2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Bit-Plane Compression: Transforming Data for Better Compression in Many-core Architectures

Jungrae Kim* Michael Sullivan** Esha Choukse* Mattan Erez*

*The University of Texas at Austin
{dale40, esha.choukse, mattan.erez}@utexas.edu

**NVIDIA corporation
misullivan@nvidia.com

***Abstract*—As key applications become more data-intensive and the computational throughput of processors increases, the amount of data to be transferred in modern memory subsystems grows. Increasing physical bandwidth to keep up with the demand growth is challenging, however, due to strict area and energy limitations. This paper presents a novel and lightweight compression algorithm, *Bit-Plane Compression (BPC)*, to increase the effective memory bandwidth. BPC aims at homogeneously-typed memory blocks, which are prevalent in many-core architectures, and applies a smart data transformation to both improve the inherent data compressibility and to reduce the complexity of compression hardware. We demonstrate that BPC provides superior compression ratios of 4.1:1 for integer benchmarks and reduces memory bandwidth requirements significantly.**

I. Introduction

The computational throughput of many-core architectures is increasing exponentially and we need corresponding increases in data transfer to feed the compute units. The data bandwidth of off-chip main memory scales poorly, however, due to pin and energy limitations—the ITRS projects that package pin counts will scale less than 10% per year [1]. At the same time, per-pin bandwidth increases come with a difficult tradeoff—a high-speed interface requires additional circuits (e.g. delay-locked loops, phase-locked loops, on-die termination [2], [3], [4], [5]) that burn static and dynamic power. These factors conspire to often make the main memory link a system performance bottleneck in many-core architectures [6].

Data compression can solve this growing challenge by increasing the effective memory throughput without augmenting the physical bandwidth [6], [7]. Compression for memory is challenging, however, because it must be low latency and be able to handle relatively fine-grained random accesses. As a result, current compression techniques that target main memory or caches generally only achieve compression ratios of around 2:1 for integer benchmarks and rarely more than 1.5:1 for floating point benchmarks. In this paper, we present a novel data compression technique, *Bit-Plane Compression (BPC)*, with an average compression ratio of 4.1:1 for integer benchmarks and 1.9:1 for floating-point benchmarks.

Applications targeting many-core architectures generally make nearly all data accesses to large arrays in memory; we show that this fact holds across a set of CUDA GPGPU benchmarks (from Rodinia [8], Parboil [9], and Lonestar [10]). Many of these arrays are of primitive type (e.g., arrays of `int`). Other arrays are of composite types, but they contain multiple fields of the same primitive type (e.g., arrays of `struct {int node; int edge;}`). Arrays of primitive and composite-homogeneous types both have a single data type and can be compressed efficiently by using a compression algorithm that exploits type-related word-to-word similarities.

Prior to encoding, BPC transforms each memory data block using a novel and lightweight transformation, *Delta-BitPlane-XOR (DBX)*, to improve the inherent compressibility of data. The transformed data has significantly higher inherent compressibility and increased value locality when the data block is homogeneously typed. BPC combines run-length encoding with a type of frequent pattern encoding to compress the pre-coded data. Because of its effective transform and matching efficient encoding, the low-cost BPC algorithm provides significantly better compression ratios than prior techniques across a large range of benchmark applications.

Conventional memory interfaces access DRAM at a coarse granularity (typically a half or a full cacheline). Such accesses curb the benefits of compression because the coarse-grained transfers lead to fragmentation and waste savings. We therefore evaluate the performance benefits of BPC in the context of a packetized memory interface, such as that of the Hybrid Memory Cube [11], [12]. We use a modified packetized interface that can densely pack the compressed transfers to measure the maximum bandwidth reduction that is achieved with BPC.

We target a GPGPU system in our evaluation because the performance on such systems is often bound by memory bandwidth. Also, they are designed to easily extend to growing core and functional unit counts, where bandwidth is of even greater concern.

We summarize the key contributions of this paper below:

- We develop *Bit-Plane Compression (BPC)*, which introduces the concept of a bit-plane transformation to memory compression. BPC significantly outperforms prior memory compressors achieving an average compression ratio of 4.1:1 and 1.9:1 for integer- and floating-point heavy benchmarks respectively; the best existing compressors, C-pack [13] and Base Delta Immediate [14], achieve just over 2.3:1 and 1.5:1 for integer and floating-point respectively.
- BPC is a general memory compression mechanism but we focus our evaluation on the benefits it provides for bandwidth-constrained GPGPU many-core processors. In this context, we integrate BPC with a packetized memory

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.37

interface and demonstrate that BPC can either achieve performance benefits at a given link bandwidth or match performance goals while consuming about half the link bandwidth, or less.

II. Background

This section reviews the concepts and terminology that are fundamental to a full description and evaluation of BPC and its associated packet interface.

A. Data Compression Background

Lossless compression encodes data to store or transfer it with fewer bits and can later decode the data into precisely its original form. Lossless compression techniques can be grouped into two categories: *fixed-width coding (FWC)* and *variable-width coding (VWC)*. FWC compresses data by replacing each *fixed-length input symbol* with a corresponding *variable-length output code symbol*. The most frequent symbols are assigned to the shortest codes to statistically reduce the aggregate data size. Typically, the length of each code symbol is approximately proportional to the negative logarithm of its probability (e.g. Huffman coding [15]). The maximum compression ratio of FWC is bound by the *entropy* of its input data [16], and some encoders (e.g., Arithmetic coding [17]) can come very close to this bound. In information theory, the entropy of a set of elements with occurrence probabilities of $p_1,\cdots,p_n$ is defined as: $H(p_1,\cdots,p_n)=\sum p_i log_2 \frac{1}{p_i}$. Data with lower entropy has better compressibility because a smaller number of symbols dominates the input data.

Compression ratio can be improved by increasing input symbol size such that frequently-appearing large input symbols are represented by short output code symbols. However, the mapping space of FWC explodes with longer symbols, making such encodings impractical. VWC, on the other hand, maps frequent subsets of multi-symbol sequences with varying width into output codes (of fixed or variable width) and can achieve a high compression ratio while limiting table size; the well known *run length encoding (RLE)* and *Lempel-Ziv (LZ)* schemes perform variable width compression. Typically, VWC relies on a large dictionary of static or dynamic frequent strings and the storage requirement of the dictionary can be prohibitively high for hardware compressors on small blocks. An important exception is run-length encoding, which is a light-weight VWC, where simple sequences (repetitions of a symbol) are detected without an explicit dictionary. VWC can outperform the entropic limit by combining many input symbols in a compact way.

A compression algorithm can optionally transform the data prior to encoding. This is common with lossy compression schemes, where transformations are used to reduce the perceived loss of information (e.g., perceptual video encoding [18]). As we discuss later, transformations can improve compression ratios for lossless compression as well.

B. Many-Core Application Data Characteristics

Applications targeting many-core architectures, such as GPGPUs, often utilize an SPMD (single-program multiple-data) execution model with explicit or implicit SIMD/SIMT

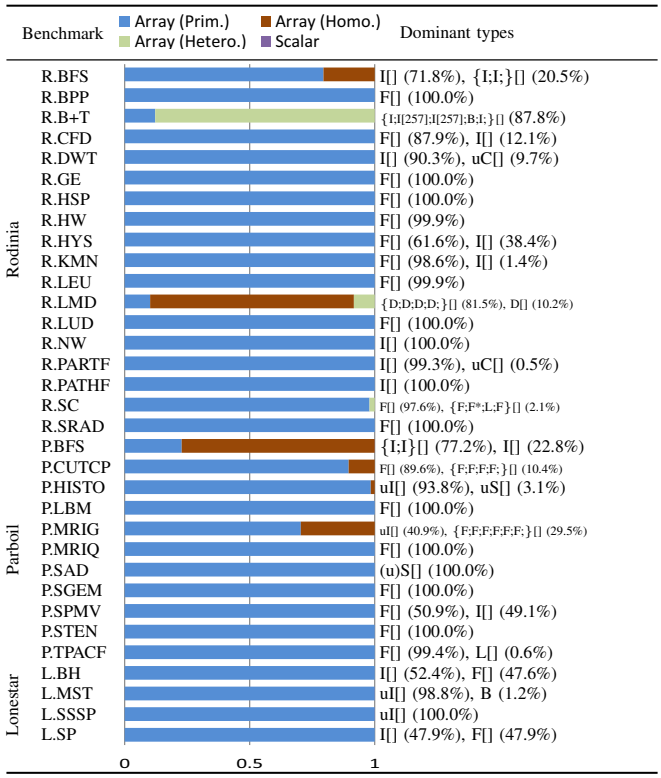


TABLE I: The data types in the global memory allocated for different GPGPU benchmarks. u/B/C/S/I/L/F/D/{} stand for unsigned/bool/char/short/int/long/float/double/struct, respectively. There is at most only 1.6e-4% of memory devoted to scalar data; most memory is either used for arrays of primitives or arrays of composite-but-homogeneous types. A full version of this table is available on http://goo.gl/BrIVWi.

