Exploiting Adaptive Data Compression to Improve Performance and Energy-Efficiency of Compute Workloads in Multi-GPU Systems

Conference Paper · May 2019

DOI: 10.1109/IPDPS.2019.00075

CITATIONS
0

READS
324

4 authors, including:

Mohammad Khavari Tavana
Northeastern University
26 PUBLICATIONS **185** CITATIONS

SEE PROFILE

Yifan Sun
Northeastern University
22 PUBLICATIONS **149** CITATIONS

SEE PROFILE

David Kaeli
Northeastern University
435 PUBLICATIONS **3,949** CITATIONS

SEE PROFILE

Exploiting Adaptive Data Compression to Improve Performance and Energy-efficiency of Compute Workloads in Multi-GPU Systems

Mohammad Khavari Tavana, Yifan Sun, Nicolas Bohm Agostini, and David Kaeli
Dept. of Electrical and Computer Engineering
Northeastern University
Boston, USA

{mkhavaritavana, yifansun, agostini, kaeli}@ece.neu.edu

***Abstract*—Graphics Processing Unit (GPU) performance has relied heavily on our ability to scale of number of transistors on chip, in order to satisfy the ever-increasing the demands for more computation. However, transistor scaling has become extremely challenging, limiting the number of transistors that can be crammed onto a single die. Manufacturing large, fast and energy-efficient monolithic GPUs, while growing the number of stream processing units on-chip, is no longer a viable solution to scale performance.**

GPU vendors are aiming to exploit multi-GPU solutions including connecting multiple GPUs in the single node with high bandwidth network (such as NVLink) or exploiting Multi-Chip-Module (MCM) packaging, where multiple GPU modules are integrated in a single package. The inter-GPU bandwidth is an expensive and critical resource for designing multi-GPU systems. The design of the inter-GPU network can impact performance significantly. To address this challenge, in this paper we explore the potential of hardware-based memory compression algorithms to save bandwidth and improve energy efficiency in multi-GPU systems. Specifically, we propose adaptive inter-GPU data compression to efficiently improve both performance and energy efficiency. Our evaluation shows that the proposed optimization on multi-GPU architectures can reduce the inter-GPU traffic up to 62%, improving system performance up to 33% and save energy spent powering the communication fabric by 45%, on average.

***Index Terms*—Multi-GPU system, Multi-Chip-Module, Compression algorithms, Bandwidth management, Performance.**

I. Introduction

GPU-based computing has become an attractive path to achieve high performance and energy-efficient computation in a range of applications, including, big data analytics, machine learning, natural language processing, and computer vision [13]. To efficiently handle the unprecedented volume of data being generated today, we need to exploit scalable and energy-efficient computational infrastructure.

The end of Dennard's Scaling has signaled the end of enjoying the significant performance per watt improvements of simply moving to the next technology node [9]. CMOS transistor scaling has reached its limit. The number of transistors that can be crammed onto a single die grows slowly, hampered by several technical challenges. While we can manufacture a large energy-efficient monolithic GPU by increasing the number of compute units, this path to increasing the capacity of a GPU is becoming increasingly challenging. Given the physical constraints on the maximum die size [38], and resulting lower yields for large die sizes [19], growing the size of a monolithic GPU has become technically infeasible or too costly.

We believe that next-generation technologies will shift away from increased chip density through smaller geometries, and instead, invest in features that support new device architectures and programming models. For example, a Multi-Chip Module (MCM) is a state-of-the-art technology, that has been promoted by industry to increase the number of transistors and resources on CPUs [24] and GPUs [6]. When using an MCM, multiple CPUs or GPUs are assembled onto a single device and encapsulated in a single package. Multiple dies are interconnected using an interposer, resulting in a single high-density integrated circuit that can overcome chip-area constraints. An MCM can improve both bandwidth and latency of a communication fabric by reducing the length of the interconnection, capacitance loading, and crosstalk disturbance [6].

A multi-GPU programming model that leverages message passing [11], [18] enables the programmer to employ more than one GPU to run their application. This approach requires significant programming effort to re-write existing GPU applications, which reduces the pace of adoption of workloads to newer multi-GPU architectures. However, run-time drivers can transparently leverage multiple GPUs by presenting them to the programmer as a single large GPU [21], [26].

Given that the GPU's data can be located in any of the GPUs' on-board memory, the run-time system will need to consider a Non-uniform Memory Access (NUMA) model when trying to optimize the performance of a multi-GPU application [23], [27]. Achieving scalable multi-GPU performance is not trivial, as the communication fabric that interconnects GPUs will potentially be the main bottleneck in many applications [6], [20], [27], [43], [44], especially given the cost of data movement and synchronization overhead.

In a multi-GPU system, inter-GPU bandwidth is a valuable resource. Previous work has reported that many GPU compute

workloads possess a good amount of compressibility (up to 70%) [22]. Therefore, there can be significant saving if we can compact data before transferring it to other GPUs, saving bandwidth and energy consumption. To the best of our knowledge, this paper is the first work that investigates the potential of data compression in the context of multi-GPU architectures. In this paper we make the following contributions:

- We show that there is a significant opportunity in many GPU compute workloads to compress data in a multi-GPU system. Exploiting data compression across GPUs, we achieve up to a 53% improvement in performance, and up to a 45% reduction in energy consumption of the data communication fabric.
- We provide a comprehensive quantitative comparison of state-of-the-art hardware-based data compression algorithms in the context of a multi-GPU architecture. Extensive characterization and evaluation are performed to analyze inter-GPU data communication.
- We propose an adaptive compression scheme that decides when to use compression, as well as which compression algorithm is most beneficial, during different phases of application execution. Our scheme adapts based on the compressibility of the data, as well as the compression/decompression latency, and avoids performance and energy overheads if data is not compressible.

The rest of this paper is organized as follows: In Section II, we provide background on different types of multi-GPU systems. In Section III, we discuss hardware-based memory compression algorithms and compare them based on latency, energy consumption and hardware overhead. In Section IV, we characterize inter-GPU data access patterns and describe our adaptive compression schemes. Our simulation strategy and results are presented in Section VI and Section VII, respectively. Section VIII reviews related work, and finally, Section IX summarizes the lessons learned in this paper.

II. Multi-GPU Systems

To scale performance beyond a single GPU, multi-GPU systems have been proposed [6], [20], [27], [43], [44]. Multi-GPU systems can speed up applications via package-level, board-level, or system-level GPU integration. The design goal of a multi-GPU system is to achieve the performance of a single large monolithic GPU. The utility of such a system relies on run-time software to make the underlying multiple GPUs appear as a single logical GPU to the programmer [6], [11], [21], [26]. Therefore, applications can exploit multiple GPUs with more resources, without imposing additional programming effort.

Another enabler for multi-GPU systems is a high-bandwidth interconnection fabric that can provide efficient communication across GPU modules, providing good performance when accessing shared data across multiple GPUs. The bandwidth and energy consumption per bit improve when communication changes from across different nodes, to on-chip or on-die. GPU communications can be categorized as following:

On-chip communication: If multiple GPUs are integrated on a chip, the bandwidth for communication can reach up to 10's of TB/s, incurring low energy overhead. However, today this is not a viable solution due to the physical constraints on the maximum die size [38], and a very low yield of large dies [19].

