PACKET COMPRESSION IN GPU ARCHITECTURES

A Thesis

by

PRIYANK DEVPURA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2017

Major Subject: Computer Engineering

ABSTRACT

Graphical processing unit (GPU) can support multiple operations in parallel by executing it on multiple thread unit known as warp i.e. multiple threads running the same instruction. Each time miss happens at private cache of Streaming Multiprocessor (SM), the request is migrated over the network to shared L2 cache and then later down to Memory Controller (MC) for supplying memory block. The interconnect delay becomes a bottleneck due to a large number of requests from different SM and multiple replies from the MCs. The compression technique can be used to mitigate the performance bottleneck caused by a large volume of data. In this work, I apply various compression algorithms and propose a new compression scheme, Data Segment Matching (DSM). I apply approximation to the floating point elements to improve compressibility and develop a prediction model to identify number of approximation bits. I focus on compression techniques to resolve this bottleneck. The evaluations using a cycle accurate simulator show that this scheme improves Instructions per Cycle (IPC) by 12% on an average across various benchmarks with compressibility 50% in integer type benchmarks and 35% in floating-point type benchmarks when the proposed scheme is applied to packet compression in the interconnection network.

DEDICATION

To my family and friends

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. Eun Jung Kim, for supporting me throughout my research and providing valuable input. I would also like to thank Dr. Peng Li and Dr. Jiang Hu for serving on my committee and providing useful feedback on my work.

I would also like to express my gratitude towards Kyung Hoon for helping and guiding throughout the work. I would like to take this opportunity to acknowledge all the help I received from my friends and faculty during my time at Texas A&M University.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Eun Jung Kim of the Deparment of Computer Science and Engineering and Professor Peng Li and Professor Jiang Hu of the Department of Electrical and Computer Engineering.

All work conducted for the thesis was completed by the student, under the advisement of Professor Eun Jung Kim of the Department of Computer Science and Engineering.

Funding Sources

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

NOMENCLATURE

MC	Memory Controller
SM	Streaming Multiprocessor
IPC	Instructions per Cycle
CSN	Consecutive Same Nibble
DSM	Data Segment Matching
FP	Floating Point
NI	Network Interface
GPGPU	General-Purpose computing on Graphics Processing Unit
DRAM	Dynamic Random-Access Memory
CUDA	Compute Unified Device Architecture

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

1. INTRODUCTION

During the last decade, we have seen a significant number of applications such as image processing, data mining, data analytics, computer vision using graphics processing units (GPUs) due to their tremendous computing power [1][2][3][4]. GPU possesses a simpler control hardware. It has more hardware for computation and is potentially more power efficient. GPU is designed for throughput rather than latency. Its primary goal is to maximize the number of tasks completed per unit time. For instance, in computer graphics, we care more about pixels per second rather than the amount of time taken by a particular pixel. GPUs are also used in Bioinformatics and life sciences to improve the throughput of DNA sequencing alignment [5]. Overall, GPUs provide high computation power and are used for many data intensive applications that make researchers motivate to work in this domain.

GPU is composed of streaming multiprocessors (SMs) and on-chip memory controllers (MCs) that are connected to each other through a global crossbar interconnect network. The interconnection network plays an essential part in the GPUs and is critical to determine its performance. GPU applications involve large amount of data which results in many memory accesses. GPUs have thousands of ALU and employ a Single Instruction Multiple Thread (SIMT) architecture [6]. It can run hundreds of processes and has many threads running in parallel for executing the instruction with a combination of multiple threads known as warp. In the modern GPU architecture with a high degree of parallelism, a major challenge is how to efficiently feed data to thousands of threads running at the same time. On a Load instruction, SM requests the data from L1 cache. Every time, a miss happens at L1 cache, this miss results in a request for data needed to be served by L2 shared cache or MC directly connected to main memory. This results in a lot of read/write requests packet traffic from SM to MC and similarly Read/Write Reply packet traffic from MC to SM inside the network. GPU is designed to hide long memory access latency of stalled threads by interleaving execution of other threads on processing cores. Increasing parallelism would be able to hide more latency, but the limited resources on the cores constrain the maximum number of active threads. Eventually, this long memory access latency cannot be hidden and often becomes a major limiting factor to performance. As the memory latency cannot be hidden, this leads to overall system performance degradation. A considerable portion of memory access time comes from the MC bottlenecks where a large amount of reply data from MCs to SMs cannot be injected into the network due to restricted terminal bandwidth at the MC routers even when the data is ready to be sent [7]. Therefore, it is critical to explore solutions in the interconnect network to alleviate the MC bottlenecks.

1.1 Motivation

A lot of real world applications using GPGPU are data intensive, and there is a large volume of data going through the network. A miss at L1 cache would require the request to be sent over the network. Such requests are frequent, and the cores place heavy stress on a few MCs by sending a large number of requests in a bursty manner. The few MCs respond to the many cores by sending the memory reply data from L2 cache or DRAM. This burstiness and large volume of reply data from MCs create a severe bottleneck in the reply network. In the figure 1.1, we can see that on an average, MCs are being stalled during 39.06% out of the total execution time as reply network cannot accept reply packets. This MC bottleneck affects the overall execution time. It is critical to reduce the number of reply packets in the network. One intuitive way for lowering network traffic is to reduce the number of packets injected into this network. Data compression can be a viable solution that can effectively increase network bandwidth by reducing traffic data size.

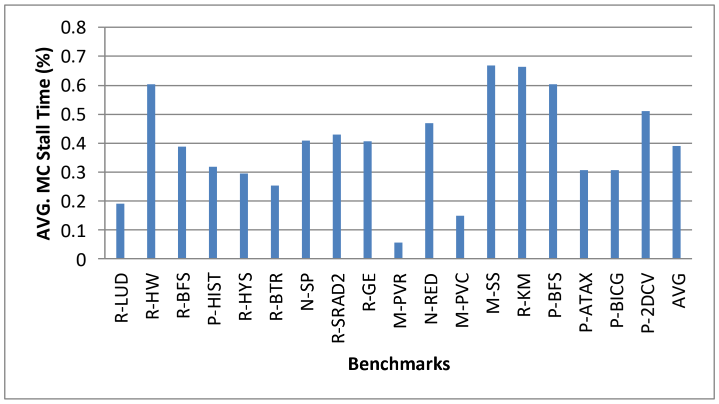

Figure 1.1: MC Stall percentage across various Benchmarks

We can apply compression on the data replied by the MC and send a compressed packet through the network. The ideal packet compression should be fast and simple regarding execution and hardware. Since in the case of packet compression, both compression and decompression lies on the critical, therefore, latency overhead should be minimal. The additional delays of adding this hardware should not offset the gains from packet compression. In addition to simpler hardware, the compression algorithm should also demonstrate a high amount of compressibility. Therefore, low latency and high compression ratio are critical design constraints for packet compression. However, existing compressors for lossless compression satisfy only one of the constraints. BDI [8] shows low compression latency with moderate compressibility. FPC [9] also shows moderate compressibility as it focusses on narrow value i.e. smaller values stored in large datatypes. SC^2 [10], FPH [11] and CPACK [12] achieve high compressibility by using a dictionary of frequent values, while BPC [13] uses a smart data transformation technique and applies compression with a combination of Run Length Encoding and FPC. However, these schemes have long latency overhead.

Due to limited compressibility in floating point (FP) data with lossless data compression algorithms, lossy data compression has been adapted data to further enhance compressibility at the cost of an application error [14]. The least significant bits of mantissa part of FP data is truncated for approximation. This idea of reducing FP precision is also widely studied in several areas [15][16][17][18][19]. Many applications in domains like computer vision and machine learning are tolerant to inaccuracies in computation, however, it is not clear to decide how many mantissa bits can be approximated to guarantee a user-defined error range for a given application. Truncating more number of mantissa bits can result in high error. For example, an application error in a common data mining application, Kmeans, is sensitive to cluster size and input data size. Even with the same number of approximation bits, an application with slightly different input makes a huge difference in the application error.

1.2 Our Approach

In this thesis, we design and apply packet compression across various compression algorithms and propose a new compression algorithm. We implement compressor and decompressor unit for compression algorithms and place it inside the GPU. A compressor compresses the data coming from L2 cache or main memory. The compressed output is stored in a compression buffer. Before injecting this into the network, we first need to convert into a network transfer unit called flit. Flits is then injected into the network. Similarly, when the flits are ejected out of the network, we first recombine the flits to form a compressed packet, it is then be decompressed using decompressor unit and send it to the SM. We can apply this approach to existing compression algorithms to evaluate their effectiveness for packet compression.

Apart from existing algorithms, we approach the compression in an another way. Initially, some of the data may not be compressible, but we can try to process the data to make it compressible. For instance, a 32B data may contain eight integer elements. These integer elements may have dynamic least significant bytes but identical higher bytes. We can preprocess the data in such a way that all the corresponding nibble positions of integers are brought next to each other which increases the chances of compression. We can use the processed data and try to map in the resolution of 8 bytes of identical nibbles by a single nibble. This way we can try to compress the packet which was earlier uncompressed. We apply lossy compression for FP data elements. We also develop a prediction model to estimate the approximation bits satisfying an error range considering application characteristics, input data and error evaluation metrics. We evaluate this compression technique with the existing compression algorithms.