(single-instruction multiple-data/thread) components. This execution model enables very high computational throughput with manageable control overheads. Such programs typically perform the same operation on multiple data that are stored in large arrays. Table I shows a set of GPGPU benchmarks, written in CUDA, and their input data along with a breakdown of global memory allocation based on program data types. We measure the allocated sizes of cudaMalloc(), followed by source code analyses to match data type. Some applications (MUMmerGPU and k-Nearest Neighbor in Rodinia and some applications in Lonestar) are excluded as they fail to run on GPGPU-Sim due to unsupported APIs. Myocyte (Rodinia) is excluded due to its small memory footprint (812B); the listed applications have footprints ranging from 0.5MB to 900MB.

The table indicates that most applications have no scalar data in global memory and that the allocation of non-array data (if there is any) is tiny compared to the large allocations of array data. An array can be a primitive-type array, a composite-but-homogeneous-type array, or a composite-and-heterogeneous-type array (e.g. array of struct {`int` node; `float` weight}). The first two array types are *homogeneously-typed* and often have high value locality. In the

table, all but three applications have practically 100% of global memory allocated as homogeneously-typed arrays. R.B+T has a heterogeneous array with struct of 516 `int` and 1 `bool`, which is almost homogeneous as 16 out of its 17 128B cache lines have purely `int`s. R.LMD has 8.4% of data in struct of 4 `int` and 1 `long` and R.SC has only 2.1% of data in heterogeneous array while the rest of data is homogeneously-typed.

C. Memory Background

DRAM is widely used as system main memory for its high density and low cost. An access to DRAM initiates a burst of data transfer to amortize control overhead and the unit of transfer is 32B to 128B, depending on data channel width and DRAM burst length. The data block is transferred over a bidirectional data bus and the memory controller prevents bus contention by scheduling commands based on expected data transfer duration. This interface can be problematic with data compression because: (1) the fixed and coarse transfer granularity can easily overwrite the fine-grained bit savings from compression, and (2) the transfer duration of variable-length compressed read data is unpredictable to the memory controller, which prevents from reusing the saved time slots for other blocks to improve effective bandwidth.

These limitations are, however, changing as some recent high-bandwidth memories use packetized unidirectional link interfaces to improve data bandwidth. Unidirectional links have a single driver so do not suffer from bus contention issues. The *Hybrid Memory Cube (HMC)* [12], [11], for example, uses a pair of undirectional data links and achieves up to 15Gb/s per direction per lane, compared to the 7Gb/s possible with high-end GDDR5 parts. HMCs have a packet-based interface to interleave read and write streams on the same link. Requests are sent to memory as a packet via a downstream link. A memory controller on the memory side (called a vault controller), receives the requests and issues DRAM commands based on internal DRAM state. Responses are sent back to the processor via an upstream link. To provide out-of-order service, each response is tagged with its request information to identify the originating request. All packets have a header and a tail to hold the tag and other meta data, such as checksum for link error detection. Packets without data (read request and write response) have only a 64-bit header and a 64-bit tail, while packets with data (read response and write request) have a 64-bit header and a 64-bit tail with multiple 128-bit data FLITs (flow control digits) in between. This fine-grained control over data size (16B) also reduces internal fragmentation with data compression, compared to 32B to 128B granularities found in other DRAMs.

III. Prior work

In this section, we talk about prior research on compression algorithms and the architectural changes to get the benefits from data savings.

A. Memory Compression Algorithms

There is significant amount of prior work on compression algorithms for memory data, targeting effective storage capacity, effective bandwidth or power efficiency. There are several challenges that need to be addressed to get benefit out of compression in the memory system. Most importantly, the random access nature of the memory system limits the compression block size, leading to a lower compression ratio. Furthermore, sensitivity to access latency prevents the use of compute-intensive compressors. As a result, most existing memory compression algorithms provide modest compression ratios of roughly 2:1 for integer-type benchmarks. We briefly describe some prominent prior work below.

Frequent Pattern Compression (FPC) [19] focuses on universally frequent patterns of integer values and encodes each 32-bit symbol using a static mapping. Patterns such as small integer value in a large data type are compressed to a smaller type and decompressed by a sign extension. This FWC is augmented by *Zero Run-Length Encoder (Z-RLE)*, a VWC with static mapping, to compress frequent zero patterns.

Frequent Value Compression [20] observes that caches have a small number of frequent 32-bit values and replaces those values with indices into a dictionary of the frequent values. IBM Memory Expansion Technology (MXT) [21] and X-RL [22] divide memory into large blocks (a few KBs) and maintain a per-block dictionary to avoid re-compressing other blocks. MXT uses VWC to replace variable-width strings with a fixed-width index to the dictionary. X-RL on the other hand, replaces 32-bit fixed-size input with a variable-length index to a fully or partially matching entry and uses Z-RLE on it.

Benini et al. [23] propose to compress caches using two dynamic memory-wide dictionaries. At each point in time, one dictionary acts as a master and the other as a slave. A dictionary transitions from being a master to slave based on changing data patterns. Once all cache lines compressed by a dictionary are either evicted or re-compressed, the master and slave dictionaries are swapped. SC^2 [24] observes that data statistics change occasionally and re-compression overhead can be acceptable in caches. It uses Huffman code [15] to build dynamic dictionary across a cache, exploiting statistical redundancy, and achieves an exceptional compression ratio of up to $4\times$.

Although dynamic dictionaries provide high compression, they are a poor match for main memory. This is because, unlike caches, re-compression overhead can be prohibitively high in main memory due to bandwidth and energy issues. Additionally, they rely on content-addressible memory for lookup, resulting in costly compressor energy and area overheads.

C-pack [13] utilizes both static patterns and a dynamic dictionary. 32-bit input symbols are compared against entries in a per-block dynamic dictionary, and frequent patterns on partial matching are encoded using a fixed mapping. *Base Delta Immediate (BDI)* [14] exploits the small dynamic range of values which is common in integer and pointer array types. It encodes a block of data as a single base value, followed by a set of differences relative to that base.

Two recent memory compression papers, COP [25] and Frugal ECC [26] use new forms of memory compression for ECC metadata storage. Because they focus on metadata, their compression ratio goals are very modest and the techniques they present are unlikely to result in significant

link performance improvements.

GPUs employ compression for image data, but not for general-purpose data. AMD's Lossless Delta Color Compression and NVIDIA's Delta Color Compression store image data as a delta from the previous pixel and save bandwidth [27].

B. Memory Compression Architectures

While a good compression algorithm can reduce data size, it must be accompanied by an architecture that can turn these savings into performance or other system-level benefits. One significant hurdle is that memory is accessed in fixed-sized blocks, which commonly results in high fragmentation after compression. Also, variable-sized compression makes the starting address of compressed data blocks unpredictable, which makes locating a block challenging. This is especially true in main memory, where additional access for location information is very expensive in terms of latency and energy. Prior work has addressed some of these issues as we describe below. We do not discuss prior work related to compressed caches (e.g. indirect index cache [28], [29], [30], [13]), as it is not pertinent to the architecture we are evaluating.

One common solution for locating the compressed data is to maintain a table of translations between original and compressed addresses. IBM MXT stores compressed data of a 1KB raw data line into fixed size sectors of 256B to avoid external fragmentation. For locating the data, MXT maintains a translation table on a large L3 cache chip to redirect original address to sectors. Ekman [31] uses a TLB-like structure to maintain the sizes of each block and calculate the compressed data address from sizes of its prior blocks. The latency of address calculation is hidden by executing in parallel to LLC access, which increases power consumption.