Inter-die communication: When GPUs are built in modules on separate dies and interconnected together through a common package, the architecture is called a Multi-Chip-Module GPU (MCM-GPU). On-package or inter-die communication results in major power savings of roughly 1-2 pJ/b, with 100 GB/s to 1 TB/s total bandwidth [7]. This emerging MCM architecture has been proposed as an attractive approach to achieve scalable performance in future GPU systems [6].

Inter-package/board/socket communication: Multiple GPU packages can communicate on separate sockets [27], or single/separate boards [20]. In current multi-GPU systems, GPUs are interconnected through PCIe channels or higher bandwidth interconnect technology such as NVLINK [17]. The bandwidth of inter-package communication is $\approx$ 100 GB/s to 200 GB/s, incurring $\approx$ 10 pj/b to 12 pJ/b of energy cost [6], [7], [17], [27].

Inter-system/node communication: Multiple GPUs can be interconnected through an Infiniband interconnect. The bandwidth available on a state-of-the-art inter-node communication fabric is in the range of 12.5 GB/s to 25 GB/s, assuming four lanes per port. The performance of many GPU applications is limited by the bandwidth between nodes. Sending data over this network is not very energy efficient ($\approx$ 250 pj/b) [6].

These multi-GPU systems can provide high aggregate throughput as compared to single GPU system. However, there still remains a significant performance loss for applications that perform any amount of inter-GPU communication. In this paper we evaluate the potential of memory compression algorithms to save on multi-GPU bandwidth and improve energy-efficiency in future multi-GPU systems.

III. Hardware-based Compression Algorithms

In this work, we consider adopting three existing memory compression schemes to compress data exchanged between GPUs. These compression algorithms have practical hardware implementations, are efficient to compress small data sizes (i.e., from a single *word*, to an entire cache line), and are suitable for hardware-based memory compression.

All of the compression algorithms proposed for compacting data in memory are tailored to perform best in the presence of specific data patterns [3], [12], [31], [33]. Data patterns tend to repeat in applications, mainly due to data initialization, input types, and the underlying algorithm used to transform the data. In the following, we discuss different data access patterns which are common in GPU applications.

A. Common Data Patterns

Zero Words. A zero data pattern is the most commonly observed pattern in many of applications [31]. The reasons behind this fact include: the initialization of variables to zero, the initialization of null pointers, and the zeroing out of pages

before memory allocation. Moreover, some algorithms exploit sparse data and sparse data structures. For example, many neural networks use non-linear activation functions that clamp all negative values to zero. It is been shown that for typical data sets, 50-70% of the output data values after the activation layers are changed to zero [28]. Due to the frequent appearance of zeros in memory, most memory compression algorithms treat zeros as a special case [3], [12], [31], [33], [35].

Repeated Words. Repeated words in memory is another common pattern observed in applications [37]. Some applications, such as numerical optimization, use a common value to initialize large arrays being allocated during program execution. Additionally, in image processing applications, adjacent pixels of a image will often have the same assigned color value, which leads to frequent repeated values in memory.

Narrow Words. Many variables in applications are over-provisioned, while having a small number of values throughout the execution of a program. Programmers tend to provision more than required to cover the worst-case or to reduce their programming effort. Data alignment in memory is another source of narrow words imposed by compilers or the operating system. Narrow words are common data patterns in applications, and can be exploited to improve performance and reduce energy consumption [10], [16], [41].

Low Dynamic Range Pattern. As opposed to the previously discussed patterns, that are applicable to a single word, this pattern emerges from common behavior across all of the data in a memory block or cache line. When the differences between words stored in a cache line are small, the range of data values will be small, making them a good target for compression [31]. In memory, a data structure, such as an array of pointers, is a very good example of this pattern, where the differences between consecutive words is equal to the data type size of the data referenced by the pointers. In image processing applications, typically there is low dynamic range in consecutive pixels residing in a given cache line.

Spatial Similarity. In some applications, there is correlation between the data values stored within a cache line, and in most cases, the values stored are similar to each other. Compression algorithms can keep track of the most frequently appearing byte or word values, and encode them in smaller values to compact the data. For instance, applications that work with a sequence of vector (or scalar) values that depend on time (e.g., stock price time series, sensor value time series) can be a good example of a spatially similar pattern.

Note that the classes of patterns described above are not mutually exclusive. In a cache line there can be multiple narrow words with low dynamic range values, while the words are also partially similar to one another. It is key to design an effective compression algorithm that can dynamically detect a pattern and encode the values to reduce the size of a cache line effectively.

B. Compression Algorithms

In this paper, three different memory compression algorithms (i.e., FPC, C-Pack+Z, and BDI) are employed, where each covers a subset of the data patterns discussed in Section III-A. Table I shows the data patterns supported by each compression algorithms. In the following, we briefly describe how each compression algorithm performs and compare results against the other schemes.

TABLE I: Supported data patterns by different memory compression algorithms

Data Patterns	FPC	BDI	C-PACK+Z
Zero Word/Block	✓	✓	✓
Repeated Word	✓	✓	✓
Narrow Word	✓	Partial	Partial
Low Dynamic Range	✗	✓	✗
Spatial Similarity	✗	✗	✓

FPC. Frequent Pattern Compression or FPC [3] encodes seven patterns of data that are most frequently seen in applications, using a smaller number of bits to represent them. The focus of FPC is to take advantage of many small values that only need a small number of bits to represent them. FPC is a static compression algorithm and performs data compression on a word-by-word basis. FPC can detect patterns such as zero words, repeated words, and five different kinds of narrow words. It adds a 3-bit prefix to each word (32-bit or 64-bit words) to distinguish between the encoded classes. FPC can compress a 512-bit cache line in 3 cycles and decompress it in 5 cycles, assuming 12 FO4 gate delays per cycle [3].

C-Pack. Cache Packer (C-Pack) [12] is another hardware-based memory compression algorithm which is implemented using a dictionary. It detects and compresses frequently appearing patterns, such as zero words and repeated words. C-Pack uses a small dictionary with 16 entries and populates the dictionary by progressing word by word in the data block. It is not necessary to attach the dictionary to the associated compressed data because the dictionary can be generated on-the-fly, based on the compressed block. C-Pack exploits partial matching to find any similarities between the processed data in a cache block. Partial matches find similarity between data and the dictionary at a half-word, three byte, and word granularities. If no match is found in the dictionary, the word is inserted in the dictionary. To better compress the data, we use a modified version of C-Pack (C-Pack+Z) that also detects zero blocks, saving more bits in a compressed format [34]. C-Pack+Z compression and decompression latencies are 16 and 9 cycles, respectively.

BDI. The Base-Delta-Immediate (BDI) compression algorithm [31] is tailored to exploit low dynamic range patterns. BDI explores compression opportunities at a cache line granularity and stores data as one or more base values, along with an array of relative differences to the base value. BDI provides a high compression ratio and low decompression latency. BDI's efficient implementation exploits two bases, one is the first value (a 2, 4, or 8 byte value) of the cache line, and the second base is the implicit base with a zero value (i.e., no need to store this base because it is always zero).

The second base enhances BDI's efficiency significantly since it helps to compress narrow words in the cache line. The

TABLE II: Pattern Encoding for Different Compression algorithms.