The key criteria to determine the effectiveness of these compression algorithms will be compression ratio(CR), latency, Instructions per Cycle (IPC) improvement and area overhead. A packet is divided into multiple flits. CR is defined as the number of flits reduced divided by the number of original flits.

We determine the compressor latency which is number of cycles taken to compress the packet and decompressor latency which is number of cycles to decompress the packet. We calculate IPC improvement with respect to the baseline where no compression is applied. The cost of additional hardware is related to the area overhead of the compressor and decompressor.

We implement the compression and decompression algorithms inside GPGPU simulator. We test these algorithms across multiple benchmarks and run it in the simulator. The benchmarks can be integer type and floating type and support various real world applications such as page view rank, string matching, Linear decomposition. We also model the latency in our design to make it realistic. We simulate the compression algorithm across various benchmarks and calculate IPC improvement and compression ratio.

1.3 Organization

Chapter 2 describes the prior work in compression inside GPU and lossy and lossless compression algorithms. Chapter 3 describes the conventional GPU architecture and proposed design for compression in the GPU. Chapter 4 illustrates proposed lossless compression scheme. Chapter 5 discusses lossy compression in the proposed scheme and hardware design for the overall compression scheme. In Chapter 6, we evaluate various existing compression algorithms as well as proposed compression algorithm with key metrics: IPC, compression ratio, area, power and latency overhead. In the end, we present conclusion and future work in chapter 7.

2. PREVIOUS WORK

Several works have been proposed in the field of compression. There are generally 2 types of compression algorithms: lossy and lossless. In the lossy algorithm, the data used for compression and data received after decompression may or may not be same. To achieve higher compressibility, exactness needs to be sacrificed and lossy compression can be used in applications such as image and voice where loss of data does not affect the overall application quality. Whereas in the lossless algorithm, data recovered from decompression will be exactly same as the original data used for compression.

There are frequent redundancy patterns seen across multiple application data which can contribute to increasing compressibility. Firstly, there are a lot of zeros present in the application data. Zeros are used to initialize any data, to represent NULL values or false boolean values. Many compression schemes consider zeroes as a particular case or model their design around it to take benefit from the many zeros. Secondly, in the multimedia application, a large contiguous region of memory may contain a value repeated multiple times. Since these values are similar, there is a high of compression and representing them by few bits. For example in an image, with a large section of region having the same color will exhibit the similar pattern. Thirdly, a lot of time a smaller value is stored using a large data type. This is because the design is made generic and to account for the worst case and datatypes used in this case may not require data to consider all the bits. For example, one-byte value stored as a 4-byte integer. So, here the only byte has a significant value and rest bytes are zeros, and that makes compression quite practical. Compression technique can take advantage of these patterns and can reduce the size of the data. In this thesis, we will focus on hardware compression algorithm specifically employed in GPU and general lossy and lossless compression algorithms.

2.1 Data Compression in GPGPU

Several studies for data compression have been conducted in GPU architecture for energy savings and off-chip memory bandwidth reduction. To reduce the power consumption of register files, Lee et al. proposed a warp level register compression scheme. The register values of threads within the same warp exhibit similar values and sometimes there is a small variation between the successive thread registers. This paper employs BDI [8] compression scheme to reduce the data redundancy and therefore reducing the effective register width and power consumption. Pekhimenko et al. addressed increased dynamic energy caused by the frequent communication switching from 0 to 1 or vice versa, while transferring compressed data across on-chip and off-chip interconnect [20]. The transfer of compressed data in comparison with uncompressed data leads to higher number of bit toggles from 0 to 1 or 1 to 0 due to the increased per bit entropy as same amount of data is represented by smaller bits.

Sathish et al. applied both lossless and lossy compression. The paper uses CPack [12] for lossless compression for data transferred from GPU to off-chip memory to alleviate the memory bandwidth bottleneck. To further reduce this bottleneck and have high compressibility, lossy compression is proposed in which least significant bits of data are truncated [14].

BPC [13] focusses on compression between MC and off-chip to address the off-chip bandwidth bottleneck problem in GPU. It starts with a smart data transformation, Delta-BitPlane-Xor (DBX) to improve the compressibility of the data. DBX consists of three transformations. First, it subtracts the value from the neighboring values which help in creating small values. It then rotates the input values such that each of its output value includes one bit of every input symbol. Lastly, it takes xor of the neighboring values. It uses preprocessed data and applies the compression algorithm. It applies FPC and run-length encoding (RLE) by maintaining a pattern table. If the bit plane is zero, it uses RLE to encode with neighboring bit planes if there are any. For non-zero, it uses 4 patterns. First, it encodes a bit plane with all ones by a 5-bit code. Second, a bit plane with zero Delta Bit Plane and non-zero DBX is encoded with 5 bits. Third, two consecutive ones in the DBX then it is encoded with 10 bits. Another pattern is single one in which DBX has a single one and rest all zeros. which has patterns such as all ones, single one, consecutive ones, etc. One bit is used to denote the uncompressed data. Overall this compressor has a compression latency of 11 cycles.

The recent studies explored compression and applied in register files, on-chip to off-chip memory data transfer but packet compression has not been explored yet.

2.2 Lossless Compression Algorithms

The compression algorithms can be categorized as pattern-based and dictionary-based. The pattern-based approach compresses data with predefined static patterns into fewer bits whereas dictionary based maintains a global dictionary for frequent values.

FPC [9] compression algorithm is based on Significance Based Compression which compresses the cache lines on a word by word basis. Zeroes are frequent in the data and it encodes them into fewer bits. It also builds on the observation that some of the data patterns are frequent and can be compressed to a fewer number of bits. Many integers can be of small value and can be stored in 4, 8 or 16 bits but are usually stored in 4-byte or 8-byte data type depending on the architecture. Since the occurrence of this pattern is frequent, these can be stored in a compact form inside cache thereby increasing cache capacity. It applies compression and decompression on each cache line. It divides the cache line into smaller segments of 4-byte resolution. For example, for a 128B cache line, there will be 32 segments or words. It applies compression on each word if it matches a pattern. FPC maintains a fixed pattern table of 8 entries and has patterns such as all zeros, 4-bit sign extended, 1-byte sign extended, 2-byte sign extended, halfword padded with zero halfword, two halfwords each a byte sign extended and containing repeated bytes. If the word does not match, it is encoded as 111. Since there are eight entries, it uses 3-bit prefix to identify the type of pattern used for compression and appends it to the compressed word. For example, if the word is 0x00000004 then the compressed word size will be 7 bits. 3 bits will be used for encoding status and 4 bits for the non-zero nibble.

BDI [8] is a low latency compression algorithm which compresses data between on-chip caches and main memory. The last level cache stores the compressed data. It is based on the observation that there is regularity in the data which can be seen when memory is allocated to represent large data. Values in a cache block have a low dynamic range within their values that is the differences between the values stored in the cache line are small. BDI uses two base values to compress the data. One of the bases chosen is always zero which takes care of small integer values which are close to zero. Second base chosen is based on the first uncompressed value and typically pointers in a cache might have a small difference between other uncompressed values. BDI utilizes patterns such as all zero, repeated value and narrow value. It compresses the data with two bases: zero as inherent base value and first uncompressed value as second base. BDI divides the value in a cache line into words of 8-byte, 4-byte or 2-byte. It then tries to represent the value in a compact form by using a base value and representing other words with respect to that common base by taking the difference. BDI uses only two bases as multiple bases can improve compressibility but lead to an extra overhead due to storage of the bases. The compression logic of BDI consists of 8 compressor units. Six units for different base sizes of 8, 4 and 2 bytes and two units for zero and repeated value compression. The compressor takes in the cache line and tries to compress the data using all the compressor units in parallel. It selects compressed block whichever has higher compression and append the encoding bits corresponding to that base.

The dictionary-based approach maintains frequent redundant values in a dictionary table, and encode its appearance with a few bits. CPack [12] maintains a pattern encoding table which has six entries: all zeros (zzzz), 3 zero and one non-zero (zzzx), not matching with any patterns (xxxx), word matches in the dictionary (mmmm), 2 bytes matched and rest is unmatched (mmxx) and 3 bytes matched and 1 byte unmatched. It takes two words as input and compares each word with zzzz and zzzx pattern. If there is a match, it appends the encoding bits and unmatched part. Otherwise, it compares the word with the contents of the dictionary and tries to match with other patterns. It appends the encoding bits, unmatched bytes and index of the dictionary entry. This approach has a certain bottleneck when we are adopting this compression method for packet compression over the network, as each time we compress and send a packet over the network, we need to ensure that we have updated global dictionary before the same packet is decompressed. This implies that every time we compress any packet, we need to inject extra control message in the network for global dictionary update. This may negate the improvement in the IPC we got from packet compression if there are many dictionary updates.

SC^2 [10] uses Huffman-coding [21] which produces variable length encodings. It maintains a dictionary of top 1k entries word and constructs the encoding. It assigns more frequent word smaller length encoding whereas less frequent word gets more number of bits. FPH [11] is designed for compressing FP data. It divides floating point into four groups: sign, mantissa low, mantissa high and exponents. It maintains a value frequency table for exponent, mantissa low and mantissa high and builds a separate Huffman tree for each of these groups similar to SC^2. In this dictionary based approach, there is long compression and decompression latency overhead on achieving high compression due to variable length encoding.