Linearly Compressed Pages [32] gives another solution for locating compressed memory blocks. It fixes a target compression ratio so that compressed data address can be extracted from the original address using a linear equation, rather than a table lookup. Blocks that cannot be sufficiently compressed are located in reserved storage in the same DRAM row buffer and need an indirection.

As off-chip data transfers consume a growing portion of system energy, data compression for reducing off-chip traffic is gaining more importance. MemZip [33] uses memory compression to reduce memory bandwidth usage instead of trying to increase effective storage capacity. MemZip reduces data transfer granularity with rank subsetting and transfers only valid compressed data from just a subset of memory chips, thus saving bandwidth. Sun Rock CMT processor [34] uses hardware to compress packets sent over links between processor and memory controller. Its data bandwidth, however, is still bounded by uncompressed data between memory controller and memory.

IV. Bit-Plane Compression

BPC starts with a smart data transformation, *Delta-BitPlane-XOR (DBX)*, to improve the compressibility of data while keeping the encoding complexity comparable to existing compressors. Then, we combine two light-weight compression techniques, frequent pattern compression (FPC) and run-length encoding (RLE), to turn the improved compressibility into real bit savings.

A. Transformation for Bit Plane Compression

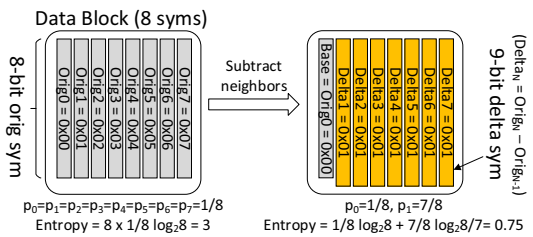
An important but often under-appreciated aspect of compression is the data transformation before encoding. We present and evaluate the novel *Delta-BitPlane-XOR (DBX)* transformation that can: (1) reduce entropy to increase potential compressibility; (2) increase the frequency of all-zero symbols for efficient run-length encoding (RLE); (3) increase the frequency of other simple patterns for frequent-pattern compression (FPC); and (4) have a low-latency and low-cost hardware implementation. We evaluate DBX overheads later, when discussing the full compressor algorithm.

The DBX transformation is composed of three simple transformations applied in sequence: Delta that subtracts neighboring values (Figure 1a); Bit Plane that rotates input symbols such that each of its output symbol includes one bit of every input symbol, all at the same bit position (Figure 1b); and an XOR of neighboring symbols.

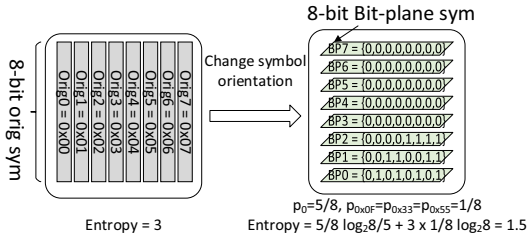
Like prior work ([14]), we use Delta to expose value locality. When neighboring values are similar, subtracting them results in a smaller range of values. Delta processes a 128B block transferred from memory as 32 32-bit input values. The output of Delta is a sequence of difference symbols, where each symbol is one bit wider than the input symbols to accommodate the borrow bit for subtraction; the first symbol is the original input *base symbol*. Despite the extra bit, when value locality exists, entropy is reduced and value range compressed (see example in Figure 1a).

While the reduced-range values already reduce entropy and can be used for compression (e.g., BDI compression [14]), we apply a further important insight: value locality can be further increased when considering bit planes rather than the values themselves. On a homogeneously-typed block, the difference symbols often have another layer of value locality. DBX applies the *Bit-Plane (BP)* transformation (Figure 1b) on the Delta symbols to change their symbol orientation and exploit this value locality. *A bit-plane* is a set of bits corresponding to the same bit position within each word in a data array; in this case the output of Delta. The BP transformation has been used for media compression in the past [35], [36] and we show it works well for memory compression as well. In our case, the output of the combined *Delta-BitPlane (DBP)* transformation is 33 symbols, each corresponding to a bit-plane of the Delta value sequence. A small positive Delta has its upper bits filled with zeros and a small negative Delta has its upper bits filled with ones. Therefore, by XOR-ing neighboring bit-planes, the overall DBX transformation yields a large number of all-zero symbols for small Deltas. The top symbol is the base and contains the sign information of the deltas. We discuss the effectiveness of DBX using analysis of several benchmarks and their memory streams below.

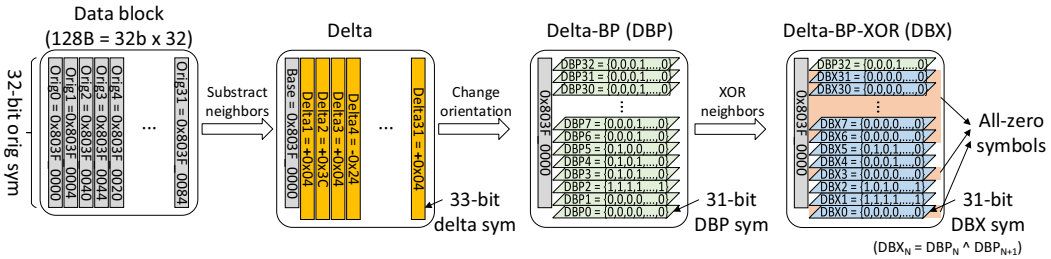
1) Transformation Evaluation: To show the enhanced compressibility with transformations, we start with a detailed analysis of observed symbol patterns and then measure data entropy



(a) An example of delta transformation on 8 8-bit symbol data block. Entropy reduces from 3 to 0.75.



(b) An example of BitPlane transformation on 8 8-bit symbol data block. Entropy reduces from 3 to 1.5.



(c) An example of Delta-BitPlane-XOR transformation on 32 32-bit symbol data block. In the output, many DBX symbols are all-zeros.

Fig. 1: Examples of data transformation to change compressibility.

and repeated zeros of the untransformed data and the Delta and DBX transformations. These metrics are inherent to the data and independent from any particular compression algorithm.

We run the benchmarks listed in Table I on the GPGPU-Sim simulator configured to generally match the NVIDIA GTX480 processor (Table 4b). The simulations run to completion unless first terminated by an overflow of the 32-bit interconnect packet ID; note that L.MST finishes its computation but does not terminate. We collected memory traffic data at the LLC–DRAM interface (i.e., writebacks and cache fills) and match to program data types based on addresses recorded during allocation. Note that the data type breakdown in Table I is based on allocations and is different from memory traffic due to different access patterns and locality (Figure 2a).

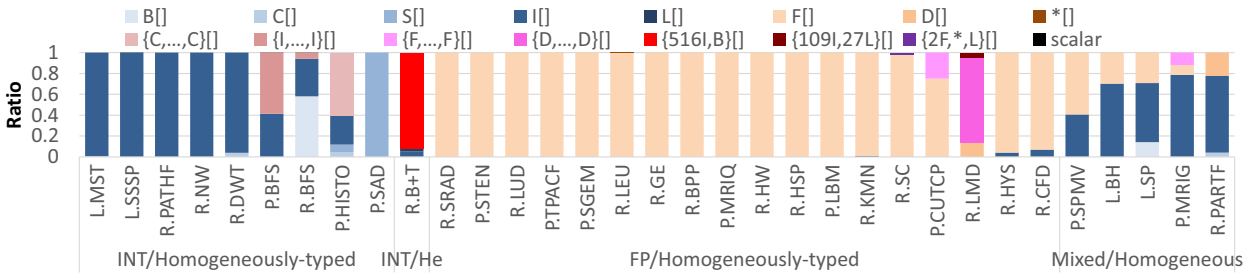
2) DBX Symbol Patterns: Figure 2b shows the detailed DBX symbol statistics of each bit position across several representative applications. For each application, the top most horizontal bar corresponds to the base bit plane and each of the 32 horizontal bars below it correspond to one of the 32 bit positions of the DBX transform. The colors represent different counts of `1`s at that bit-plane. This representation allows us to understand what the benefits of the transformation are and how value characteristics of different applications affect transformation effectiveness. We make 3 important observations and explain the analysis that leads to them below: (1) DBX yields many all-zero symbols, especially toward the higher bit planes, which enables very efficient RLE; (2) a fair number of DBX symbols exhibit only a single, or very few, ones and we will take advantage of this for FPC; and (3) the first two observations hold for floating point (FP) and mixed-type data, though to a lesser degree than for homogeneous integer (INT) types.