FPC					
Prefix	Pattern#	Encoded Pattern	Original Data	Compressed Data	Total Data Size (data + metadata)
000	1	Zero block	Z_{512}	-	0 + 3 bits
001	2	Zero word	Z_{32}	-	0 + 3 bits
010	3	Word with repeated bytes	$N_8N_8N_8N_8$	N_8	8 + 3 bits
011	4	4-bit sign-extended	$X_{28}N_4$	N_4	4 + 3 bits
100	5	one byte sign-extended	$X_{24}N_8$	N_8	8 + 3 bits
101	6	halfword sign-extended	$X_{16}N_{16}$	N_{16}	16 + 3 bits
110	7	halfword padded with zero halfword	$N_{16}Z_{16}$	N_{16}	16 + 3 bits
111	8	two halfword, each a byte sign-extended	$X_8N_8X_8N_8$	N_8N_8	16 + 3 bits
N/A	9	Uncompressed	N_{512}	N_{512}	512 + 0 bits
C-Pack+Z					
Prefix	Pattern#	Encoded Pattern	Original Data	Compressed Data	Total Data Size (data + metadata)
00	1	Zero block	Z_{512}	-	0 + 2 bits
01	2	Zero Word	Z_{32}	-	0 + 2 bits
10	3	New word (insert into the dictionary)	N_{32}	N_{32}	32 + 2 bits
1100	4	Word is matched	M_{32}	D_4	4 + 4 bits
1101	5	Halfword is matched	$M_{16}N_{16}$	D_4N_{16}	20 + 4 bits
1110	6	Narrow word with one byte significance	$Z_{24}N_8$	N_8	8 + 4 bits
1111	7	Three bytes are matched	$M_{24}N_8$	D_4N_8	12 + 4 bits
N/A	8	Uncompressed	N_{512}	N_{512}	512 + 0 bits
BDI					
Prefix	Pattern#	Encoded Pattern	Original Data	Compressed Data	Total Data Size (data + metadata)
0000	1	Zero block	Z_{512}	-	0 + 4 bits
0001	2	Repeated words	$N_{64} \times 8$	N_{64}	64 + 4 bits
0010	3	Base: 8 bytes, Δ: 1 byte	for all i (B_{64}-Δ_8^i or Z_{64}-Δ_8^i) $\leq$ CAP(Δ_8)	(B_{64}+Δ_8^i) $\times$ 8	128 + 12 bits
0011	4	Base: 8 bytes, Δ: 2 bytes	for all i (B_{64}-Δ_{16}^i or Z_{64}-Δ_{16}^i) $\leq$ CAP(Δ_{16})	(B_{64}+Δ_{16}^i) $\times$ 8	192 + 12 bits
0100	5	Base: 8 bytes, Δ: 4 bytes	for all i (B_{64}-Δ_{32}^i or Z_{64}-Δ_{32}^i) $\leq$ CAP(Δ_{32})	(B_{64}+Δ_{32}^i) $\times$ 8	320 + 12 bits
0101	6	Base: 4 bytes, Δ: 1 byte	for all i (B_{32}-Δ_8^i or Z_{32}-Δ_8^i) $\leq$ CAP(Δ_8)	(B_{32}+Δ_8^i) $\times$ 8	160 + 20 bits
0110	7	Base: 4 bytes, Δ: 2 bytes	for all i (B_{32}-Δ_{16}^i or Z_{32}-Δ_{16}^i) $\leq$ CAP(Δ_{16})	(B_{32}+Δ_{16}^i) $\times$ 8	288 + 20 bits
0111	8	Base: 2 bytes, Δ: 1 byte	for all i (B_{16}-Δ_8^i or Z_{16}-Δ_8^i) $\leq$ CAP(Δ_8)	(B_{16}+Δ_8^i) $\times$ 8	272 + 36 bits
N/A	9	Uncompressed	N_{512}	N_{512}	512 + 0 bits

Z: Zero bits, **X**: All ones or all zeros bits, **N**: No any specific pattern, **M**: Matched Patterns found in the dictionary, **D**: indirection to the entry of the dictionary, **B**: Base value, Δ: delta value, **CAP (Δ)**: capacity of Δ with specific number of bits. The subscripts show the number of bits each symbol represents. Total Data size shows the size of the compressed data.

TABLE III: Cost and overhead of different memory compression algorithms in a 7nm technology, and at 1 GHz frequency.

Compression Scheme	Compression Latency (Cycles)	Decompression Latency (Cycles)	Total Area (μm^2)	Compressor Power (mW)	Decompressor Power (mW)	Energy Consumption‡ (pJ)
FPC†	3	5	4428	4.6	4.6	36.9
BDI	2	1	162	0.6	0.2	1.3
C-Pack+Z	16	9	766	1.8	1.3	40.0

† For FPC, combined power dissipation was reported by Das et al. [14]. We assume the contributions of the compressor and decompressor equally. ‡ Combined energy consumption for the compression and decompression of 512-bit data is reported.

delta values, which are represented relative to the implicit zero base, can be thought of as immediate values which match the name of this compression algorithm. A bit mask needs to be stored along with the data, indicating whether the saved base or the implicit zero base should be used with the data word.

BDI is tailored to perform compression at a cache block granularity. It performs efficiently whenever blocks contain a set of pointers or values with low dynamic ranges (i.e., with help of the first base), a set of narrow values (i.e., with the help of the second base), or the combination of both. The ability of BDI to compress narrow words is limited as compared to FPC. Narrow words in a cache block need to follow the same behavior to be compressible with the BDI algorithm. For instance, if one of the words in a cache block is a narrow word with a sign-bit extension and another is narrow word without sign extension, BDI cannot compress the cache block.

Table II shows how different memory compression algorithms encode the incoming data to their compressed format. The total size of the compressed data depends on which pattern is detected by the compression algorithm and the associated metadata. If space is not saved after applying a compression algorithm, the data is transmitted in its uncompressed form. Therefore, one extra bit is needed to distinguish between compressed and uncompressed data. Note that the reported data sizes for FPC and C-Pack+Z are at a word granularity, but for BDI at a cache line granularity (512 bit).

Each of the compression algorithms discussed here provides different benefits and each comes with its own costs and overhead to the system. We report on compression/decompression latencies, total area overheads, compres-

TABLE IV: Benchmarks Description

Benchmark	Abbrev.	Description
Advanced Encryption Standard	AES	256-bit encryption AES involves a large number of bitwise and shifting operations.
Bitonic Sort	BS	Sorting algorithm with a irregular access pattern, suits the GPU's massively parallel architecture.
Finite Impulse Response Filter	FIR	A fundamental algorithm from the digital signal processing domain which has adjacent access pattern.
Gradient Descent	GD	Important algorithm with gather pattern used in optimization problems such as neural networks training.
KMeans	KM	An important clustering algorithm widely used in unsupervised machine learning applications.
Matrix Transpose	MT	A fundamental matrix operation that is used in many scientific and engineering applications.
Simple Convolution	SC	An important operation in convolutional neural networks and image processing applications.

sor/decompressor power dissipation, as well as the total energy consumption for the different compression algorithms in Table III. The C-Pack+Z implementation [34], FPC implementation [14], and BDI implementation [30] are synthesized using 32nm, 45nm, and 65nm libraries, respectively. We scaled all the overheads to a 7nm technology, assuming a 1 GHz frequency, with constant voltage scaling [32] which is a reasonable assumption for the post-Dennard scaling era.

