost 25% after the prediction, which shows occasional misprediction can actually increase the error. Second, even after optimizations, TSLC-OPT has an error of 7.2% for *FWT*. There are no error bars for *BFS, BFS2, MST, PVR*, and *SSSP* because these benchmarks have no safe to approximate load-/store and therefore, no approximation is applied to them [3].

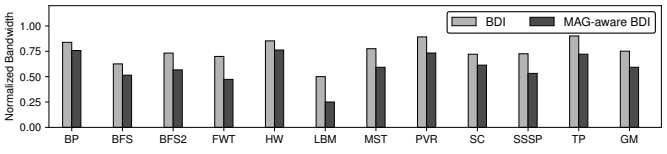
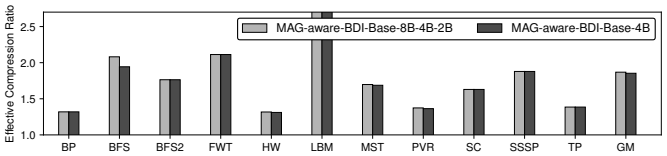
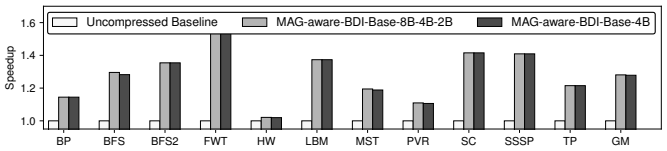


Fig. 16: Bandwidth reduction.



(a) Effective compression ratio after the inclusion of 8- and 2-byte base sizes.



(b) Speedup after the inclusion of 8- and 2-byte base sizes.

Fig. 17: Effective compression ratio and speedup after the inclusion of 8- and 2-byte base sizes.

E. Bandwidth Reduction

Figure 16 shows the reduction in bandwidth requirement for BDI and MAG-aware BDI compared to an uncompressed baseline. The geometric mean of the reduction in bandwidth is 25% for BDI and 41% for MAG-aware BDI. The reduction in the off-chip memory bandwidth is due to the use of compression and the bandwidth reduction is higher for MAG-aware BDI because it has a higher effective compression ratio.

F. Effect of More Base Sizes

Figure 17 shows the effect of including 8- and 2-byte bases on the effective compression ratio and speedup for MAG-aware BDI. The average effective compression ratio is increased by only 1.5% and the maximum gain of 14% by *BFS* when we also include 8- and 2-byte base sizes.

Figure 17b shows the speedup for MAG-aware BDI with 8-, 4-, 2-byte base sizes, and with only 4-byte base size. The average speedup is increased by only 0.2% by adding 8- and 2-byte base sizes. The maximum speedup of 0.5% is gained by *BFS*. As the average increase in the effective compression ratio and speedup with the inclusion of 8- and 2-byte base sizes is only marginal, our work commensurate with the previous work [13], [22] that has shown the dominant data types in GPU workloads are integer and floats. The additional base sizes require more compressors, metadata bits, and the gain is relatively low, so we only include a 4-byte base in MAG-aware BDI. However, if a situation warrants additional base sizes, MAG-aware BDI can be extended straightforwardly.

G. Analysis of Different MAG Sizes

DDR/GDDR/HBM popularly employs a 32-byte MAG size. The reason for using 32-byte MAG seems to be that some parts of the micro-architecture have been optimized for it and keeping the same access granularity enables GPU architectures that are optimized for one generation of memory to transition to next-generation with minimal effort. Nevertheless, a MAG size may vary across systems. A MAG size affects the size and number of deltas as well as the number of compressors as one compressor is needed for each delta size. For example, for a MAG size of 32 bytes, we use three compressors, one each for 6-, 14-, and 22-bit deltas, to compress a block to 32, 64, and 96 bytes. For MAG size of 16 bytes, three compressors are not enough to leverage the more fine granular accesses. We need seven compressors, one each corresponding to 2-, 6-, 10-, 14-, 18-, 22-, and 26-bit deltas, to compress a block to 16, 32, 48, 64, 80, 96, and 112 bytes. Similarly, for a MAG size of 64 bytes, we only need one compressor to compress a block to 64 bytes. For a compressed block $\leq$ 64 bytes, a memory controller will fetch 64 bytes, and for a compressed size $>$ 64 bytes, it will fetch 128 bytes. Furthermore, we need 3 bits for storing 8 encodings for MAG size of 16 bytes, and only 1 bit to store two encodings for MAG size of 64 bytes.