Integer applications exhibit many `1`s at the top bit-plane position (the base bit-plane with DBP transformation) because this bit plane contains the sign of the deltas of the different words. Some applications (e.g. R.B+T, R.KMN) have mostly positive Deltas because their values repeat or monotonically increase and their base bit-planes are mostly all-zeros. L.MST and R.DWT, whose data is nearly 100% `int` arrays, have the rest of the upper bit planes (DBX symbols) dominated by all zeros. R.BFS has 58.2% of its data in a `bool` array. As boolean values are stored as 8-bit types in CUDA, only the 0/8/16/24th bit of positions of the original data contain information. This leads to the large fraction of non-zero symbols to the left and right of these bit positions (positions 0/7/8/15/16/23/24), while the other positions are dominated by all-zeros. P.SAD calculates the sum of absolute differences for motion estimation in a video codec and performs a brute force search on image data to find the closest matching block. Because of this exhaustive search, its data is highly random and has relatively fewer zero DBXs.

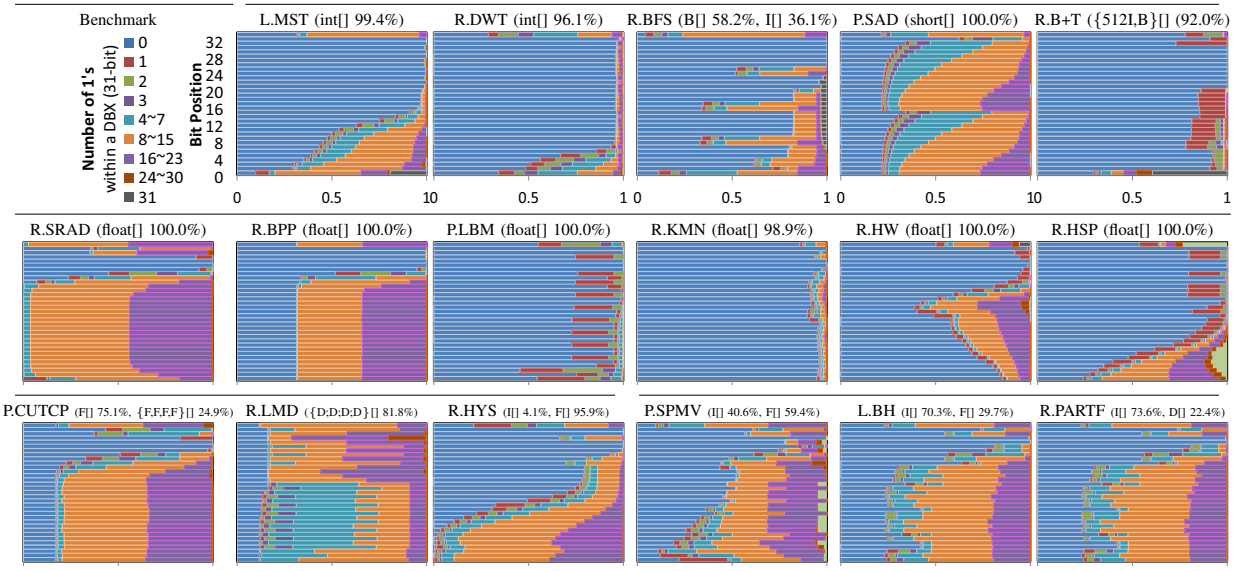
Among the benchmarks that we evaluated, only R.B+T has significant traffic originating from heterogeneous arrays. However, its heterogeneous data type is almost-homogeneous with 16 out of its 17 128B cache lines pure `int`. The other cache lines contain 31 `int`s and 1 `bool`, where the boolean value generates a single outlier value and results in DBX symbols with a single `1`.

With FP data types, a small value difference often results in very different values in the fraction field, significantly hampering compression. The sign and exponent bit fields, however, do not change as rapidly and exhibit value locality. R.SRAD, R.BPP, P.LBM and P.KMN, for example, have frequent all-zero DBX symbols at these positions. R.KMN is dominated by 0.0 float values (the input set is realistic and 0.0 values are common in some classification tasks because of sparse data). P.LBM has repeating non-zero values, since there are long zero runs after DBX but short ones in the original symbols (Figure 2d).

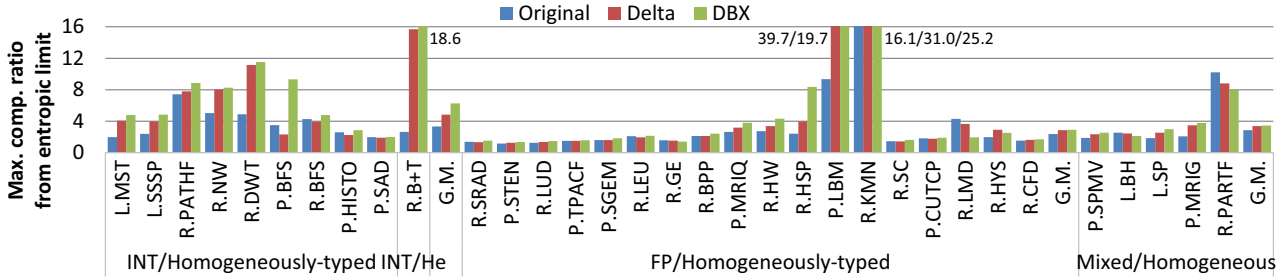
DBX also reveals some interesting value redundancies of floating-point data. R.HW has an outlier distribution of more



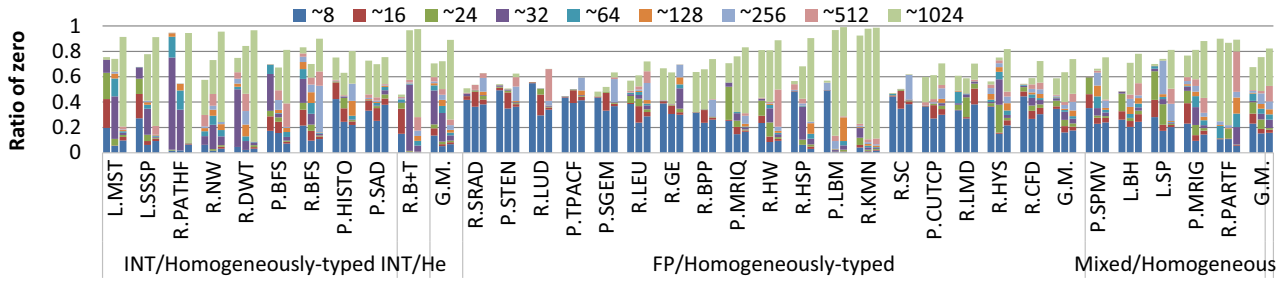
(a) Data type of memory blocks (128B) observed at LLC-DRAM interface; benchmarks are sorted by percentages of int types.



(b) Some examples of DBX transformed data. Each DBP/DBX position is broken down by the number of 1's within a bit-plane. Values are highly biased to zeros and upper bit-planes of INT and sign/exponent bit-planes of FP are frequently all-zeros



(c) Maximum compression ratio of different transformations. The maximum compression ratios are measured by {symbol size} / {weighted average of symbol entropies}, which is described in more details in the text.



(d) Percentage of zero bits in a memory block, broken down by zero run length ranges. First/second/third column for each benchmark represent Original/Delta-transformed/DBX-transformed data, respectively.

Fig. 2: Evaluation of DBX transformation on GPGPU main memory link traffic.

all-zero DBX symbols at lower fraction bit positions, since it stores integer values from image data as floats. R.HSP and R.HYS show a similar pattern. R.LMD has double-type data and its 64-bit values are treated as two consecutive 32-bit values in the DBX transformation resulting in alternating patterns at even and odd positions.