IV. Inter-GPU Data Characterization

In this section, we first describe our selected benchmarks for multi-GPU computation. Next, we characterize the inter-GPU data communication showing the potential of using memory compression algorithms to save bandwidth. Finally, we show that there is similarity in the entropy of data during consecutive inter-GPU transactions in multi-GPU benchmarks, which we exploit to improve the compression ratio and energy consumption of a multi-GPU system.

A. Multi-GPU Benchmarks

To the best of our knowledge, the OpenCL benchmarks developed for evaluating multi-GPU benchmarks are limited. Therefore, we selected some of the previous single GPU applications from public-domain libraries and benchmark suites that possess diverse collaborative execution patterns to evaluate our multi-GPU system. All of the benchmarks are modified with new OpenCL kernels supporting multi-GPU execution.

Table IV shows the name, abbreviation, and a brief description about each benchmark. BS, MT, SC are ported from the AMDAPPSDK 3.0 [1], while AES, FIR, KM are ported from the HeteroMark benchmark suite [40]. We developed the GD benchmark from scratch. In GD, data is partitioned into mini-batches and distributed among GPUs. When the gradient calculation for each batch is finished, the GPUs communicate in order to average out the results.

B. Data Characterization

The compression algorithms targets various patterns present in data. One algorithm may perform better in an application or a phase of an application as compared to another. This section aims at characterizing inter-GPU data from different perspectives, and provides insight into the efficiency of the different compression algorithms.

Our target multi-GPU system has four interconnected GPUs. The details of our simulation methodology are presented in Section VI. All inter-GPU data is transferred on a cache line granularity. We characterized various parameters associated with these transfers. Table V provides the number of remote read and write accesses, along with the entropy of the data (captured at a byte-level granularity) that is transferred between GPUs.

TABLE V: Inter-GPU Data Characteristics

Bench.	Read (Kilo)	Write (Kilo)	Entropy	Compression Ratio		
				BDI	FPC	C-Pack+Z
AES	3,522	49	0.96	1.00	1.03	**1.04**
BS	1,336	1,321	0.02	9.6	31.68	**37.10**
FIR	1,945	98	0.50	**2.41**	1.00	1.73
GD	990	198	0.46	1.26	**1.38**	1.20
KM	4,129	203	0.11	1.37	5.63	**7.79**
MT	3,146	3,146	0.29	2.84	**3.10**	2.69
SC	5464	49	0.49	**2.69**	1.03	1.82

The benchmarks studied show a diverse range of entropies, from a very small value for BS (0.02), to a large value for AES (0.96) (expected for an encryption algorithm). The less entropy, the more opportunity for a compression algorithm to compact data values. In general, one of the compression algorithms outperforms the others, on average, when compressing the inter-GPU data, but there is not a one-size-fits-all algorithm (it is shown in **bold** text, in Table V). For instance, the compression ratio (i.e., size of the uncompressed data over the size of the compressed data) is significantly higher when using C-Pack+Z instead of FPC for the KM benchmark.

As expected, the AES benchmark is practically not compressible at all, while BS shows an attractive opportunity. The input size for BS is rather small compared to the other benchmarks (i.e. $\approx$ 131,000 number). However, a very large number of kernels are launched to perform the computation, and most of the communication is due to address calculations and related information, such as kernel sizes. The transferred data contains a lot of zeros. In contrast, for the AES benchmark the inter-GPU data is almost random, and there is a very small opportunity for data compression.

By studying how individual data value patterns can be exploited by each compression algorithm, we are able to develop a better understanding of the benefits of each algorithm (see Table VI). In Table VI, (#),% indicates the pattern number (which is defined in Table II), and the percentage of each pattern.

The compression algorithms only exploit two or three detected patterns in all the benchmarks. However, the detected patterns are different in each benchmark. In cases where a compression algorithm can compress the data, if we exclude the zero blocks, pattern numbers 4, 5, and 7 are utilized more in FPC, and pattern numbers 5, 6, 7 are utilized more in C-Pack+Z and BDI.

TABLE VI: Three most detected patterns by compression algorithms

Algorithm	Pattern Contribution	AES	BS	FIR	GD	KM	MT	SC
FPC	(1^{st}), %	(9), 96%	(2), 52%	(9), 99%	(7), 66%	(2), 86%	(5), 83%	(9), 94%
	(2^{nd}), %	(2),<3%	(1), 35%	(7),<1%	(9), 33%	(9), 12%	(4), 15%	(6), 5%
	(3^{rd}), %	(1),<1%	(9), 11%	(8),<1%	(2), <1%	(1), 1%	(2), 1%	(8),<1%
C-Pack+Z	(1^{st}), %	(3), 95%	(2), 52%	(7), 76%	(3), 63%	(2), 86%	(6), 98%	(7), 87%
	(2^{nd}), %	(2), 4 %	(1), 35%	(5), 17%	(7), 28%	(3), 6%	(2), 1%	(5), 6%
	(3^{rd}), %	(1),<1%	(3), 11%	(3), 6%	(4), 4%	(7), 4%	(3),<1%	(3), 6%
BDI	(1^{st}), %	(9), 99%	(1), 89%	(6), 74%	(9), 66%	(9), 56%	(6), 99%	(6), 91%
	(2^{nd}), %	(3), <1%	(9), 9%	(7), 25%	(6), 29%	(5), 25%	(5),<1%	(7), 8%
	(3^{rd}), %	NA	(6),<1%	(8),<1%	(7), 4%	(1), 18%	(9),<1%	(9),<1%

(#),%: # indicates the order of the major pattern contributors, and % indicates the percentage of the contributor.

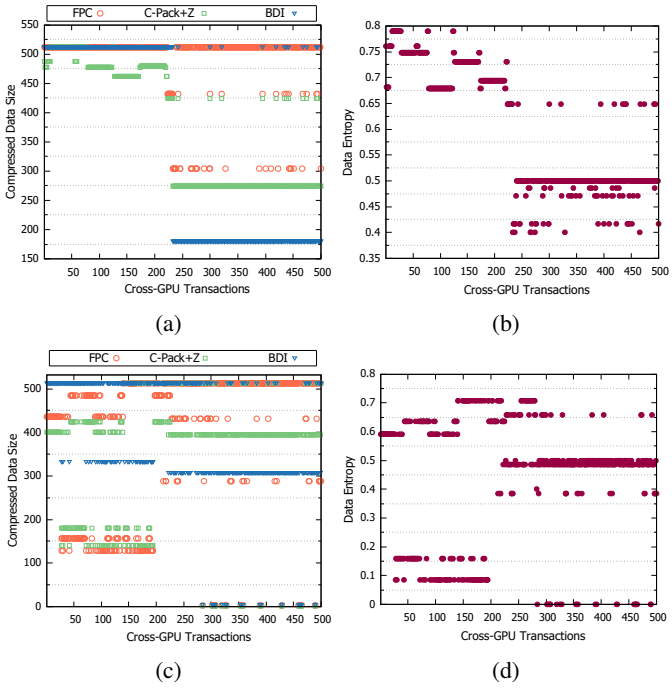

Fig. 1: Performance of compression algorithms and corresponding entropy for 500 inter-GPU data accesses for SC benchmark (a), (b). Performance of compression algorithms and corresponding entropy for 500 inter-GPU data accesses for FIR benchmark (c), (d).