2.3 Lossy Compression Algorithms

A few research done for lossy data compression can be categorized in three ways [22]. First, the compression exploits inherent impreciseness characteristics of storage devices. Many applications in domains like computer vision and machine learning are tolerant to inaccuracies in computation. Sampson et al. saves energy and gains higher write speed by relaxing strict data precision constraint on write operations in PCM [23] and improves lifetime by using failed cell blocks to store approximate data. Liu et al. propose a technique to reduce refresh power in DRAM memories. It uses regular refresh rate for a portion of memory containing critical data while employs low refresh rate for non-critical data in DRAM to achieve energy savings at the cost of increased data corruption [24]. Second, the compression exploits value similarity between existing data. The intuition behind this approach is easily understandable from the similarity between neighboring sets of pixels in an image. Miguel et al. deduplicate cache blocks with approximately similar values [25]. Miguel et al. design a new cache where the addresses of similar data blocks share the same cache location so that the data is reused for all accesses of associated addresses [26]. Third, data is approximated explicitly. Sathish et al. truncate a certain number of bits from the mantissa bits [14]. This work is similar to us in terms of the approximation target, but it does not address how to decide approximation precision to satisfy a user-defined error.

3. GPU ARCHITECTURE

In this section, we discuss the conventional GPU architecture. We also describe the proposed platform for packet compression inside the GPU.

3.1 Conventional GPU Architecture

In this section, we first discuss the basic architecture of GPU as shown in figure3.1. It consists of multiple cores known as Streaming Multiprocessors (SM) in Nvidia terminology and Computing Unit in AMD terminology. Each SM contains private L1 cache, texture cache, constant cache and shared memory. The SMs connected to memory partitions by the interconnection network. The other end of the network is directly connected to a L2 cache which is shared by all the cores and MCs. There are multiple MCs and each memory partition is controlled and managed by one MC. The memory controller requests and schedules the data from the memory partition. The CUDA compiler compiles the program and split into two sections: CPU and GPU. The parallel parts of the application are written in CUDA [27] and are offloaded to the GPU by launching the kernel. The CPU and GPU have separate memory and CPU copies the data into the GPU memory. Once the kernel is finished, GPU copies computed data back to GPU.

3.2 Proposed Architecture

We design data compression scheme to effectively remove network bottlenecks by reducing the amount of traffic in the network. Figure 3.2 illustrates the proposed platform. We apply data compression in a reply network from MC to SM. The L2 cache will contain the uncompressed data which it gets via MC and off-chip memory. When a global memory access request arrives in L2 cache, the data accessed from L2 cache or main memory is first compressed. Before this data is injected into the network interface, it needs to be

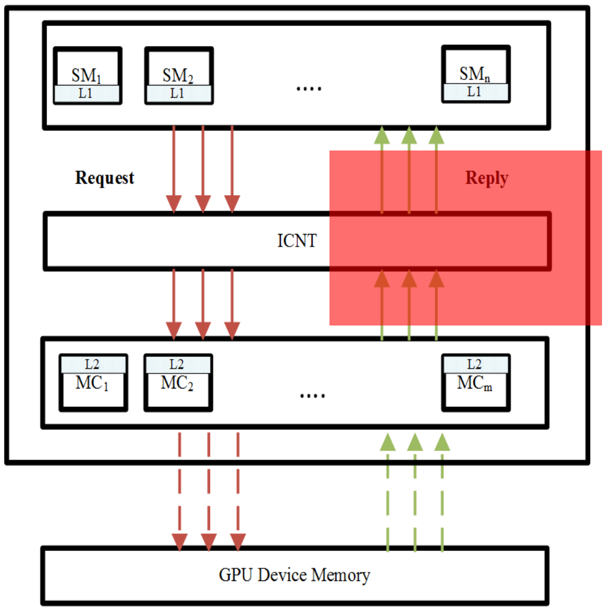

Figure 3.1: Conventional GPU Architecture

converted into network transfer unit called flits. The compressed data is delivered through the network interface. The data in a reply packet is decompressed right after the packet is depacketized in SM NI and thus L1D cache keeps uncompressed data only.

The floating point number show limited compressibility due to the significant variation in mantissa bits. We have enabled lossy compression that approximates the data at the cost of the application errors. The platform controls the application errors under user's acceptable error limit. To support lossy data compression with application output quality control, we propose a prediction model that is used to estimate the approximation precision satisfying user's acceptable application error. The approximation platform performs lossy compression based on the precision for approximable data, while it performs lossless compression for non-approximable data. The scope of the approximation is limited to FP

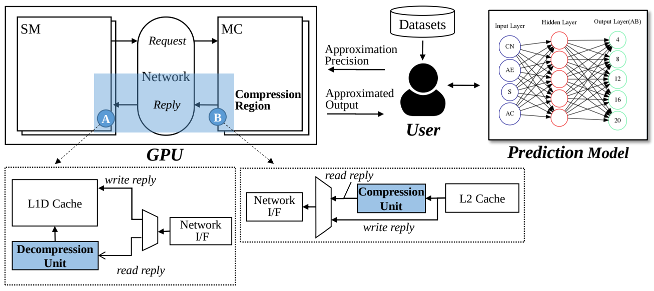

Figure 3.2: Overview of Platform with Lossless/Lossy Data Compression Capability

data. To utilize the proposed approximation platform, a user is assumed to run an application with a batch of datasets. A user runs an application with a subset of datasets on the approximation platform with different predefined approximation precisions and obtains application errors according to each precision to collect training data for the prediction model. The approximation precision can be defined as approximating 4, 8, 12, 16 and 20 mantissa bits. The application error depends on the approximation precision as well as characteristics of input data and application constraint. So, for the training data, metadata of each input dataset, application constraint and measured application error are used as attributes of the model. By providing the trained model with metadata of a new dataset and user-acceptable maximum application error, a user can obtain the approximation precision for the new dataset.

4. LOSSLESS DSM COMPRESSION ARCHITECTURE

In this chapter, we design a new lossless compression scheme, Data Segment Matching (DSM). We preprocess the uncompressed data to increase its chances of compressibility. The preprocessed data is used by the DSM compressor to compress it with low latency overhead.

4.1 Data Remapping

Before applying data compression, we preprocess an uncompressed cache block data into the compressible form. The Data preprocessing helps us in creating frequent Consecutive Same Nibble (CSN) pattern that are used by our proposing compressor as a compressible pattern. CSN pattern refers to nibbles with similar values consecutive to each other. For Example, let's consider a 4byte preprocessed data 0x11111111, where all the nibbles are one and it is a CSN pattern.

The data remapping function clusters the corresponding nibbles across data elements. A GPU often works with primitive data types, especially INT and FP. The numerically intensive computation from a host CPU is offloaded to a GPU. This results in storing of series of data elements at contiguous memory locations with the displacement corresponding to data type size. We observe that the corresponding nibbles often have the same value in the data elements with value redundancy. When the corresponding nibbles are closely rearranged, it is possible to create CSNs. For instance, in Figure 4.1, we take a 16-byte uncompressed data. Here, a nibble of 3 is present near MSB in all the four elements. These nibbles can be contiguously rearranged to form four consecutive nibbles. Thus, we can create value redundancy by applying data remapping. In this thesis, we explain the data remapping for 4B data elements since the most benchmarks rely on INT or single precision FP. However, this can be easily extended to other data types such as double precision

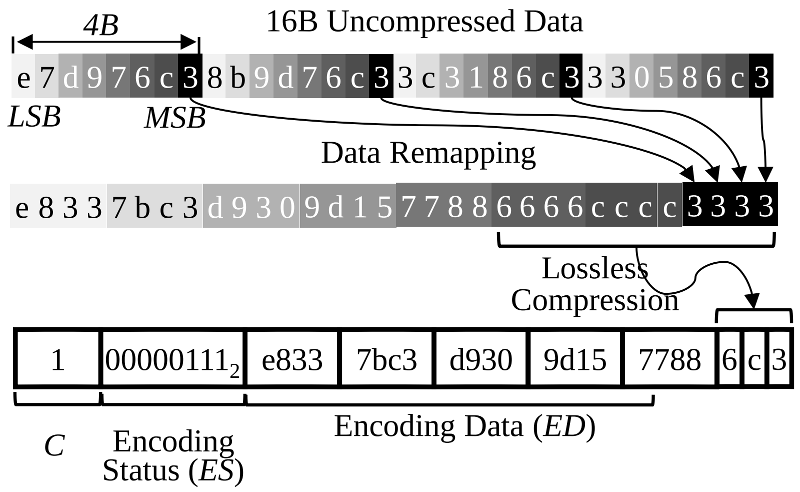

Figure 4.1: Example of Data Remapping and Lossless DSM Compression

FP (8B) and half precision FP (2B).

4.1.1 CSN Pattern Formation

Data remapping can help us in creating CSN for a narrow value and redundancy. However, we need to see if we can frequently form CSN with other redundancy patterns such as all zeros, repeated value and similar values that other compressors rely on. All zeros pattern is often found when program variables are initialized to zeros and since all the nibbles are same, it is itself a CSN pattern. Second, repeated values occur when variables are initialized as a value. For example, when the variables are set to a maximum value of INT type, all of them store 7fffffff. Thus, data remapping creates CSN patterns of 7 and f. Third, value similarity pattern is found in image files or video frames. A pixel is stored in 4B space where 3B RGB and 1B zeros for padding are placed. Due to the color closeness in adjacent pixels, the corresponding RGB data is similar. For same RGB values and padding zeros, data remapping clusters these values to form a CSN pattern. Data remapping effectively creates CSNs when data elements with the same type of data redundancy are contiguously stored, which is commonly seen in GPU.