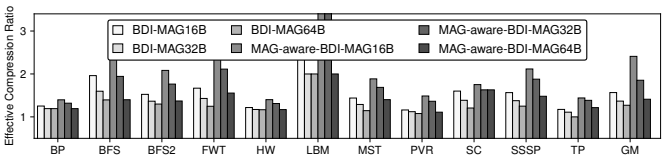


Fig. 18: Effective compression ratio of BDI and MAG-aware BDI for different MAG sizes.

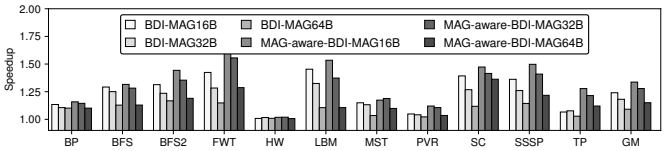


Fig. 19: Speedup of BDI and MAG-aware BDI normalized to uncompressed baseline for different MAG sizes.

Figure 18 shows the effective compression ratio for different MAG sizes. The geometric mean (GM) of the effective compression ratio is 1.57, 1.37, and 1.27 for BDI for 16-, 32-, and 64-byte MAG sizes, respectively. The GM of the effective compression ratio is 2.41, 1.85, and 1.41 for MAG-aware BDI for 16-, 32-, and 64-byte MAG sizes, respectively. We observe that MAG-aware BDI provides a higher effective compression ratio. The effective compression ratio decreases with the increasing MAG size. This is because a block has a lower probability of achieving a higher effective compression for a large MAG size. For example, for a MAG size of 16 bytes, a block can be effectively compressed for any compressed size $\leq$ 112 bytes, while for a MAG size of 64 bytes, a compressed size should be $\leq$ 64 bytes to be effectively compressed. Moreover, the maximum compression ratio is much higher for a small MAG size. For example, for a MAG size of 16 and 32 bytes, the maximum effective compression ratio can be 8$\times$ and only 2$\times$, respectively. *LBM* benefits the most from a smaller MAG (effective compression ratio of 7.9$\times$ for 16-byte MAG) as many blocks have leading zeros that can be compressed to 16 bytes by MAG-aware-BDI-16B,

while MAG-aware-BDI-32B and MAG-aware-BDI-64B will compress these blocks to 32 bytes and 64 bytes, respectively.

Figure 19 shows the speedup of BDI and MAG-aware BDI for different MAG sizes. The average speedup is 1.24, 1.18, and 1.09 for BDI for 16-, 32-, and 64-byte MAG sizes, while the average speedup is 1.34, 1.28, and 1.15 for MAG-aware BDI. The speedup decreases with the increase in the MAG size, which corroborates with the effective compression ratio (Figure 18). The average speedup is 10% for MAG-aware-BDI over BDI for 16- and 32-byte MAG sizes. The gap between the raw and effective compression ratio increases with the increase in the MAG size, however, it also gets difficult to achieve a high effective compression ratio as the probability to compress a block to a multiple of a MAG size decreases.

VII. Conclusions

The large memory access granularity (MAG) of off-chip memory that is employed in multi-/many-core systems to amortize the control overhead limits memory compression techniques from achieving a high effective compression ratio. While MAG-aware approximation has been used to increase the effective compression ratio, its usability is limited to error-resilient applications. While a lossless MAG-aware compression has a wider application, unfortunately, we show that state-of-the-art lossless memory compression techniques except BDI cannot be adapted to a MAG size due to the way they perform compression. The advantage of BDI over other memory compression techniques such as FPC, C-PACK, E^{2}MC is that whereas BDI compresses all words of a block with the same encoding that leads to a static compressed size, the other techniques compress each word separately that leads to a dynamic compressed size. As the potential compressed sizes are known statically, we propose extensions and GPU-specific optimizations to adapt BDI to be MAG aware to increase the effective compression ratio. We leverage the observation that deltas in a BDI-compressed block often contain enough leading zeros to compress a block to a multiple of MAG without any loss of information. We show that MAG-aware BDI provides, on average, 48% higher effective compression ratio, 10% (up to 27%) higher speedup, and 16% bandwidth reduction compared to BDI. While BDI has a compression ratio similar to several state-of-the-art techniques, MAG-aware BDI outperforms FPC, CPACK, SLC by 56%, 47%, and 33%, respectively. MAG-aware BDI also provides a higher effective compression and speedup across different MAG sizes. While a larger MAG results in a bigger difference between the raw and effective compression ratio, the probability to compress a block to a multiple of a MAG size also decreases.