C. Temporal Similarity in Inter-GPU Data Entropy

From Fig. 1b and Fig. 1d, we can see that there is some similarity in the entropy of the data communicated over time between GPUs. The main reason for this behavior is that many GPU kernels show temporal regularity in the computations performed, as well as regularity in the data types used by the kernels. For example, the input to the SC benchmark is a set of images (RGB channels). During its first phase, SC performs adjacent memory accesses in two dimensions, uniformly. Later, the image to be convolved is partitioned across multiple GPUs, and, finally, each GPU needs to access a remote partition to read the input pixels outside its own partition boundaries.

The entropy of 500 consecutive inter-GPU data transfers (each 64 bytes), analyzed at a byte-level granularity, is shown in Fig. 1b and Fig. 1d for SC and FIR benchmarks, respectively. The corresponding compressed data sizes using different compression algorithms are provided in Fig. 1a and Fig. 1c.

Fig. 1 shows there is temporal correlation in the data entropy values over consecutive inter-GPU transactions. For a given data entropy, compression algorithms perform differently depending on the data pattern. But as expected, there is a strong correlation between entropy of the data and the performance of a compression algorithm.

Fig. 1a shows that the bus transactions for the SC benchmark can be clearly broken into phases. In the first phase, C-Pack+Z outperforms the other compression algorithms, whereas in the second phase, BDI outperforms the other compression algorithms. A very similar behavior is observed in the FIR benchmark. As Fig. 1c shows, in the first phase, FPC and C-Pack+Z are able to compress data, while BDI cannot, while in the second phase, BDI outperforms the others.

V. Adaptive Compression Algorithm

Motivated by the fact that different compression algorithms may have different compression ratios over time, we need to find the best compression algorithm, not only for the GPU kernel, but for each individual inter-GPU communication. It is possible to execute the three compression algorithms concurrently and transfer the data with the one that leads to the minimum data size. However, this approach always adds to the maximum latency of the compression algorithm used for the inter-GPU communication. The extra latency and increased energy consumption imposed by executing all three compression algorithms may outweigh all the benefits that are achieved from inter-GPU link compression.

Due to the entropy behavior of consecutive bus transactions, it is efficient to only run the three compression algorithms for a small sample of the data and then decide which algorithm to use for the future incoming data. We show different phases of our adaptive compression scheme in Fig. 2, selecting the best compression algorithm based on the outcome of the *sampling phase*. After the sampling phase, one of the algorithms is selected based on a predefined criteria (i.e., energy consumption, compressed data size, energy-delay product, etc.). The

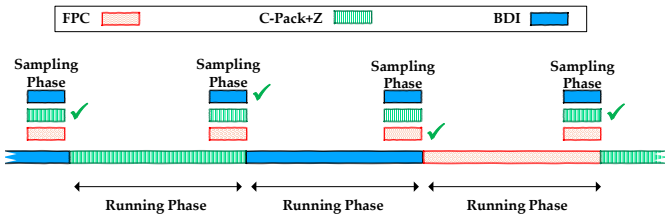

Fig. 2: Sampling and running phases of adaptive compression scheme that select one of the compression algorithms with respect to the outcome of the sampling phase.

selected algorithm is kept for compressing data at least until the completion of the *running phase*.

The durations of the sampling and running phases are set using a fixed numbers of inter-GPU transactions and do not depend on the number of cycles. To improve the fidelity of the selection logic and reduce the noise, we sample seven data transfers and select the compression algorithm that outperforms the others in at least three of the samples (i.e., outcome voting). In this work, we evaluate 7 samples for every 300 transactions, achieving a balance between sampling accuracy and efficiency.

As each compression algorithm has its own overhead (in terms of latency and energy consumption), selection logic that only considers the resulting size of the data will not perform efficiently. To balance bandwidth savings, performance and energy savings, an adaptive algorithm is adopted that is guided by the following penalty function:

$$P = N + \lambda(L_C + L_D), \quad (1)$$

where N is the size of the compressed data in bits, and L_C, and L_D are the latency of the compression and the decompression in cycles, respectively. The selection logic chooses a compression algorithm that minimizes the penalty. The λ parameter allows us to trade-off bandwidth for performance. A lower λ value favors compression algorithms that can compress data to be the smallest in size. In other words, a smaller λ value guides the system to select the algorithm that offers the best compression ratio, and performs best for a system with a slow interconnect. A larger λ value suggests a faster compression algorithm, and benefits cases where communication latency is critical to the overall performance.

We select the lambda value statically, based on empirical results across different λ values, thereby avoiding the additional complexity of dynamic selection. However, the λ value can be configurable to allow driver developers or system administrators to optimize the system based on different constraints and goals. Once a λ value is selected, it cannot be changed by hardware during execution of an application.

In inter-GPU communication messages, we add 2 extra bits to denote which compression algorithm is used (see VI-B for more details). We reserve the value 0 for "not compressed" and the rest of the bit values to denote the 3 other compression algorithms. When a GPU RDMA receives a data packet with a 0-valued compression field, it simply bypasses the decompression stage, thereby saving on decompression cycles.

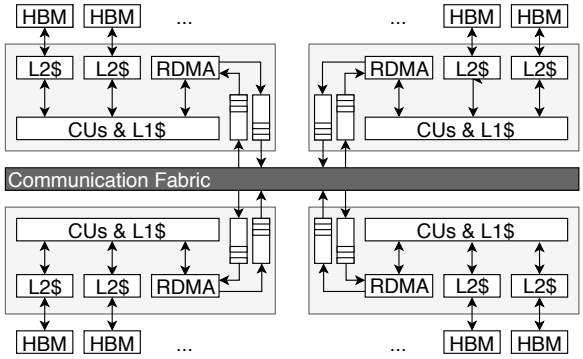

Fig. 3: Multi-GPU system.

In addition, as the metadata is attached to each packet, GPUs do not need to exchange information and can use different compression algorithms from GPU to GPU. This solution is scalable, and if we need to support more compression algorithms, we can add extra bits.

The complexity of the selection logic is negligible, as the second part of the equation is constant for each algorithm, and only the N value is changing for data communications. In the case where data is not compressible, an adaptive algorithm bypasses the compression circuitry to not only save extra energy, but also avoids the extra compression latency which would be added unnecessarily. This configuration is maintained until the next sampling phase.

Note that implementing more than one compression algorithms is not necessary to exploit the adaptive compression scheme. The proposed scheme still works, even when only one compression algorithm has been integrated in the GPUs. In this case, the goal is changed to turning on and off the compression circuit, based on whether performance can be improved or energy reduced.

VI. Experiment Methodology

In this section we describe our simulation framework, assumptions and parameters. Next, evaluate and discuss the results.