4.2 Lossless DSM Compression and Decompression

DSM compression algorithm compresses a packet on a segment-by-segment basis by detecting segments of consecutive same nibble (CSN) patterns. We decompose an input data into a set of segments with the minimum compression resolution size (8B). If a segment matches the CSN pattern of a nibble value, then we encode it into the nibble value (4bit). Otherwise, the segment is encoded as its raw value. DSM produces the compressed data as an output if it is smaller than the uncompressed one; otherwise, its output is the uncompressed data. We set the compression flag (C) to 1 if the output is the compressed format. Each bit in *ES* indicates if the corresponding segment is compressed or not. The encoding data for all segments is stored at ED in order.

DSM decompression algorithm recovers a compressed segment as consecutive 16 nibbles of its 4bit code. An uncompressed segment is recovered with its encoding value. We examine each bit in ES to see whether a segment is compressed or not. To read the corresponding encoding data for each segment, we compute the offset from the starting address of ED. The bitwise offset of the ith segment is calculated by using the encoding status bits of the previous $i - 1$ segments, which is the sum of the number of compressed segments $\times$ 4 and the number of uncompressed segments $\times$ resolution bits.

Figure 4.1 shows an example of compressed data, assuming that the compression resolution is 2B to ease explanation. The data remapping creates 3 CSN-patterns of nibble value *6*, *c* and *3*. They are encoded as their corresponding nibbles, while others are encoded as its 2B raw value. The encoding status of 8 segments is encoded as 00000111_2 in the ES field.

5. LOSSY DSM COMPRESSION ARCHITECTURE

Due to limited compressibility in FP elements, we add another step of preprocessing in lossless DSM which is Data Approximation to enhance the compressibility in FP. In this chapter, we use a DSM(Data Segment Matching) and make two levels of data preprocessing: data remapping and data preprocessing as shown in figure 5.1. We apply data remapping and compression as described in the previous chapter. We discuss data approximation and overall hardware for the DSM compression scheme in this chapter.

5.1 Data Approximation

It is difficult to create redundancy for the FP elements just by applying remapping as most of the time these values are normalized and mantissa part is varying. We attempt to approximate floating point values to overcome a fundamental limit of lossless compressor's low compressibility. Data approximation replaces LSBs of FP element with zeros by the number of bits that a user specifies. The mantissa part is 23 bits on a single precision FP. Since the corresponding nibbles of data elements are already grouped by the data remapping, the approximation can be performed together for the data elements. Figure 5.2 shows an example where a user wants to approximate 8 LSBs. The first two groups of 4 nibbles which contain 8 LSBs of each data elements are set to zero. This results in consecutive nibbles of 0. Blindly, truncating the fixed number of LSBs can result in information loss. Therefore, we need to identify the ideal number of LSBs that can be truncated. However, to avoid the use of extra hardware, we rely on our compressor where the nibbles can be compressed with smaller bits.

In order to approximate FP data elements, we need to tackle two major issues: how to identify approximable data and how to decide the approximation bits (i.e. the number of approximable LSB bits in the mantissa part).

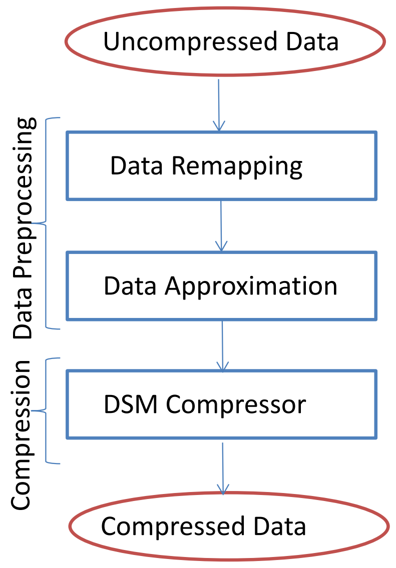

Figure 5.1: Compression Flow

We require the programmer to specify approximable memory space with non-zero approximation bits to identify approximable data. Before a host CPU launches a kernel, it often allocates the memory space in GPU through CUDA API (cudaMalloc) to keep three types of data: input data from a CPU, intermediate computing results and output data returning to the host. The first two types of data is a key target for approximation. If the memory space is used for storing approximable FP data, the programmer specifies the number of approximation bits as an extra function parameter. CPU completes the internal operation of CUDA API on the GPU by sending a set of commands to the GPU. Through this process, the user defined approximation bits is assumed to be sent. Based on the allocated memory space and the approximation bits delivered by cudaMalloc, GPU maintains an approximation memory map where the start and end address of the approximable region are associated with the approximation bits. The memory access within the approximable address range triggers data approximation function.

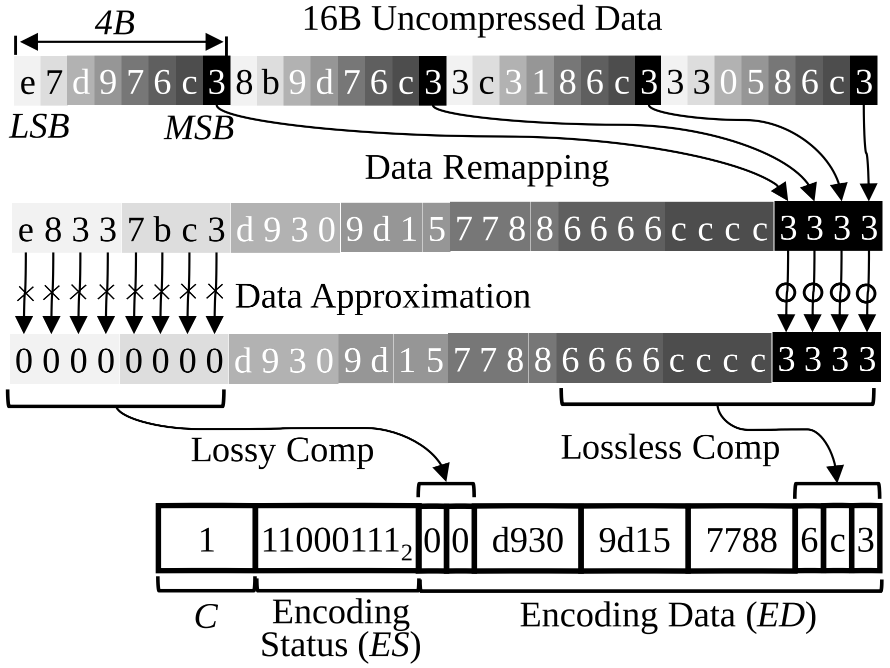
4B
16B Uncompressed Data
e 7 d 9 7 6 c 3 8 b 9 d 7 6 c 3 3 c 3 1 8 6 c 3 3 3 0 5 8 6 c 3
LSB
MSB
Data Remapping
e 8 3 3 7 b c 3 d 9 3 0 9 d 1 5 7 7 8 8 6 6 6 6 c c c c 3 3 3 3
Data Approximation
0 0 0 0 0 0 0 0 d 9 3 0 9 d 1 5 7 7 8 8 6 6 6 6 c c c c 3 3 3 3
Lossy Comp
Lossless Comp
1
11000111₂
0 0
d930
9d15
7788
6 c 3
C
Encoding Status (ES)
Encoding Data (ED)

Figure 5.2: Example of Data Preprocessing and DSM Compression

We implement a prediction model that satisfies a user defined application error to decide the approximation bits. It is believed that the impact of reducing FP precision on application errors is minor [14]. However, it is application-dependent. The error varies with not only the approximation bits, but there are also other three factors such as input data, application constraint and error metric. To obtain reliable approximation bits, a user trains an application-specific prediction model with prior knowledge of errors obtained from different configurations. To collect the training data, a user first chooses an application constraint and an error metric, and then execute an application with different approximation bits on different datasets. A user computes an error by comparing a precise output to a approximated output based on the chosen metric. The user collects information such as measured error, input metadata and application constraint as input attributes, and approximation bits as a target attribute. Through the trained model, a user later obtains the approximation bits for a new dataset by providing the model with input metadata, application constraint and target application error.

We implement the prediction model by employing a neural network with backpropagation created with an input layer of 4 nodes, a hidden layer and an output layer of 5 nodes. We define an input vector $X = \{S, CN, AC, AE\}$ where S is an input data size, CN a base-10 log of a condition number (a property of a matrix that assesses the accuracy of the solution to linear system) [28] of a matrix input, AC an application constraint and AE a measured application error. An output vector O =$\{4, 8, 12, 16, 20\}$, which is the approximation bits. The application constraint is a flexible configuration defined by an application. An example of application constraint is a cluster size in Kmeans application. We employ a mean square error as the stopping criteria of iteration and gradient descent for weight update and set a learning rate as 0.01.

5.2 DSM Compression and Decompression

The output of data approximation is used to apply compression. Figure 5.2 shows an example of compressed data, assuming that the compression resolution is 2B to ease explanation. As, previously discussed, the data remapping creates 3 CSN-patterns of nibble value *6*, *c* and *3* using data remapping. The data approximation creates 2 more CSN-patterns of *0*. This improves the overall compressibility. The encoding status of 8 segments is encoded as 11000111_2 in the ES field.