For Mixed-type applications (P.SPMV, R.BH, R.PARTF), where some blocks are homogeneously INT type and the others are homogeneously FP, the distribution of lower all-zero DBX symbols is closely related to the fraction of integer blocks.

3) Entropy: Figure 2c shows the entropic compression ratio bound, which is the maximum compression ratio achievable using fixed-width compression. The inherent compressibility increases by 87% (geometric mean) for INT (including both homogeneous and heterogeneous), 23% for FP, and 20% for Mixed. INT applications have better (lower) entropy after DBX transformation due to frequent all-zeros in upper DBX symbols. The maximum inherent compression ratio of 18.6 is exhibited by R.B+T, despite its partially heterogeneous types. This is because it has frequent all-zero DBX symbols across positions. Many of the FP applications have moderate increases in compressibility due to limited all-zeros from the sign and exponent fields. However, R.HW, R.HSP, P.LBM, and R.KMN exhibit compressibility greater than $4:1$ because of the small dynamic range and many zeros as discussed above. Mixed-type applications tend to compress better than FP ones because of the prevalence of INT values.

DBX only lowers the inherent compressibility for three applications out of the entire 33 applications we evaluated. Both R.LMD and R.PARTF have 64-bit values, which lead to many non-zero DBX symbols because DBX is implemented for 32-bit input symbols. The DBX layout can be changed to cover 16 64-bit symbols rather than 32 32-bit symbols. We do not evaluate entropy of this derivative of DBX because the different sizes of original symbol (64-bit) and bit-plane symbols (15-bit) makes a direct comparison not meaningful. A compression algorithm based on the 16 x 64-bit DBX shows slight compression ratio improvement over 32 x 32-bit DBX in Section V-A. Another application, R.BH, shows a 17% reduction in inherent compressibility because its Deltas have poor value locality.

4) Repeated Zeros: A bit in a DBX symbol can be `1` only if the corresponding delta is non-zero and its absolute value is smaller than the bit position. As a result, DBX-transformed data is highly biased toward `0`. Figure 2d details the fraction of zeros in each 128B block of the original and transformed memory data stream, as well as the statistics of different zero run lengths. Note that zero runs are capped by the block size at 1024 bits. The figure shows that the transformations are generally very effective at increasing both the overall number of `0` bits (70.5/72.3/89.1% for Original/Delta/DBX, respectively, for INT) and the zero run lengths (62.8% of bits are `0` and belong to runs longer than 512 bits with DBX). This is an ideal scenario for cheap and effective run-length encoding. We discuss this further in Section IV-B.

For the FP applications, the improvements from transformation are less dramatic, still substantial with DBX. On average, there are 58.8/63.6/73.9% bits as `0` for Original, Delta, and BPX, respectively. Many of the `0`s added by the transformations in FP come from the sign and exponent components, which account for 28% (9 out of 32 bits) of the bits in single-precision FP. For Mixed-type applications, the fraction and lengths of zeros are in between INT and FP.

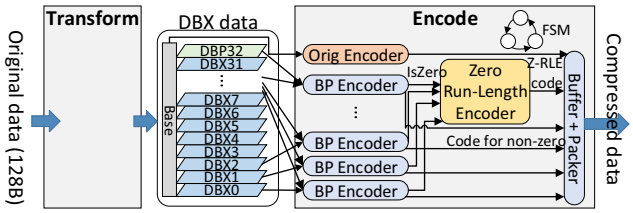
In summary, the DBX transformation reduces data entropy by exploiting type homogeneity and small dynamic ranges, and tailors data for a light-weight VWC, Z-RLE, by maximizing the number of zeros and their run lengths.

B. Encoding for Bit Plane Compression

Based on the improved compressibility of DBX transformed data, we design an efficient compression algorithm, *Bit Plane Compression (BPC)*, to achieve a high compression ratio with acceptable hardware overhead. While sophisticated FWC compressors can approach the entropic limit, their implementation overhead is prohibitive for large symbols: for instance, Huffman code on 31-bit symbols requires a 2^{31}-entry dictionary. Instead, BPC encodes DBP/DBX symbols with a combination of FWC and VWC with small static mappings.

In BPC, long zero runs of bit-planes are encoded by *Zero Bit-Plane Run Length Encoding (ZBP-RLE)*, while non-zero bit-planes are encoded with frequent pattern encoding. This is a similar approach to that taken by some prior work [19]; however, the DBX transformation radically increases the effectiveness of this simple compression algorithm.

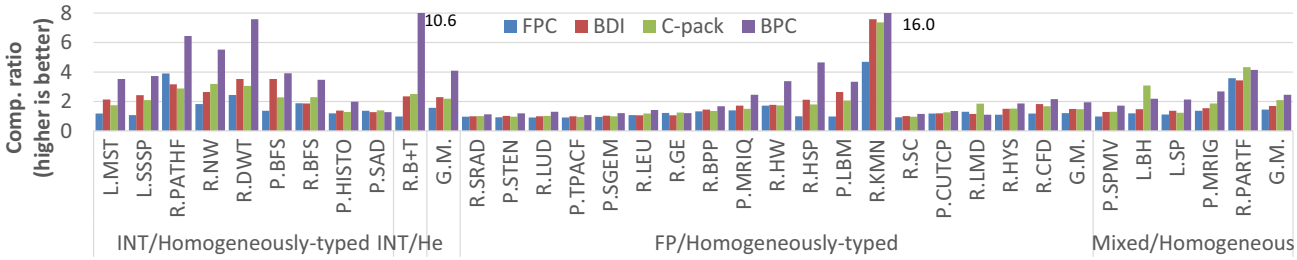
Figure 3a shows an overview of the BPC implementation (a pipelined one, which reuses 9 bit-plane encoders over 4 cycles to process 33 bit-planes). The input to the BPC consists of DBP or DBX transformed bitplanes. In BPC, each input bit-plane is compared against the frequent patterns shown in Table 3b. If a bit-plane is zero, it is collectively encoded with neighboring zero bit-planes, if any, by ZBP-RLE. For non-zero bit-planes, BPC uses one of the 4 patterns as follows. First, a bit-plane with all `1`s is encoded as a 5-bit code. Second, a bit-plane with zero DBP but non-zero DBX is encoded as a 5-bit code. The third common pattern in a bit-plane is two consecutive 1s. If the original symbol has an exceptional value than the rest in that block, its deltas with previous and next symbol have exceptional values, resulting two consecutive `1`s. Likewise, fourth pattern is a bit-plane with one 1, which happens if a block has two consecutive groups with different value clusters. The delta at the boundary then has an exceptional value than other small deltas, resulting in a single `1` in the bit-plane. Both the third and fourth patterns are encoded as 5-bit prefix, followed by the position of first 1. If none of the 4 patterns match, a 1-bit flag indicates a compression failure, increasing the size from 31 to 32 bits. The base (first original) symbol is compressed separately by original symbol encoder as {3'b000}, {3'b001, 4-bit data}, {3'b010, 8-bit data}, or {3'b011, 16-bit data} if its value is 0 or fits into 4/8/16-bit signed integer, respectively. Otherwise, the base symbol is encoded as {1'b1, 32-bit data}.



(a) An overview of BPC.

DBP/DBX Pattern	Length	Code (binary)
0 (run length 2∼33)	7-bit	{2'b01, (RunLength - 2)[4:0]}
0 (run length 1)	3-bit	{3'b001}
All 1's	5-bit	{5'b00000}
DBX!=0 & DBP=0	5-bit	{5'b00001}
Consecutive two 1's	10-bit	{5'b00010, StartingOnePosition[4:0]}
Single 1	10-bit	{5'b00011, OnePosition[4:0]}
Uncompressed	32-bit	{1'b1, UncompressedData[30:0]}

(b) DBP/DBX symbol encoding in BPC. N'bxxxx represents an N-bit code with value xxxx.



(c) Compression ratio comparison.

Fig. 3: Bit-Plane Compression (BPC) overview, code map, and evaluation results.