A. GPU Modeling

We model a multi-GPU system (as shown in Fig. 3) that is composed of 4 AMD R9-Nano GPUs [25], running the third generation graphics core next (GCN3) instruction set architecture (ISA). When the Compute Unit (CU) needs to read from or write to the memory, whether the access is to a remote or local GPU, the CU sends the read/write request to the per-CU L1 cache. In the case of an L1 miss, the L1 cache checks if the the data should be on the local GPU or on an remote GPU. Local GPU requests are dispatched to the L2 caches, while remote accesses are sent to the Remote Direct Memory Access (RDMA) engine. RDMA engine routes requests to the GPU that owns the data through the central communication fabric.

To model the multi-GPU system, we use a publicly available multi-GPU simulator, MGSim [39]. The simulator has been validated against AMD Radeon R9 Nano GPU hardware [25].

TABLE VII: Simulation Parameters

Parameter	Value
Number of CUs	64 per GPU
L1 Vector Cache	16 KB, 4-way
L1 Scalar Cache	16 KB, 4-way, shared by 4 CUs
L1 Instruction Cache	32 KB, 4-way, shared by 4 CUs
L2 Cache	8 units per GPU, 256 KB each, 16-way
DRAM	8 channels per GPU
Cache Line Size	64 Byte
Fabric Bandwidth	160 Gb/s

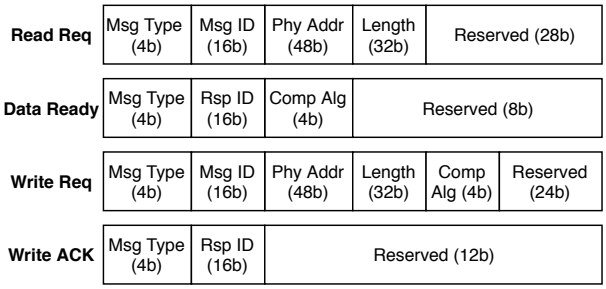

Fig. 4: Inter-GPU Communication Message Header Format.

We configure simulator parameters (listed in Table VII) to best match the R9 Nano GPU hardware, and configure the inter-GPU interconnect with a 160 Gb/s bus structure.

We layout the memory by interleaving 4 KB pages over the 32 memory controllers (8 for each GPU), utilizing all of the GPU channels effectively. Workgroups are scheduled using a round robin scheme across all the CUs on all 4 GPUs.

B. Inter-GPU Communication Modeling

GPUs access remote memory with four types of messages, including *Read*, *Data-Ready*, *Write*, and *Write-ACK*, as shown in Figure 4. Note that no matter what compression algorithm is used, we only compress the payload and the header is preserved The four types of messages are differentiated with the first 4 bits of data. In the requests (Read and Write), we use a 16-bit field to represent the sequence number of the request. The responses (Data-Ready and Write-ACK) can embed this sequence number in the field *Rsp ID*, so that the response receiver knows which original request matches the respond. This sequence number enables out-of-order request fulfillment. For Read and Write requests, we also add physical address and the length fields, indicating what data the request wants to read or write. Data-Ready and Write messages has payloads, and hence, needs to specify what compression algorithm to use in the *Comp Alg* field. We reserve the value `0` for "not-compressed", so that we can avoid compression if the compression algorithm cannot reduce the payload size. Finally, we reserve extra bits to align the payload with a full byte.

All the inter-GPU communication messages, as well as the CPU-GPU communication messages, are sent over a PCIe-like fabric. We model the fabric as a bus and only one message can be sent at a time. Each message takes an integer number of cycles to transfer and no two messages can share the same cycle. For example, as we model a 20 bytes per cycle running at 1 GHz communication fabric, a 62-byte message takes 4 cycles to transfer and the next message can only start transfer from the 5th cycle. For all the CPU and GPUs connecting to the fabric, we use a round robin arbitration. Each CPU and GPU are equipped with both a 4KB input buffer and a 4KB output buffer, so that the messages can wait in the buffer without blocking the fabric and the CPU or the GPUs.

VII. Results and Discussions

In this section, we begin by evaluating system performance and inter-GPU traffic for different compression algorithms. Next, we explore the proposed adaptive compression scheme while changing the λ value. We also evaluate energy savings resulting in a multi-GPU system. Finally, we consider the hardware overhead of our implementation.

A. System Performance and Traffic

1) Static Compression Algorithms: Data traffic between GPUs occurs not only for actual data shared across GPUs, but also for transferring metadata that is generated due to packetization and control flow management. Exploiting a compression algorithm can significantly reduce the amount of data transferred between GPUs, which can directly translate to cycle savings and improved system performance. We compare the resulting data traffic and execution times in Fig. 5. We normalize the traffic and execution times to the execution performance without no compression.

For BS we see a significant reduction in inter-GPU traffic and a reduction in the overall execution time. BS and KM launch a large number of kernels, where for each kernel launch the GPUs exchange some constant information, such as the kernel dimensions and input pointers, which tend to be highly compressible as compared to the more random input data. SC is another benchmark that offers a good compression ratio. This is mainly a result of zero padding in the image margins. The inter-GPU communication is a result of exchanging margin pixels, which are highly compressible, particularly with the BDI algorithm.

In general, the reductions in execution time of each benchmark track the reduction in inter-GPU traffic (i.e., compression ratio). This is an indication that the inter-GPU network

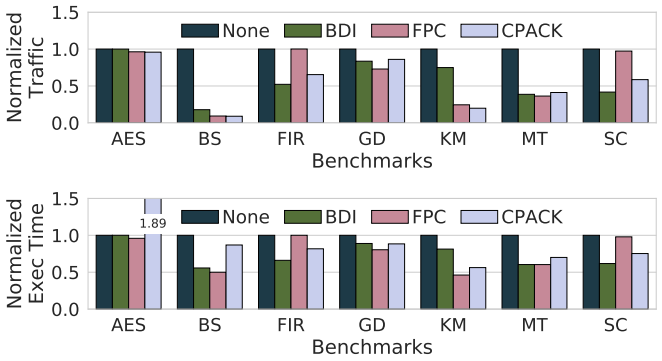

Fig. 5: Inter-GPU Traffic and Execution Time with Static Compression Algorithms.

contention is a major bottleneck in the system. In addition, we observe that C-PACK+Z sometimes adds extra execution time due to the long compression and decompression cycles. This issue is especially apparent in the AES benchmark. C-PACK+Z can only compress the data by a small fraction. Moreover, its latency cannot be hidden in our multi-GPU configuration.

2) Adaptive Compression Algorithm: Using an adaptive algorithm, we can dynamically adjust the compression algorithm used to match workload behavior. We configure the sampling phase to profile 7 requests, and configure the running phase to 300 requests. The selection criteria tries to minimize the penalty function presented in Eq. (1). We explore the impact of different λ values on the overall performance of multi-GPU system in Fig. 6.

When $\lambda = 0$, the smallest amount of traffic results. This suggests that our adaptive algorithm can help the system minimize inter-GPU communication. However, by ignoring the compression latency, we sometimes end up selecting the most expensive compression algorithm in terms of latency, even though the compression ratio is very low. Using this policy results in an unnecessary performance loss in the AES workload, and misses the opportunity to improve performance in the BS benchmark.

On the contrary, when λ is large ($= 32$), the system strongly prefers a low-latency compression algorithm (BDI in this case). Overall, this configuration improves overall performance versus when $\lambda = 0$. However, in some cases, using a value of 32 misses golden opportunities to compress the data. Selecting $\lambda = 6$ leads to a better balance between performance and bandwidth savings, and yields the best performance improvement (by 53%), and cuts down inter-GPU traffic by 62%.