5.3 Hardware Design

In this section, we discuss the hardware design of Data Remapping Unit, Data Approximation Unit, DSM Compressor Unit and DSM Decompressor Unit.

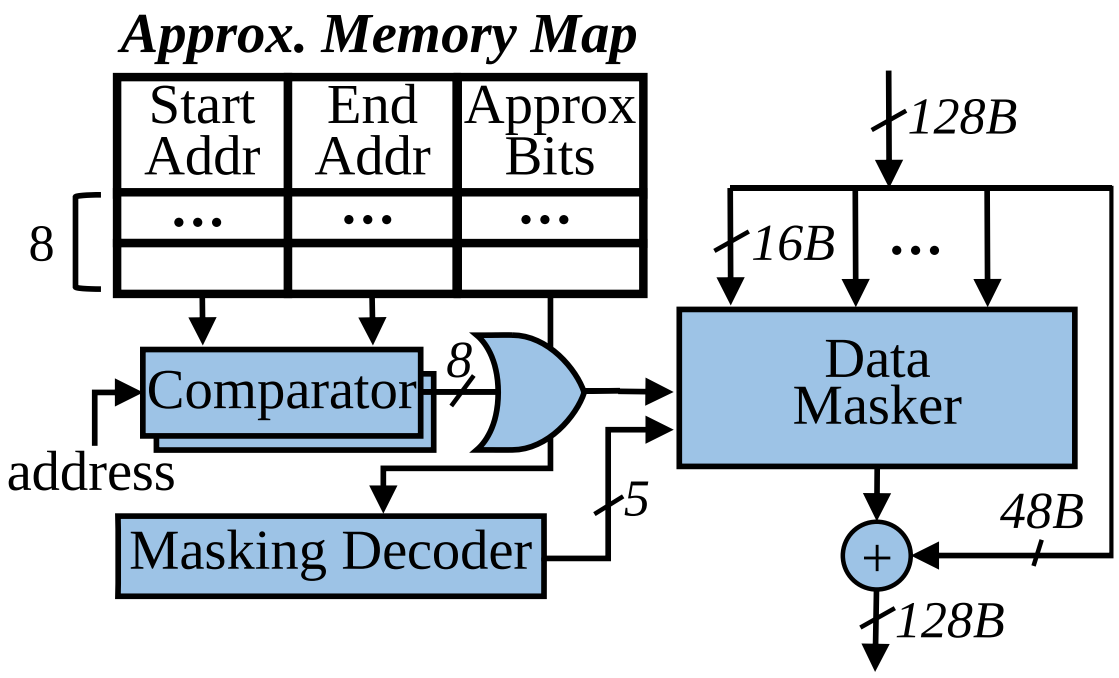

Figure 5.3: Hardware Design of Data Approximation Unit

5.3.1 Data Remapping Unit

Data Remapping Unit rearranges the data to improve compressibility. The signals of the input data is remapped to produce the output as illustrated in Figure 5.2. The remapped data is readily recovered to an original data through reverse remapping. The proposed remapping is obtained with negligible hardware overhead.

5.3.2 Data Approximation Unit

Figure 5.3 illustrates hardware design of data approximation unit. The design has a 8 approximation entry memory map which contains start address and end address for the approximation. There is also a 5-bit approximation number which tells us how many levels we have to approximate. With this information, we can get the block address from the

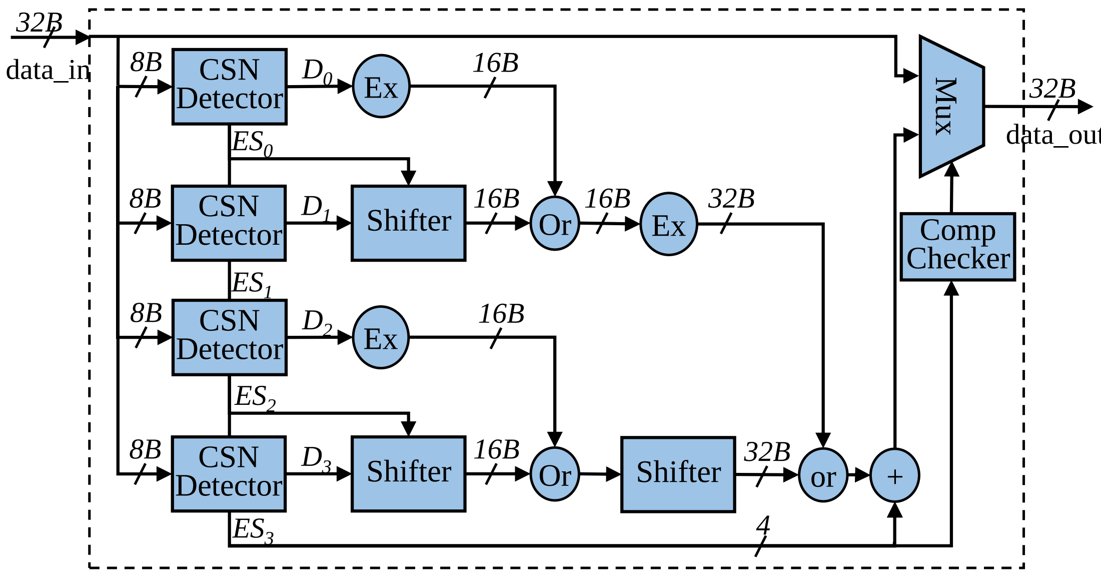

Figure 5.4: Hardware Design of DSM Compressor

cache block and compare if it lies in the approximable address ranges. If the address range is present between the start address and end address, then based on the approximation bits which are one corresponding contiguous block of data is set to zero. For a 128B cache block, let's say, the decoding bits 11100_2 generated for the approximation bit numbers 12 tells us that the first three 16B blocks are set to zeros.

5.3.3 DSM Compressor Unit

Figure 5.4 illustrates detailed hardware design of the compressor unit with 8B resolution. Given a 32B data, it produces a compressed output which contains compression flag, encoding status bits and encoded data in Figure 5.2. The hardware has various components such as CSN detector, variable length concatenator and compression checker. The input data is divided into 8B segments, each of which is fed to a corresponding CSN Detector. A CSN detector contains comparators that will compare if each nibble in an 8B is same or not. Figure **??** shows the design of a 2B CSN detector. Here, first comparator com-

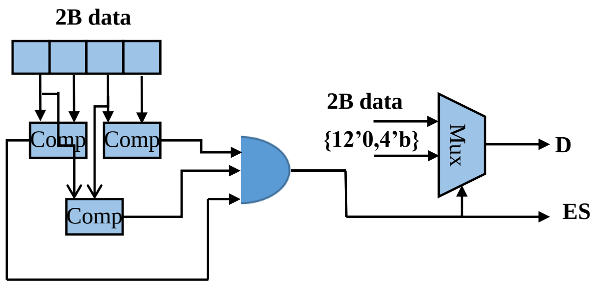

Figure 5.5: Hardware Design of 2B CSN Compressor

pares nibble 1 and nibble 2 and next one compares the other nibbles. The next comparator compares the nibble 1 and nibble 3 and then detect if all the nibbles are same by checking the comparator output. The output is produced based on ES bit. This can be easily extended to 8B CSN detector. CSN detectors work in parallel and examine if the segment is a CSN pattern. Then, each CSN Detector produces both the examination result as a 1bit encoding status bit and 8B segment value as an output. Combining all encoding status bits creates ES bitstream field. However, the encoding data of segments have variable length, 4bit or 8B whether or not the CSN pattern is found in a segment. Thus, we design a circuit for concatenating variable-length encoding data in order. As shown in Figure 5.4, the concatenation is performed for two consecutive segments and then the concatenated segments are again merged into a single encoding data. The compressor finally creates a compressed data by combining a compression flag bit set to 1, the encoding status bits and the encoding data. If no CSN pattern is found, all encoding status bits are set to ones, which is detected by Comp Checker. Then, DSM outputs an input data as it is. Otherwise, it outputs compressed data.

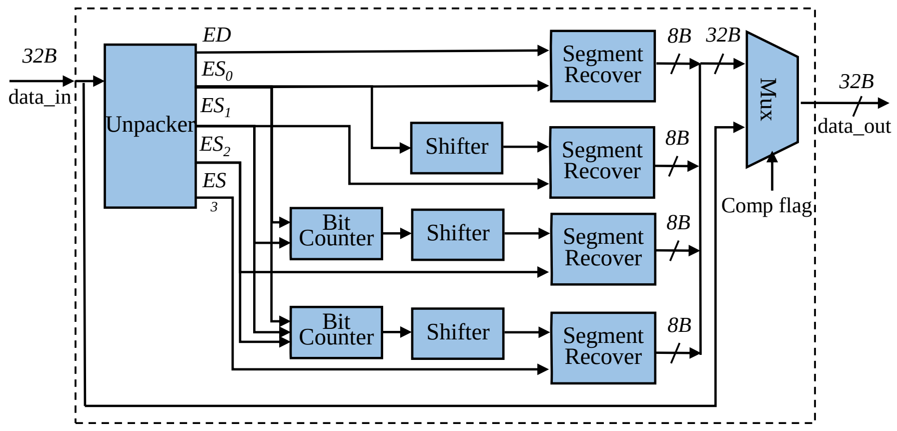

Figure 5.6: Hardware Design of DSM Decompressor

5.3.4 DSM Decompressor Unit

Figure 5.6 shows hardware design of the decompressor unit. The decompression unit has various components such as unpacker, bit counter, shifter and segment recover. The decompression starts by using unpacker which decodes the compressed packet into compression flag (C), encoding status bits (ES) and encoding data (ED) fields. When C is set to one, the decompression logic attempts to restore the encoding data to original data. Unlike an encoding status bit that is stored at the static position for a corresponding segment, the encoding data for each segment is stored at variable positions according to the compression status of previous segments. For the ith segment, $i + 1$ possible offsets from the beginning of ED exist. For instance, for the third segment in Figure 5.4, all, one, two or none of previous three segments can be compressed. To examine this, Bit Counter counts the number of ones in ES_k of previous segments ($k = 0..i - 1$). Shifter decides the start location according to the number of counts and relays an 8B value from the location to Segment Recover. If the encoding status bit associated with Segment Recover is set to one, Segment Recover takes 4bit from the relayed 8B value and produces 8B-CSN of the 4bit as a restored value. Otherwise, Segment Recover uses the relayed 8B value as an output. Decompressed data is created by combining all restored segments. If the C is set to zero, a final output will be same as input data.