V. Evaluation

While BPC is a general compressor applicable equally to caches and main memories for storage and for link compression, in this paper, we focus on main memory link compression to address the serious off-chip bandwidth problem in GPGPUs. We first compare the compression ratio of BPC against previous compression algorithms and evaluate the area and latency overheads of BPC. Then, we integrate BPC with an HMC-like packetized link interface to measure the benefits BPC provides in reducing memory bandwidth requirements and improving performance.

A. Compression Ratio Evaluation

We evaluate BPC and existing algorithms (FPC, BDI, and C-pack) on the memory traffic data described in Section IV-A.1. Figure 3c shows the compression ratio comparison. We did not include SC^2 as it cannot be extended to main memory compression in a straight-forward manner due to factors such as the need to recompress after every dictionary change. Note that some applications use input datasets that are randomly generated and such benchmarks unsurprisingly exhibit lower compression ratios. We believe other benchmarks provide a more realistic view of the potential of BPC.

For INT applications, the average compression ratios are 1.6/2.3/2.2/4.1 for FPC/BDI/C-pack/BPC, respectively. BPC is significantly better than the best of the competitors in 9 out of 10 applications. The only case where BPC underperforms any of them is P.SAD (1.3 with BPC vs. 1.4 with C-pack). This is primarily because the brute force search in P.SAD makes values within a block highly distributed and exhibits bit-planes with multiple 1s, resulting in symbols that are not frequent pattern in the BPC encoder. Despite its simplicity, BPC even exceeds the entropic limit of the original symbol streams in 6 out of 10 applications (L.MST, L.SSSP, R.NW, R.DWT, P.BFS, and R.B+T). L.MST and R.B+T have BPC compression ratios of 3.5 and 10.6, respectively, while the entropic limits of their original symbols are 2.0 and 2.7, respectively. This is primarily due to the reduced entropy resulting from the smart DBX transformation and multi-symbol savings from ZBP-RLE.

For FP applications, the average compression ratios are 1.2/1.5/1.4/1.9 for FPC/BDI/C-pack/BPC, respectively. If we exclude R.KMN, which has an outlier compression ratio of 16.0, the compression ratios decrease to 1.1/1.3/1.3/1.7 for FPC/BDI/C-pack/BPC. BPC is worse than C-pack by some margin on R.LMD (1.11 vs. 1.85) due to its 64-bit data type and slightly worse on R.GE (1.20 vs. 1.25). Still, 16 out of 18 FP applications have BPC as the best performing compressor and 6 of them show BPC compression ratios are better than the entropic maximum compression bound achievable with the original symbols. BPC compresses R.HW/R.HSP/R.KMN 1.9/2.2/2.1 times better than competing compressors, respectively. These significant improvements come from previously unexploited redundancy in FP data: zeros at lower fraction bits, small value range of the fraction, and long repeated values. The reasons for the zeros at lower fraction bits are a result of converting integers to float types. Also, BPC does not increase average data size after compression in any of the applications, whereas FPC, BDI, and C-pack increase the size of 8, 3, and 4 applications, respectively.

For Mixed-type applications, the average compression ratios are 1.5/1.7/2.1/2.5 for FPC/BDI/C-pack/BPC, respectively. BPC does worse than C-pack for L.BH (2.2 vs. 3.1) and R.PARTF (4.1 vs. 4.3) but is better by a significant margin for the other applications.

The biggest advantage of BPC is that the DBX transform creates long runs of all-zero sequences that are encoded with ZBP-RLE. 69% of the blocks of R.B+T (INT) have zero bit-plane runs of 32 or longer, enabling compression of more than

992 bits of data (in bit-plane symbols) into a 7-bit coded output symbol. Note that R.B+T originally has poor compressibility (its entropic limit with original symbols is only 2.7:1). While its values in the original symbols are quite diverse, their deltas are mostly similar resulting in zero bit-planes and BPC achieves a high compression ratio of 10.6. R.DWT (INT) has 95% of blocks with zero bit-plane run lengths of 24 or greater. The worst compression achieved by BPC is 1.08:1 from P.TPACF (FP). Most of the blocks of P.TPACF have ZBP runs of length $3-6$ in sign and exponent fields. While this is not a high compression ratio, ZBP-RLE still prevents BPC from requiring more bits than the original stream (FPC/BDI/C-pack increases average block size by 8.6%/0.4%/4.8%, respectively).

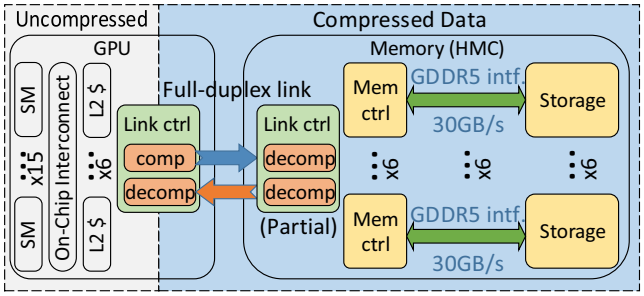
We test two other configurations of BPC that offer marginally better compression: an optimized frequency-based code map and changing the transform granularity to 64 bits for applications with double types.

1) Code map optimization: We now compare our code map to an optimized code map that aims to minimize the number of bits used by statistically analyzing outputs and assigning the shortest codewords to the symbols observed most frequently (similar to Huffman encoding, but targeting the run length encoding and frequent patterns we chose). We use the Rodinia INT applications as the training input and test all applications. Despite including the training set in the tests, the statistically optimized code provides marginal benefits (4.09/1.89/2.45 with the original code map vs. 4.12/1.89/2.46 optimized for INT/FP/Mixed applications, respectively). This is because nearly all the savings are from replacing long sequences of zeros and the specific code map does not matter much.

2) BPC with 64-bit symbols: The other configuration we tested is a BPC layout targeting 16 64-bit symbols instead of 32 32-bit symbols when the data has 64-bit types. The compression ratio of R.LMD, whose data is mostly double, improves from 1.11 to 1.21. The improvement is small because of the inherently high entropy of doubles, smaller bit-planes (31-bit to 15-bit), and the bigger base symbol, which is not compressed well.

B. Hardware Overhead Evaluation

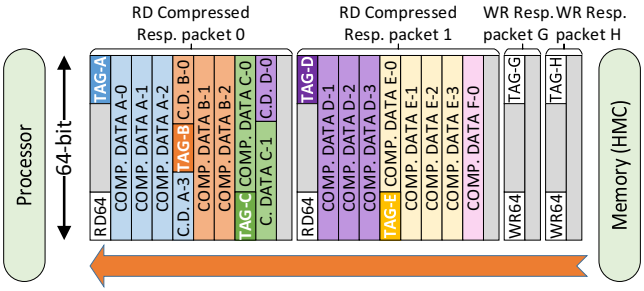
We implement a Verilog model of the compressor and conclude that BPC has a competitive gate count compared to existing compressors. On a compressor core (Figure 3a), we add a buffer block, which receives 256-bit data every cycle from a 64-bit GDDR interface to build a 128B block over 4 cycles. DBX transforms the buffered data in 1 cycle, and each DBP and DBX bit-plane is compressed individually using bit-plane compressors. To save area, we reuse 9 bit-plane encoders over 4 cycles to process 33 bit-planes. As the outputs of the original-symbol encoder, bit-plane encoder, and ZBP-RLE have variable compressed sizes (and can vary at single-bit granularity), careful effort was taken in the design of the concatenation logic to reduce area. The concatenation logic receives up to 9 compressed data blocks per cycle and concatenates them while concurrently shifting previous results for alignment using a 2-stage pipeline. The final compression latency is 11 cycles. If the entire 128B is available within a cycle (e.g., read over a wide interface from a cache), the 4 cycle buffering latency is unnecessary, reducing the overall latency to 7 cycles.



(a) GPGPU + HMC-like-memory system with BPC.