References

[1] NVIDIA, "NVIDIA Unveils CUDA™-The GPU Computing Revolution Begins," 2006, https://www.nvidia.com/object/IO_37226.html.

[2] NVIDIA, "Volta Architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[3] S. Lal, J. Lucas, and B. Juurlink, "SLC: Memory Access Granularity Aware Selective Lossy Compression for GPUs," in *Proc. DATE*, 2019.

[4] S. Lal, J. Lucas, and B. Juurlink, "E^{2}MC: Entropy Encoding Based Memory Compression for GPUs," in *Proc. 31st IPDPS'17*.

[5] A. Alameldeen and D. Wood, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," in *Technical report, University of Wisconsin-Madison*, 2004.

[6] 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. 21st PACT*, 2012.

[7] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas, "C-Pack: A High-Performance Microprocessor Cache Compression Algorithm," *IEEE Transactions on VLSI Systems*, 2010.

[8] A. Arelakis, F. Dahlgren, and P. Stenstrom, "HyComp: A Hybrid Cache Compression Method for Selection of Data-type-specific Compression Methods," in *Proc. 48th MICRO*, 2015.

[9] S. Lal, J. Lucas, and B. Juurlink, "QSLC: Quantization-Based, Low-Error Selective Approximation for GPUs," in *Proc. DATE*, 2021.

[10] 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. 21st PACT*, 2012.

[11] 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. 42nd ISCA*, 2015.

[12] 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. 42nd ISCA*, 2015.

[13] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane Compression: Transforming Data for Better Compression in Many-core Architectures," in *Proc. 43rd ISCA*, 2016.

[14] 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. ISCA*, 2015.

[15] A. Arelakis and P. Stenstrom, "SC2: A Statistical Compression Cache Scheme," in *Proc. 41st ISCA*, 2014.

[16] A. Arunkumar, S.-Y. Lee, V. Soundararajan, and C.-J. Wu, "LATTE-CC: Latency Tolerance Aware Adaptive Cache Compression Management for Energy Efficient GPUs," in *Proc. HPCA*, 2018.

[17] M. Khavari Tavana, Y. Sun, N. Bohm Agostini, and D. Kaeli, "Exploiting Adaptive Data Compression to Improve Performance and Energy-Efficiency of Compute Workloads in Multi-GPU Systems," in *Proc. IPDPS*, 2019.

[18] S. Lal and B. Juurlink, "A Case for Memory Access Granularity Aware Selective Lossy Compression for GPUs," in *ACM Student Research Competition, MICRO*, 2018.

[19] J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," in *Proc. 33rd MICRO*, 2000.

[20] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly Compressed Pages: A Low-complexity, Low-latency Main Memory Compression Framework," in *Proc. 46th MICRO*, 2013.

[21] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K. H. Kim, and M. B. Healy, "Attache: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth Overheads," in *Proc. 51st MICRO*, 2018.

[22] E. Atoofian and S. Rea, "Data-Type Specific Cache Compression in GPGPUs," *The Journal of Supercomputing*, 2018.

[23] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *ISPASS*, 2009.

[24] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies," in *Proc. 35th ISCA*, 2008.

[25] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating MapReduce with Graphics Processors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 22, no. 4, pp. 608–620, 2011.

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. IISWC*, 2009.

[27] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A Suite of parallel irregular programs," in *Proc. ISPASS*, 2009.

[28] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois, Tech. Rep., 2012.

[29] NVIDIA, "CUDA Toolkit," *https://developer.nvidia.com/cuda-toolkit*.