6. EVALUATION

We evaluate existing compressors and DSM compressor across various benchmarks and assess the IPC improvement, compression ratio and hardware area overhead. We first explain how to integrate the DSM compressor to MC NI for packet compression. Then, we describe evaluation methodologies and discuss the evaluation results such as IPC improvement, compressibility, area, power and application error.

6.1 DSM Compressor Integration

We attempt to compress a read reply data returning from MC to SM before it is sent to network, while we send the write reply data without compression as shown in Figure 3.2. We integrate a compressor before a sender NI and a decompressor after a receiver NI. Thus, the compression lies on the critical path of packet transmission. To hide some of the compression latency, some compressors [29][30] have overlapped a flit traversal time and the compression operation. In the same principle, the decompression time can be hidden. However, we do not choose such design to compare all compressors with their inherent latency overhead. Decompression latency has a minor impact on throughput performance of GPU.

6.2 Methodology

We integrate compressors into a cycle-accurate GPU simulator, GPGPU-Sim 3.2.2 [31]. Table 6.1 shows the detailed system parameters to model our baseline Fermi GPU architecture [1]. There are 15 SM and 6 MC and different crossbars for request and reply network. We measure performance with instruction per cycle (IPC) and packet compression rate as the number of reduced flits divided by the number of flits for uncompressed data.

We evaluate DSM(+APPROX), BDI, FPC, CPACK and BPC. CPACK uses a dynamic

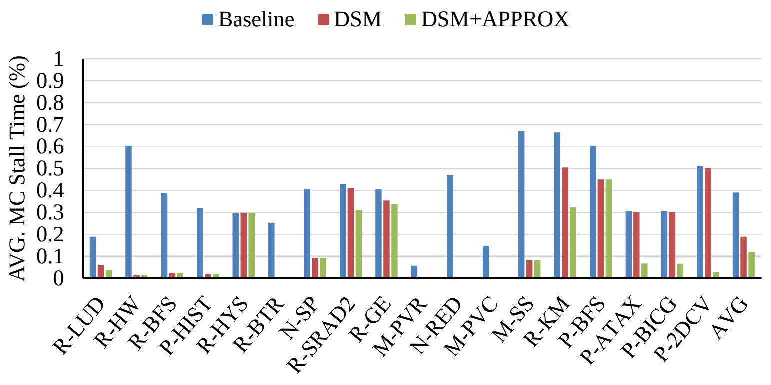

Figure 6.1: Analysis of Network Bottleneck in MC node

dictionary that is updated with the first occurrence of any words and uses the index of the dictionary for compression. To apply packet compression in CPACK, we need to maintain separate dictionaries near the MC side and SM side and these dictionaries should be synchronized to restore a compressed word correctly. Therefore, we implement extra packet communication protocol for synchronization. We have excluded other dictionary-based compressors, SC^2 and FPH from our evaluation which have the similar overhead of the extra communication for dictionary synchronization.

We choose a mix of INT type and FP type applications from multiple benchmark suites: Rodinia [32], Parboil [33], Mars [34], CUDA_SDK [35] and PolyBench [36] as shown in Table 6.2 and Table 6.3. Rodinia contains linear algebra applications which are composed of computation on a wide range of data with a good scope of redundancy. Another set of applications chosen is Parboil, which includes computational and throughput-oriented applications. These set of applications depicts the true essence of GPU architecture with high memory intensive operations. MARS includes set of applications like Page view rank, page view count and inverted index. These applications are associated with intensive data computations like web-based data whereas CUDA_SDK is a parallel computing

System Parameters	Details
Shader Core	15 Cores, 700Mhz
Memory Model	6 MCs, 924 Mhz
Warp Scheduler	Greedy-then-oldest(GTO)
L1 Inst. Cache	2KB (4 sets, 4-way LRU)
L1 Data. Cache	16KB (32 sets, 4-way LRU)
Shared Memory	48KB
L1 Texture Cache	12KB
L2 Cache	64KB (8-way LRU)
Min L2 hit latency	120 cycles
Min DRAM Latency	220 cycles
Interconnect	2 cross-bars for request/reply network

Table 6.1: System Configuration Parameters

benchmark. PolyBench suite contains benchmarks such as ATAX, BICG and 2DCONV which involves matrix multiplication and linear algebra. To obtain performance results, we run benchmarks until the end or till they reach 1 billion instructions in GPGPU-Sim. We also run a benchmark till the end through a functional simulator, CUDA-Sim [31] to obtain final output due to the longer simulation time of GPGPU-Sim. Data approximation function has been integrated to CUDA-Sim to get the approximated output.

To simulate approximation, we set the memory space keeping FP value as input data and intermediate data as approximable data regions. The number of approximable data regions for each benchmark is summarized in Table 6.3. The approximation bits is set to 4 for R-GE and R-LUD, and 12 for other FP-type benchmarks to conservatively assess increased compressibility by approximation. We compare the application output from the precise data to the output from the approximated data to evaluate an application error. We measure an error based on application-specific error metrics shown in Table 6.3. For N-SP, R-SRAD2 and R-HYS, we employ the Normalized Root Mean Square Error (NRMSE) as an error metric. For P-ATAX, P-BICG and P-2DCNV, we use the Mean Relative Error

INT Data Type	
Benchmark (Abbr.)	Dataset
Heartwall (R-HW)	50MB AVI File
BFS (R-BFS)	1M Nodes
BFS (P-BFS)	1M Nodes
Histo (P-HIST)	4MB Bins
B+tree (R-BTR)	1M Nodes
Page View Rank (M-PVR)	35MB Web Log
Page View Count (M-PVC)	43MB Web Log
Reduction (N-RED)	16M Integer Data Points
Similarity Score (M-SS)	1024×256 Data Points

Table 6.2: Summary of INT Benchmark and Dataset

FP Data Type		
Benchmark(Abbr.)	Dataset	Error Metric
Hybridsort (R-HYS)	500000 Data Points	NRMSE
LUD (R-LUD)	512×512 Matrix	MRE, NRMSE
ScalarProd (N-SP)	256 Vectors of 4096 FP Values	NRMSE
Srad2 (R-SRAD2)	2048×2048 FP Values	NRMSE
Gauss. Elimination (R-GE)	208×208 Matrix	$L\infty$-norm
Kmeans (R-KM)	20000 Instances of 34 Attributes	HM of Precision/Recall
ATAX (P-ATAX)	4096×4096 Matrix	Mean Relative Error
BICG (P-BICG)	4096×4096 Matrix	Mean Relative Error
2DCONV (P-2DCNV)	4096×4096 Matrix	Mean Relative Error

Table 6.3: Summary of FP Data Type Benchmark, Dataset and Error Metric

normalized between 0 and 1. For R-KM, we use a harmonic mean of precision and recall about the membership of input instances. For R-GE, we use a $L\infty$-norm.

6.3 IPC Performance Analysis

Figure 6.3 shows the normalized IPC when different compressors are used. To compare the effectiveness of lossless/lossy compression, we use two configurations for our algorithms. The first one, denoted as DSM is set to use DSM compressor only, and the

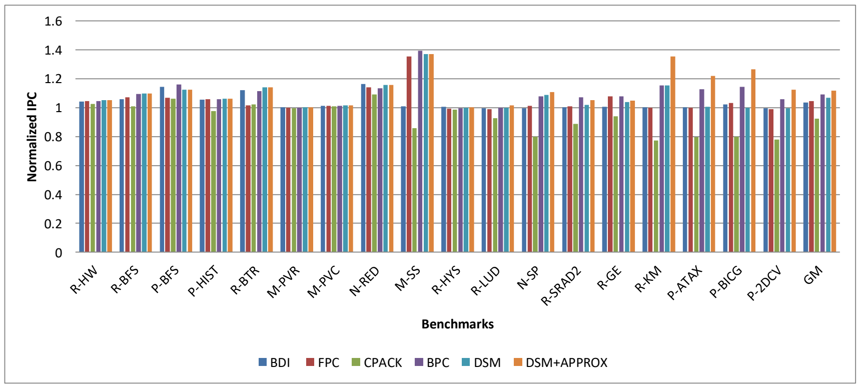

Figure 6.2: IPC Performance Comparison among Compression Algorithms

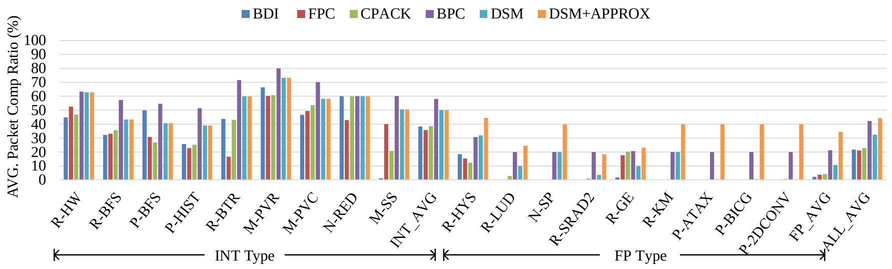

Figure 6.3: Compression Ratio Comparison among Compression Algorithms

second one, denoted as DSM+$APPROX$, adds approximation to DSM. We could make a conclusion that DSM+APPROX is more effective than other compressors in the packet compression, thereby obtaining higher IPC improvement.