Core Config.	15 Shader Cores, 1400MHz, SIMT Width = 16 2, Greedy-then-oldest Max.
Resources / Core	1536 Threads (48 warps, 32 threads/warp), 48KB Shared Memory
Caches / Core	128B Line Size, 16KB 4-way L1 D-cache, 2KB 4-way I-cache
Shared LLC Config.	128B Line Size, 128KB 16-way LLC / Memory Channel
Memory Config.	6 GDDR5s, 64-bit per Channel, 924MHz (3700Gbps), FR-FCFS
GDDR5 Timing	CL=12, WL=4, tCCD=2, tRRD=6, tRCD=12, tRAS=28, tRP=12, tRC=40, tCDLR=5, tWR=12
Link Config.	1 Pair of Links, Full-width(16b), 90GB/s Raw Bandwidth per link

(b) The GPGPU-Sim simulation configuration (based on an NVIDIA GTX480).



(c) Packed Packets (PP) in the HMC uplink.

Fig. 4: The BPC performance and throughput enhancement evaluation environment.

We synthesize our design using 40nm TSMC standard cells [37] at 800MHz.[1] The overall area is 48,000μm^2, which corresponds to roughly 68K NAND2 gates. The breakdown of the area is 17% for Transform block and its buffers, 5% for 9 DBX encoders and 1 OrigEncoder, 48% for Concatenation, and the rest for pipelining registers, etc. This area overhead is comparable to existing compressors (40K gates for C-pack) and does not represent a large amount of chip real estate considering the billions of gates available in recent processors and the large potential savings in memory capacity and bandwidth. We evaluate the performance of BPC together with the packetized protocol in the next section.

C. System Performance Evaluation

Experiments in Sections V-A and V-B show that BPC provides a high compression ratio with little overhead. We now evaluate the memory throughput and performance benefits this provides below. Before delving into results, we first

[1]A 35% margin in clock period is used to model wire-load delays and other uncertainties.

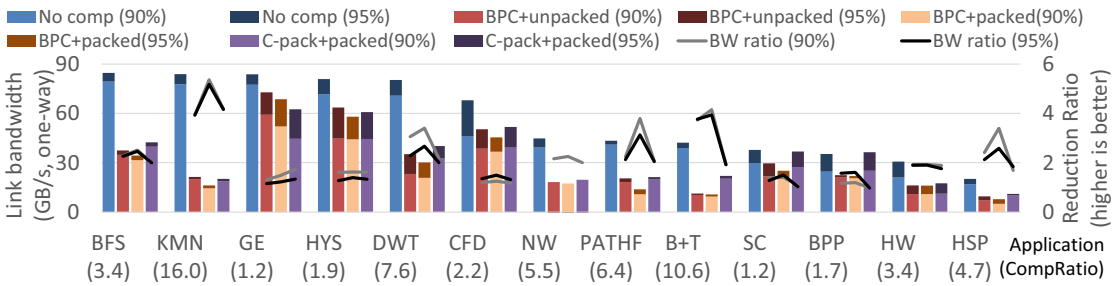


Fig. 6: Bandwidth enhancement through compression: estimate of bandwidth required to achieve 90% and 95% of baseline performance. Fraction: (bars, left axis) and improvement factor: (lines, right axis). Each application is annotated by its BPC compression ratio.

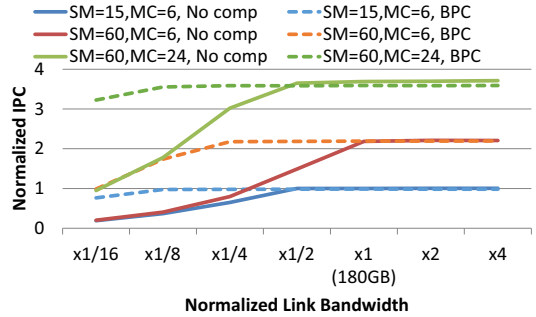


Fig. 5: Simulated IPC on the R.B+T benchmark, varying the link bandwidth, number of SMs, and number of GDDR5 channels. The baseline (GTX480) has 15 SMs and 6 GDDR5 channels with 90GB/s one-way link bandwidth.

describe the evaluation platform, including the details of the proposed packetized interface.

1) Evaluation setup and the packet-packing architecture: We evaluate BPC with GPGPU-Sim v3.2.2 [38] running the benchmarks listed in Table I with the configuration detailed in Table 4b. Figure 4a presents an overview of the system, which integrates BPC into a GPGPU with HMC-like memory. To model HMC-like memory, we replace the 6 GDDR5 channels of the baseline GPGPU (modeled after the NVIDIA GTX480) with a single 6-vault HMC-like memory that has a pair of unidirectional links. The bandwidth of each link is 90GiB/s (in one direction) and the total bandwidth matches that of the 6 GDDR5 channels of the baseline, for which the applications have been tuned. BPC uses variable-length output symbols and provides very high compression for some blocks. HMC packets provide a small 16B access granularity (i.e. FLIT size), reducing internal fragmentation and allowing the bandwidth savings from compression to be realized. We also modify the HMC protocol to eliminate fragmentation entirely by relying on hardware to partially decode compressed packets to compute their length and perform alignment. We test two link configurations: a conventional *Unpacked Packets (UP)* and our proposed *Packed Packets (PP)* protocol. In UP configuration, a compressed block transfers as a single variable-length packet, broken down into a sequence of 128-bit FLITs along with a 64-bit head and a 64-bit tail. Because of internal fragmentation and the head/tail overheads, the effective compression ratio with UP can be significantly worse than that of BPC itself. For example, with a 7:1 compression ratio a 1024-bit packet should require just 146 bits, however, with internal fragmentation and head/tail information the link must transfer 384 bits rather than the full 1152 bits—an effective compression ratio of just 3:1.

To overcome this limit, we propose PP to densely pack compressed data. In Figure 4c, compressed data (A) occupies only a part of a packet. With PP, the left-over space is shared with the following compressed data blocks B, C, and D. Tag information for B and C is stored along with their compressed data to identify the original requests. TAG for D, which spreads over two packets, resides in the head field of the second packet (which has unused bits) to minimize overhead. We do not employ head/tail optimization techniques (e.g. header compression [39], jumbo frame [40]), which can help translate compression ratio into even greater savings by reducing the base transmission overhead.

For a write operation, the link controller (memory controller equivalent) on the processor compresses the data block and transfers the compressed block over the link. The memory receives the compressed data and stores it in memory in its compressed form. With PP, the memory decodes the prefixes of received data before storage (no actual decompression, but decodes the symbols) to determine overall compressed size and separates and aligns packed blocks based on the size.

For read operations, memory fetches the compressed data from storage, partially decodes it to calculate the compressed size and transfers the compressed data over the link. An alternative to partial decoding is to maintain a table of compressed sizes [33]. However, we prefer the higher storage efficiency and simpler overall architecture of partial decoding, despite the 4 cycles of latency it adds. The link controller in the processor receives the compressed data and fully decompresses it before returning it to the cache controller.

In our evaluation, we increase one-way link latency by 6 DRAM cycles to model the packetization and SERDES (SERializer and DESerializer) overheads of the new interface. The transmission delay of the one-way link is assumed to be the same as the GDDR5 bidirectional link of the baseline. With compression, we add 15 more DRAM cycles to the latency: 11 cycles for full compression/decompression and 4 cycles for partial symbol decoding. While BPC can decompress some

blocks with VWC faster than others, we conservatively use 11-cycle latency based on the worst scenario because a slow decompression can delay subsequent decompressions due to a resource dependency. We only compress 128B blocks as such blocks dominate the memory read traffic we observe in GPGPU-Sim. We transfer other block sizes uncompressed.

2) Performance and Memory Throughput Enhancement: BPC + PP can enhance memory throughput and reduce bandwidth requirements. When performance is bound by available memory bandwidth, enhancing memory throughput can also improve performance. We explore the relation between compression, memory throughput, and performance in detail for the R.B+T benchmark (Figure 5) and then show a summary of results for Rodinia benchmarks (Figure 6).

Figure 5 shows the performance of R.B+T with and without BPC (using PP links) as available pin bandwidth is varied and for different levels of peak compute and maximum memory bank parallelism. The results show five important trends. First, the blue lines, which show the behavior with the baseline (15 SMs) with (dashed) and without compression (solid), demonstrate that BPC can maintain application performance even as bandwidth is reduced to $1/8$ of the baseline 180GiB/s and even at $1/16$ of the bandwidth, performance degradation is very small. In contrast, performance without compression degrades sharply starting at a factor of $1/4$. Second, while BPC compresses the memory traffic data of R.B+T by 10.6:1 on average, the throughput is enhanced by only about a 4:1 ratio because of the link overheads (head/tail and write-response packets).