B. Energy Consumption

The number of signal toggles of the bus links leads to power dissipation. Reducing traffic using a compression algorithm can significantly reduce the number of communicated packets, and can lead to significant power savings. Moreover, the performance improvements obtained by reducing the wait time for the requested memory blocks also results in energy savings.

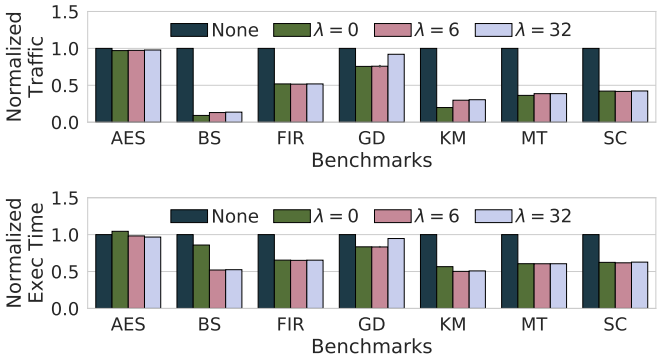

Fig. 6: Inter-GPU Traffic and Execution Time with Adaptive Algorithms.

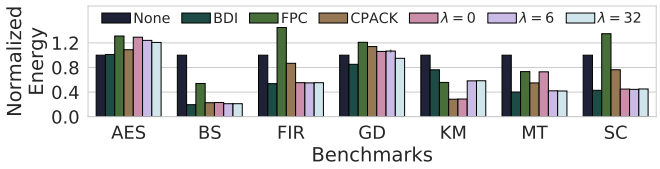

Fig. 7: Normalized Energy Consumption for the Compressors and Communication Fabric.

However, incorporating compression algorithms adds extra energy consumption. With respect to the selected compression algorithms, the energy consumption for compressing a data block (i.e., 512 bits) at the source and decompressing it at the destination varies between 1.3 to 40 pJ (Table III) This extra energy is negligible if the multi-GPU system architecture includes multiple nodes, or even multiple boards, where the energy cost for data transfer is roughly 250 pj/b [6] and 10 pj/b [7], [17], respectively. So the energy associated with a data transfer should track nicely with the amount of data traffic. Therefore, we report the energy consumption of an MCM architecture, where the on-package or inter-die communication energy is roughly 1-2 pJ/b.

The normalized energy consumption results due to the extra compression circuits and the data transfer on the communication fabric is reported in Fig. 7. Note that all the results are normalized to the case where no compression is used. The potential for energy savings depends heavily on the nature of the running application. For instance, if an application is not compressible (e.g., AES), exploiting compression adds energy overhead. FPC cannot compress the bandwidth in the SC and FIR benchmarks, which results in higher energy consumption across all cases. However, in KM, FPC shows better energy saving as compared to BDI, even though the energy overhead of using the FPC's hardware is significantly higher than that of the BDI (i.e., 1.3 pJ vs. 36.9 pJ).

The adaptive compression algorithm shows varying behavior, depending the selected λ value. $\lambda = 6$ achieves consistent energy savings for the studied benchmarks. It can save energy spent powering the communication fabric by 45%, as compared to the energy consumption without compression.

C. Hardware Area Overhead

We perform an offline calculation of the die-size requirements for the hardware-based compression and decompression algorithms. As the semiconductor industry is marching towards a 7nm technology, the R9 Nano GPU can reside in a die size of $37.25mm^2$ in 7nm technology. The size of the compressor/decompressor logic for BDI, C-Pack+Z, and FPC, when compared to the size of the GPU, is negligible, and only adds 4.35×10^{-4} %, 2.06×10^{-3} %, and 1.19×10^{-2} % area overhead, respectively.

VIII. Related Work

Hardware-based Memory Compression Techniques: Several hardware-based memory compression algorithms have

been proposed to compress caches or memory for CPU platforms [3]–[5], [12], [15], [31], [33]. The primary goals of these mechanism are to compact the data residing in memory and increase the effective memory capacity, rather than saving memory bandwidth. Our work re-purposes three important hardware-based memory compression techniques (i.e., BDI, FPC and C-Pack+Z) that were previously proposed for CPU caches in order to improve inter-GPU bandwidth and enhance total system performance. We selected the BDI, FPC and C-Pack+Z compression approaches, since they target different data patterns during compression, enabling us to tradeoff latency, power and complexity. These algorithms have hardware implementations, which allows us to perform realistic comparisons when integrated into a system.

Prior work has proposed to exploit hardware-based memory compression to save off-chip communication bandwidth between the processor and main memory. A differential compression algorithm, described by Benini et al. [8], compresses data while data is transferred from the cache to main memory. Thuresson et al. [42] propose to improve memory bandwidth by exploiting narrow values and value locality of data communicated over the link. Limited off-chip bandwidth is a very expensive resource on GPU platforms and can impact the performance of memory-bound applications dramatically [6], [36]. Therefore, Lal et al. [22] proposed an entropy-encoding based memory compression technique which employs Huffman encoding. To hide the latency of Huffman-based encoding, the approach adds more buffers, which increases the complexity and overhead.

Kim et al. proposed Bit-Plane Compression (BPC) to better manage the bandwidth of a GPU system. BPC transforms each memory block by first improving the inherent compressibility of the data. The transformed data shows higher inherent compressibility and better value locality. Then, a compression algorithm, such as frequent pattern compression, is used to compress the pre-coded data. Bit-plane transformations provide a general approach to pre-code the data and improve compressibility by reducing data entropy. This is orthogonal to our approach, and can be used to improve data compressibility by adding an extra layer before the compression algorithm.

Our work is targeted towards computation-based workloads for multi-GPU architectures, where it is challenging to customize a compression algorithm to match the custom requirements of each application.

Multi-GPU Architecture: Integrating multiple chips into a single module (i.e., MCM architecture) has been adopted in industry for several years. It promises and delivers cheaper single chip designs since it enables higher manufacturing yield [24] and better utilization of the area of a round wafer [19]. This technique has started to gain traction in industry, releasing MCM devices with multiple dies integrated on the same module, including CPUs [24], memories [19], and CPU+GPU [2].

MCM adoption is only possible with the advances in interconnection fabrics that have been designed to enable high bandwidth communication across multiple sockets or systems [17] [29], and with recent integration across multiple chips and dies [7]. However, after years of multi-chip research and development, only recently we observed efforts to explore the potential of implementing homogeneous MCM-GPU devices integrated in a single-socket [27] or single-module [6].

Milic et al. [27] investigates the impacts of non-uniform memory access (NUMA) designs on multi-socket GPUs, demonstrating that changes to the GPU's runtime, interconnect and cache architectures are necessary to achieve performance scalability. By comparing multi-socket GPUs and single-socket GPU scalability, this work provides evidence that NUMA-aware multi-socketed GPUs can be implemented to achieve better GPU performance scaling than multiple single-socket GPUs.

Arunkumar et al. investigate and evaluate the design of an MCM-GPU architecture [6]. This paper observes that several applications are scalable and can benefit from the availability of more Streaming Multiprocessors (SMs). This work suggests a few modifications to GPU architectures, such as adding a new cache level for remote data, and on-package integration of a distributed and batched co-operative thread array scheduler to leverage data locality in chips. The proposed architecture shows considerable improvement over multi-socketed GPUs.