DSM+APPROX obtains the IPC improvement, 12% on average, while BDI, FPC, CPACK and BPC does 3%, 4%, -14%, 8% improvements, respectively as shown in Figure 6.3. The performance benefit is correlated to the compression rate. Compared to BDI, FPC and CPACK, DSM+APPROX gains higher compressibility by around 22%. CPACK has slightly better compressibility than FPC and BDI, but it causes performance degradation due to the extra control packet that synchronizes a dictionary between a sender and a receiver node. The network bottleneck mitigated by compressed packets is shown as the reduced stall time in an MC NI in Figure 6.1. When DSM and DSM+APPROX are used, the stalled time is reduced by 20% and 27%, respectively compared to baseline.

6.4 Compression Performance Analysis

In this section, we analyze the compressibility of INT type and FP type in Figure 6.3. In INT type, DSM achieves higher compressibility than BDI, FPC and CPACK by around 11%. It is mainly because data remapping contributes to creating frequent 8B-CSN patterns. To understand this, we have analyzed the data redundancy patterns in INT type applications. The commonly known data redundancy patterns, all-zeros, narrow value, repeated value, and value similarity exist by 19.7%, 41.5%, 18.5% and 20% on average. The prevalence of these redundancy helps in creating frequent CSN-patterns. BPC shows higher compressibility than DSM by 8%. Unfortunately, the benefit of such higher compressibility is actually taken off by its long decompression latency, as we already discussed in Section 6.3.

In FP type, DSM+APPROX achieves 35%, while BDI, FPC, CPACK, BPC and DSM achieve 2%, 3.3%, 21% and 11.6%. As data approximation sets the partial mantissa bits of FP values, more zero CSN patterns are created, which enables higher compressibility. If DSM+APPROX aggressively approximates FP data by increasing the approximation bits to 20 bit, a compression rate can reach up to 80%. As we will discuss later, some benchmarks have near-zero errors with such high approximation bits. BPC and DSM still can achieve more than 10% compression rate by exploiting data redundancy in a sign bit and an exponent part of neighboring FP values. BPC has shown high compressibility in R-KM [13] because a default dataset provided by Rodinia benchmark suite [32] contains frequent zeros in FP type, while our dataset contains entirely non-zero FP values.

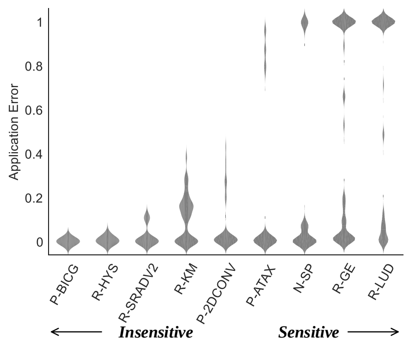

Figure 6.4: Error Distribution of Applications

6.5 Application Error Analysis

In this section, we analyze how data approximation affects application error across FP type benchmarks. The approximation bits is a primary factor affecting an application error. First, we characterize the sensitivity of benchmarks to the approximation by assessing an application error with different approximation bits. Among the approximation-sensitive applications, we analyze other factors affecting the application error. Last, we show the accuracy of the model that predicts the approximation bits satisfying a target application error.

Figure 6.4 shows violin plots for the distribution of measured errors. For each benchmark, 50 application errors are measured by varying the approximation bits from 4 bits to 20 bits for ten datasets. All benchmarks except R-GE and R-LUD achieve near-zero errors when small approximation bits (i.e. 4 or 8s) are used. In particular, three benchmarks, P-BICG, R-HYS and R-SRADV2 are insensitive to approximation, yielding 0%, 2% and 10% even with 20 approximation bits respectively. On the other hand, R-GE and R-LUD are very sensitive to the approximation and start to have near to one normalized errors in 50% of all measured cases. P-2DCONV, P-ATAX and N-SP have normalized errors from

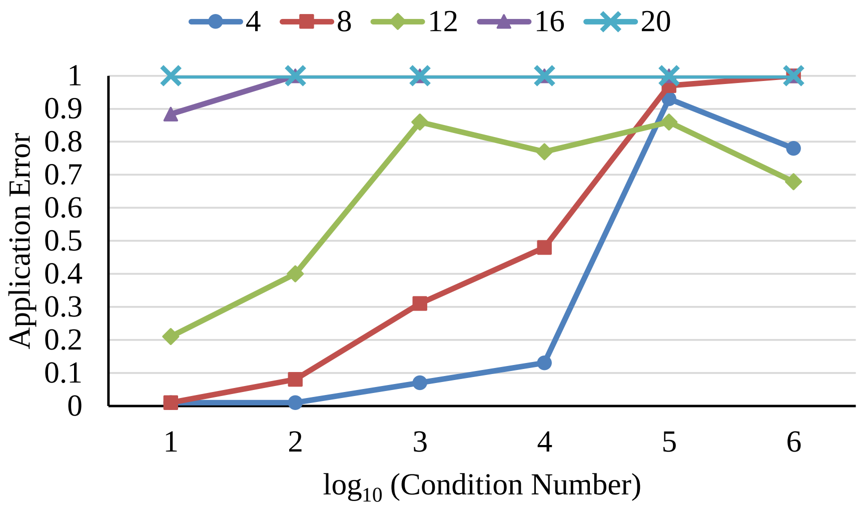

Figure 6.5: Condition Number

0.2 to 1 at high approximation bits (i.e 16 and 20 bits). Unlike other benchmarks, R-KM has application errors dispersed from 0 to 0.4.

Now we show that 4 representative factors affect the application errors through two benchmarks R-LUD and R-KM in detail. First, we observe that a condition number of an input matrix affects an application error. Figure 6.5 shows the application errors of R-GE when matrices have a log base 10 of their condition number varying from 1 to 6. For a fixed approximation bits, as the condition number increases, the error caused by approximation also increases. For instance, the approximation errors increase up to the log value 5 in 4 approximation bits, while it increases up to log value 3 in 8 approximation bits. The association of a condition number and accuracy of a solution in a linear system($Ax = b$) has been studied [28]. Coefficient errors of an ill-conditioned matrix that has high condition number can cause large error in a solution of the linear system, as we can see that 4 approximation bits make 93% application error for a matrix having 5 log base 10 of a condition number.

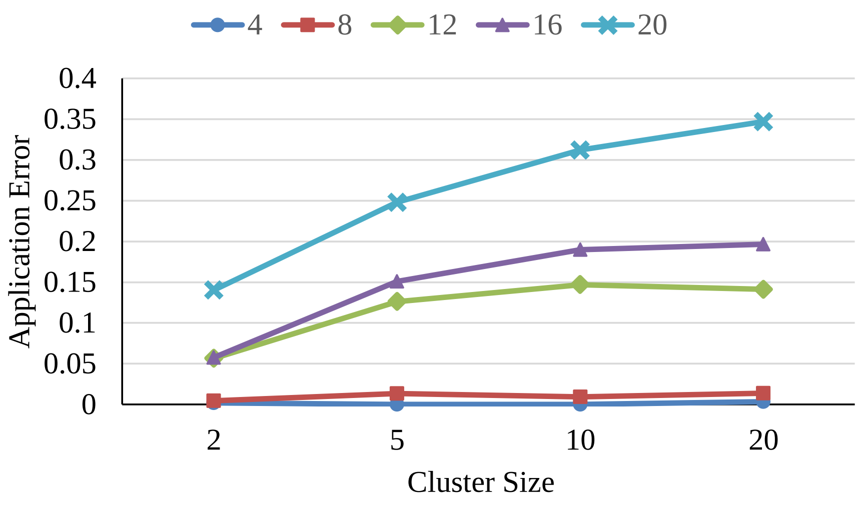

Figure 6.6: Cluster Size

Second, an application constraint, a cluster size in R-KM affects an application error. The measured application errors at 4 different cluster sizes, 2, 5, 10 and 20 are shown in Figure 6.5. When the approximation bits 4 and 8 are used, the cluster size does not have much effect on the error which constantly stays under 1.3%. However, as cluster size grows, application error also increases for approximation bits from 12 to 20. In other words, as the larger number of central coordinates are exposed to high errors, the chance of grouping input instances into different clusters increases. However, the mis-grouping is saturated for approximation bits 12 and 16 when cluster size is 20.