The three other observations relate to performance improvement. Third, when increasing peak performance by $4\times$ (60 SMs), performance is limited by the available bank level parallelism provided by the 6 vaults in our baseline configuration. Performance increases by only $2.2\times$ despite the $4\times$ increase in compute resources (red lines). When increasing bank level parallelism by $4\times$ to 24 cores, performance is increased by $3.6\times$ (green lines). Fourth, in both cases of higher peak performance, BPC is able to deliver performance with lower physical bandwidth. In fact, when enough bank parallelism is provided, performance degrades to only $3.2\times$ baseline with BPC at $1/16$ the baseline bandwidth, but drops all the way to baseline performance at that bandwidth without compression. Fifth, when both bank parallelism and compute parallelism are increased by $4\times$ and available memory bandwidth is high, there is a small penalty from compression: peak performance with compression is 1.4% lower than without compression. We attribute this to the fact that with so much compute parallelism, the pointer-heavy R.B+T graph code is impacted by latency, even with the GPGPU architecture.

To show the benefits of compression in a different way, we evaluate the minimum physical link bandwidth required to meet a certain performance target for Rodinia benchmarks. For each benchmark, we measure its performance (average IPC) with the baseline configuration, with BPC and UP links, with BPC and PP links, and with C-pack and PP links. We gradually reduce the available bandwidth and use linear interpolation to estimate the minimal bandwidth required to match or exceed 90% and 95% of the baseline performance. We exclude R.PART and R.LUD because they do not suffer any performance degradation even at $1/16$ the baseline bandwidth and also exclude R.SRAD and R.LMD, which did not execute to completion in this experiment.

Figure 6 summarizes the results and demonstrates five important points. First, BPC can reduce physical data bandwidth requirements significantly with only small impact on performance. R.BFS, R.KMN, R.GE, R.HYS, R.DWT, and R.CFD utilize more than $2/3$ of the 90GB/s one-way bandwidth without compression and reducing bandwidth significantly degrades performance. With compression, we can achieve similar performance with far less bandwidth. For the most bandwidth-demanding applications, BPC provides a significant performance improvement even with the 15-SM baseline configuration (15% and 18% for R.BFS and R.KMN). Most other applications are affected little ($\pm 2.5\%$), except R.NW which suffers a 5.8% performance degradation because it has limited parallelism and is affected by the added compression latency. Our second observation is that BPC with PP links is an effective combination. On average it can achieve 90% and 95% of the performance for these benchmarks with $2.0\times$ and $1.9\times$ less bandwidth, respectively (line graphs referring to the right axis). BPC +UP and C-pack+PP, on the other hand reduce bandwidth by only $1.8\times$ and $1.7\times$ for the same goals. The third observation is that memory throughput enhancement correlates well with compression ratio and the better compression of BPC realizes higher bandwidth reduction than C-pack. Fourth, PP is effective at reducing link overheads and always achieves higher throughput enhancements than UP. Finally, compression does add memory latency and in some cases it degrades performance. R.NW with compression was unable to achieve 95% of baseline performance regardless of the amount of available BW.

VI. Conclusion

Due to exploding amounts of data and constrained programming models, most data in many-core architectures are stored in arrays. We observe that many of the arrays are homogeneously-typed and design a novel data transformation to improve the inherent compressibility of such data with reduced compressor complexity. The DBX transformation lowers the data entropy and generates long sequences of zeros. The lightweight BPC compressor efficiently encodes the output of the transformation and achieves significantly better compression ratios than prior main memory link compressors. These bit savings from compression can improve system-level performance by lowering link bandwidth requirements or increasing computation throughput on bandwidth-constrained systems.

Acknowledgment

The authors acknowledge the Texas Advanced Computing Center for providing HPC resources. This work is supported, in part, by the following organizations: The National Science Foundation under Grant #0954107 and Intel Corporation.

REFERENCES

[1] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the bandwidth wall: Challenges in and avenues for CMP scaling," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2009.

[2] *DDR2 SDRAM Specification, JESD79-2F*, Joint Electron Device Engineering Council, Nov. 2009.

[3] *DDR3 SDRAM STANDARD, JESD79-3F*, Joint Electron Device Engineering Council, July 2012.

[4] *DDR4 SDRAM STANDARD, JESD79-4*, Joint Electron Device Engineering Council, Sep. 2012.

[5] *Graphics Double Data Rate (GDDR5) SGRAM Standard, JESD212B*, Joint Electron Device Engineering Council, Dec. 2013.

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

[7] M. Thuresson, L. Spracklen, and P. Stenstrom, "Memory-link compression schemes: A value locality perspective," *IEEE Transactions on Computers*, 2008.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[9] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," University of Illinois at Urbana-Champaign, Urbana, Tech. Rep. IMPACT-12-01, Mar. 2012.

[10] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[11] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," in *Symposium on High Performance Chips (HOTCHIPS)*, 2011.

[12] *Hybrid Memory Cube Specification 2.0*, Hybrid Memory Cube Consortium, 2014.

[13] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on VLSI Systems*, vol. 18, no. 8, pp. 1196–1208, Aug. 2010.

[14] 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 *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2012.

[15] D. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[16] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, pp. 379–423, 1948.

[17] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.

[18] F. C. Pereira and T. Ebrahimi, *The MPEG-4 Book*. Prentice-hall, 2002.

[19] A. R. Alameldeen and D. A. Wood, "Frequent Pattern Compression: A significance-based compression scheme for l2 caches," Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison, Tech. Rep., 2004.

[20] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2000, pp. 258–265.

[21] B. Abali, H. Franke, D. E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, and T. B. Smith, "Memory Expansion Technology (MXT): Software support and performance," *IBM Journal of Research and Development*, vol. 45, no. 2, pp. 287–301, March 2001.

[22] M. Kjelso, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," in *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference*, Sep 1996, pp. 423–430.

[23] L. Benini, D. Bruni, B. Ricco, A. Macii, and E. Macii, "An adaptive data compression scheme for memory traffic minimization in processor-based systems," in *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, vol. 4, 2002.

[24] A. Arelakis and P. Stenstrom, "SC2: A statistical compression cache scheme," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[25] D. J. Palframan, N. S. Kim, and M. H. Lipasti, "COP: To compress and protect main memory," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015, pp. 682–693.

[26] J. Kim, M. Sullivan, S.-L. Gong, and M. Erez, "Frugal ECC: Efficient and versatile memory error protection through fine-grained compression," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[27] B. Justice, "NVIDIA Maxwell GPU GeForce GTX 980 Video Card Review." [Online]. Available: http://goo.gl/y2i5Tm

[28] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "Design and evaluation of a selective compressed memory system," in *Proceedings of the International Conference on Computer Design (ICCD)*, 1999, pp. 184–191.

[29] E. G. Hallnor and S. K. Reinhardt, "A compressed memory hierarchy using an indirect index cache," in *Proceedings of the Workshop on Memory Performance Issues (WMPI)*, 2004, pp. 9–15.

[30] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2004, pp. 212–223.

[31] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005, pp. 74–85.

[32] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Linearly Compressed Pages: A main memory compression framework with low complexity and low latency," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2013.

[33] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[34] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance Sparc CMT processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, March 2009.

[35] M. Rabbani and P. Jones, *Digital Image Compression Techniques*. SPIE Publications, 1991.

[36] J. Kim, J. Park, J. Park, and Y. Kwon, "Hybrid image data processing system and method," Jul. 24 2012, US Patent 8,229,235. [Online]. Available: http://www.google.com/patents/US8229235

[37] Taiwan Semiconductor Manufacturing Company, "40nm CMOS Standard Cell Library v120b," 2009.

[38] "GPGPU-Sim," http://www.gpgpu-sim.org.

[39] J. Ishac, "Survey of header compression techniques," National Aeronautic and Space Administration, Tech. Rep. NASA/TM-2001-21154, 2001.

[40] Ethernet Alliance, "Ethernet Jumbo Frames," http://goo.gl/i6ktnh, November 2009.