IX. Conclusion

As transistor scaling become more challenging, new architectures are needed to scale GPU performance. Manufacturing a single, large, monolithic GPU die is not feasible. Moving to a multi-GPU system, and exploring system, board, package, and module design tradeoffs, can help us design future scalable systems. Performance scalability of traditional GPU workloads is pushing us away from a single GPU, and moving us toward multi-GPU designs.

In this work, we evaluate the potential of hardware-based memory compression to reduce inter-GPU traffic and improve system performance. We provide a comprehensive quantitative comparison of state-of-the-art hardware-based data compression algorithms in the context of a multi-GPU architecture. Our work shows that there is significant potential for compression, but that it is difficult to decide on which compression algorithm is the best. We proposed an adaptive and reconfigurable compression algorithm that reduces inter-GPU traffic significantly and improves overall performance. Our adaptive compression algorithms can reduce the traffic between GPUs by 62%, while improving system performance by 33%, on average. Moreover, it can save energy spent powering the communication fabric by 45%, on average.

References

[1] "Amd. amd app sdk 3.0 getting started," 2017. [Online]. Available: developer.amd.com/appsdk

[2] Advanced Micro Devices Inc., "AMD Opteron 6100 Series Platform Quick Reference Guide," Tech. Rep., 2010. [Online]. Available: https://www.amd.com/Documents/48101A_Opteron_6000_QRG_RD2.pdf

[3] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, vol. 1500, 2004.

[4] A. Arelakis, F. Dahlgren, and P. Stenstrom, "Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods," in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 38–49.

[5] A. Arelakis and P. Stenstrom, "Sc 2: A statistical compression cache scheme," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 145–156.

[6] A. Arunkumar, E. Bolotin, B. Cho, U. Milic, E. Ebrahimi, O. Villa, A. Jaleel, C.-J. Wu, and D. Nellans, "Mcm-gpu: Multi-chip-module gpus for continued performance scalability," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 320–332, 2017.

[7] N. Beck, S. White, M. Paraschou, and S. Naffziger, "'zeppelin': An soc for multichip architectures," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*. IEEE, 2018, pp. 40–42.

[8] 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 *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, vol. 4. IEEE, 2002, pp. IV–IV.

[9] M. Bohr, "A 30 year retrospective on dennard's mosfet scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007.

[10] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*. IEEE, 1999, pp. 13–22.

[11] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. W. Hwu, "Automatic parallelization of kernels in shared-memory multi-gpu nodes," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 3–13.

[12] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 8, pp. 1196–1208, 2010.

[13] X.-W. Chen and X. Lin, "Big data deep learning: challenges and perspectives," *IEEE access*, vol. 2, pp. 514–525, 2014.

[14] R. Das, A. K. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. S. Yousif, and C. R. Das, "Performance and power optimization through data compression in network-on-chip architectures," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 2008, pp. 215–225.

[15] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *ACM SIGARCH Computer Architecture News*, vol. 33, no. 2. IEEE Computer Society, 2005, pp. 74–85.

[16] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev, "Register packing: Exploiting narrow-width operands for reducing register file pressure," in *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. IEEE, 2004, pp. 304–315.

[17] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[18] D. Jacobsen, J. Thibault, and I. Senocak, "An mpi-cuda implementation for massively parallel incompressible flow computations on multi-gpu clusters," in *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*, 2010, p. 522.

[19] A. Kannan, N. E. Jerger, and G. H. Loh, "Enabling interposer-based disintegration of multi-core processors," in *Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on*. IEEE, 2015, pp. 546–558.

[20] G. Kim, M. Lee, J. Jeong, and J. Kim, "Multi-gpu system design with memory networks," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 484–495.

[21] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in opencl for multiple gpus," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 277–288, 2011.

[22] S. Lal, J. Lucas, and B. Juurlink, "Eˆ 2mc: Entropy encoding based memory compression for gpus," in *Parallel and Distributed Processing Symposium, 2017 IEEE International*. IEEE, 2017, pp. 1119–1128.

[23] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013, pp. 245–256.

[24] K. Lepak, G. TALBOT, S. WHITE, N. BECK, S. NAFFZIGER, S. FELLOW *et al.*, "The next generation amd enterprise server product architecture," *IEEE Hot Chips*, vol. 29, 2017.

[25] J. Macri, "Amd's next generation gpu and high bandwidth memory architecture: Fury," in *Hot Chips 27 Symposium (HCS), 2015 IEEE*. IEEE, 2015, pp. 1–26.

[26] K. Matsumura, M. Sato, T. Boku, A. Podobas, and S. Matsuoka, "Macc: An openacc transpiler for automatic multi-gpu use," in *Asian Conference on Supercomputing Frontiers*. Springer, 2018, pp. 109–127.

[27] U. Milic, O. Villa, E. Bolotin, A. Arunkumar, E. Ebrahimi, A. Jaleel, A. Ramirez, and D. Nellans, "Beyond the socket: Numa-aware gpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017, pp. 123–135.

[28] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 27–40.

[29] PCI-SIG, "PCI Express Base Specification 4.0," Santa Clara, CA, Tech. Rep., 2017. [Online]. Available: https://pcisig.com/specifications

[30] G. Pekhimenko, "Practical data compression for modern memory hierarchies," *arXiv preprint arXiv:1609.02067*, 2016.

[31] 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 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 377–388.

[32] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolić, *Digital Integrated Circuits, 2/e*. Prentice hall, 2003.

[33] S. Sardashti, A. Seznec, and D. A. Wood, "Yet another compressed cache: A low-cost yet effective compressed cache," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 3, p. 27, 2016.

[34] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2013, pp. 62–73.

[35] ——, "Could compression be of general use? Evaluating memory compression across domains," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 4, p. 44, 2017.

[36] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads," in *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*. IEEE, 2012, pp. 325–334.

[37] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1997, pp. 248–258.

[38] B. W. Smith and K. Suzuki, *Microlithography: science and technology*. CRC press, 2007, vol. 126.

[39] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, R. Ubal, X. Gong, S. Treadway, Y. Bao, V. Zhao, J. L. Abellán *et al.*, "Mgsim+ mgmark: A framework for multi-gpu system research," *arXiv preprint arXiv:1811.02884*, 2018.

[40] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization*. IEEE, 2016, pp. 1–10.

[41] M. K. Tavana, S. A. Khameneh, and M. Goudarzi, "Dynamically adaptive register file architecture for energy reduction in embedded processors," *Microprocessors and Microsystems*, vol. 39, no. 2, pp. 49–63, 2015.

[42] M. Thuresson, L. Spracklen, and P. Stenstrom, "Memory-link compression schemes: A value locality perspective," *IEEE Transactions on Computers*, no. 7, pp. 916–927, 2008.

[43] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW / SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," *MICRO 2018, 51th Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[44] A. K. Ziabari, Y. Sun, Y. Ma, D. Schaa, J. L. Abellán, R. Ubal, J. Kim, A. Joshi, and D. Kaeli, "Umh: A hardware-based unified memory hierarchy for systems with multiple discrete gpus," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 4, p. 35, 2016.