Third, input data size also affects application error in R-KM. Figure 6.5 shows application errors when four input data sizes 2500, 5000, 10000 and 20000 instances are used. We observe that application error is proportional to input data size except for the interval from 10000 to 20000. For instance, small approximation bits, 4 and 8 make a small error (less than 1.3%) in 2500 instances, while they do 5.5% and 14.3% application errors in 10000, respectively. This occurs because errors from the larger number of instances propagate to

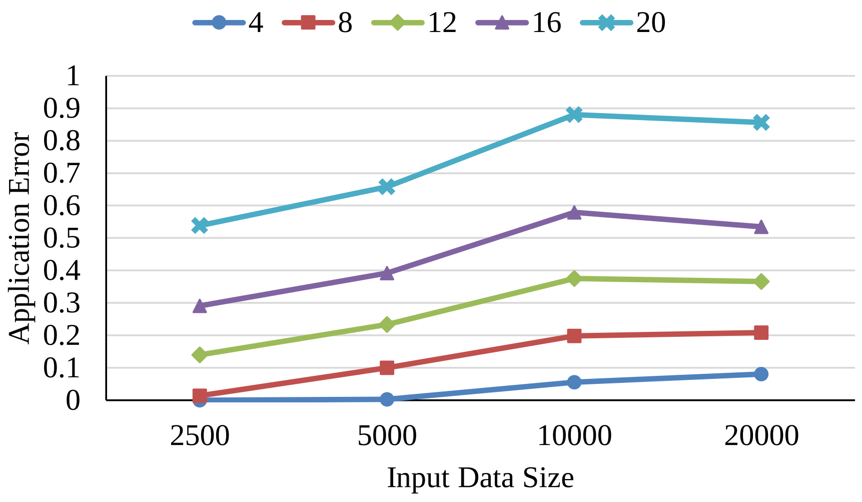

Figure 6.7: Input Data Size

the central coordinates of clusters. The similar relationship is also observed in R-LUD. Another possible characteristic of input data is the number of attributes, but our analysis (not plotted) with three attribute sizes, 17, 34 and 68 shows that the application error is independent of the attribute size.

Last, application error depends on error metrics. Figure 6.5 compares 50 application errors when we use Mean Relative Error and NRMSE for R-LUD. The values on y-axis associated with first 10 points on the x-axis indicate errors when 4 approximation bits are used for 10 different datasets, respectively. In the same manner, 10 errors for other approximation bits are plotted. The errors under NRMSE is overall measured with lower values compared to the Mean Relative Error. For instance, 100% error appears from 20 approximation bits with NRSE, while it appears from 8 approximation bits with Mean Relative Error.

With the above-detailed analysis, we design the approximation bits prediction model for each benchmark. We achieve 95% prediction accuracy as shown in Figure 6.9. R-

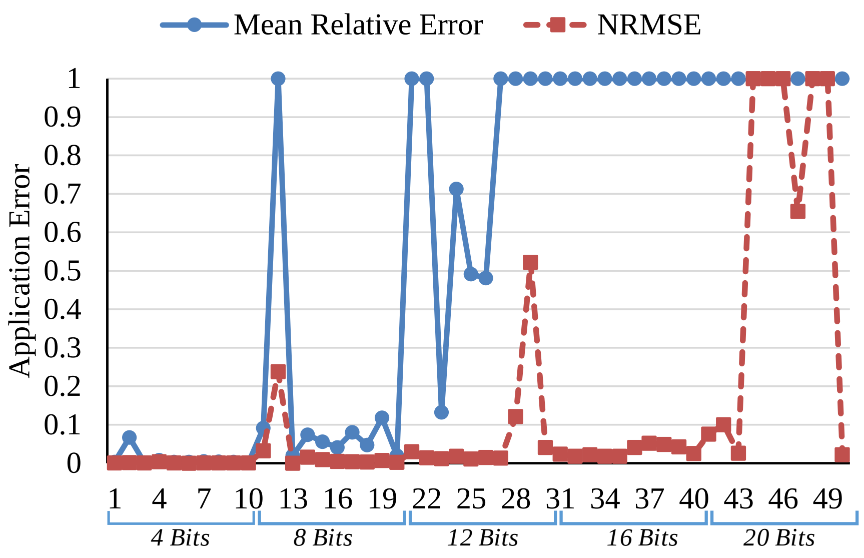

Figure 6.8: Error Metric

KM uses an input size, a measured error and a cluster size as attributes. R-GE uses the log base 10 of a condition number and a measured error. R-LUD uses a data size and a measured error. Other benchmarks use a measured error as an attribute. In the benchmarks using the single attribute, error ranges are clearly separated according to the approximation bits across different datasets. Thus, we achieve very high accuracy in those benchmarks, while we achieve an accuracy of 80%, 89% and 87% from R-KM, R-GE and R-LUD respectively.

6.6 Hardware Cost

We designed the hardware in Verilog and synthesized using Synopsys Design Vision [37] with 45nm open source standard cell library developed by Nandgate [38]. The proposed design is synthesized at 800Mhz frequency conservatively taking extra uncertainty to account for worst case delays seen during placement and routing. Table 6.4 categorizes the area, overall de/compression cycles and power at the maximum frequency.

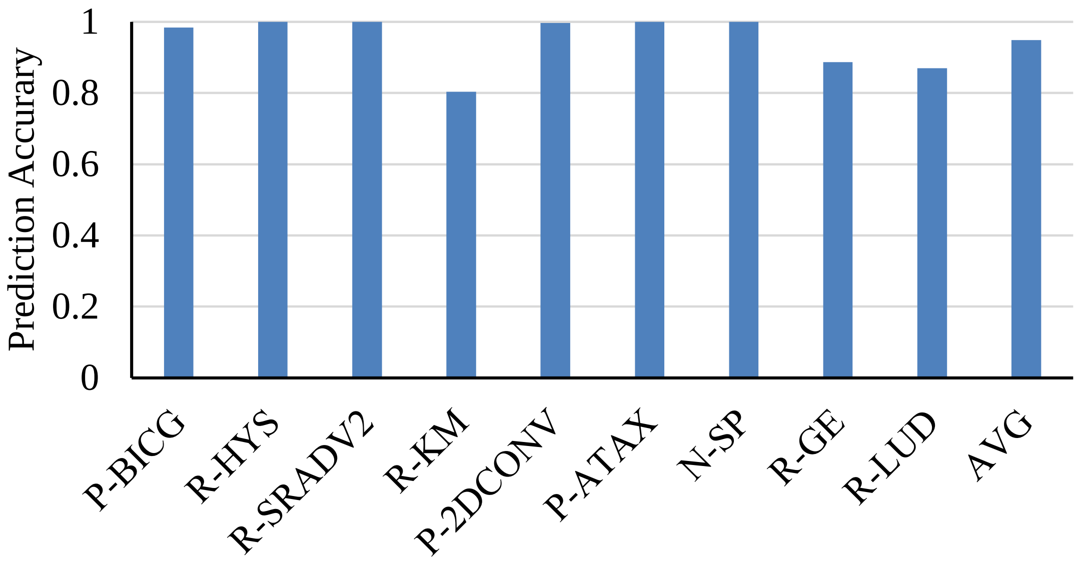

Figure 6.9: Accurary of Prediction Model

The evaluated compressors except BPC and DSM were designed for CMP that has 64B cache block. To support GPU with 128B cache block, we used their latency overhead reported in their literature, assuming that two de/compressors are integrated in parallel. FPC, C-PACK and BPC have 3, 16, and 11 cycles for compression and have 5, 9 and 11 cycles for decompression, which is high latency overhead as shown in table 6.5. Similarly, DSM is designed for 64B and two DSMs are integrated in parallel as others. DSM is simple, realistic and fully pipelined, and complete lossless compression and decompression in 2 and 1 cycles respectively, which is similar to the BDI. The approximation unit for DSM adds another 2 cycles which is used for higher compression in FP data. Since the decompression hardware has 1 cycle latency overhead, it can be effectively exploited in the latency-sensitive ones such as L1D cache and register file [39].

The overall area includes both combinational and sequential area. The overall area overhead for approximation unit, compressor and decompressor for 128B cache block is

0.0185 mm^2, 0.023 mm^2 and 0.0163 mm^2 which are roughly equivalent to 24K, 29K and 20K 2-input NAND gates, respectively. The breakdown of the area is 10% for the approximation unit,15% for the compressor, 12% for the decompressor and remaining 63% for the pipeline registers. The area overhead of DSM without approximation (29K) and DSM with approximation (53K) is smaller than BPC (68K) by 57% and 22%, respectively. DSM can be effectively used for energy savings with better power efficiency.

	Latency (cycles)	Comb. Area(mm^2)	Overall Area(mm^2)	Power(mW)
Approx. Unit	2	0.00535	0.0185	20.22
Compressor	2	0.009	0.023	20.9
Decompressor	1	0.00699	0.0163	12.76

Table 6.4: Synthesis Results of DSM De/Compressor

	BDI	FPC	CPack	BPC	DSM	DSM+APPROX
Comp. Latency	2	3	16	11	2	4
Decomp. Latency	1	5	9	11	1	1

Table 6.5: Latency Overhead

6.7 Power Comparison

In this section, we calculate dynamic power for various benchmarks across multiple compression algorithms. We use DSENT simulator [40] to calculate normalized dynamic power for the network as shown in figure 6.10. The network power across various benchmarks is normalized with respect to the baseline. The baseline power is taken as 1 in which no compression is applied. The packet compression results in the reduction of number of

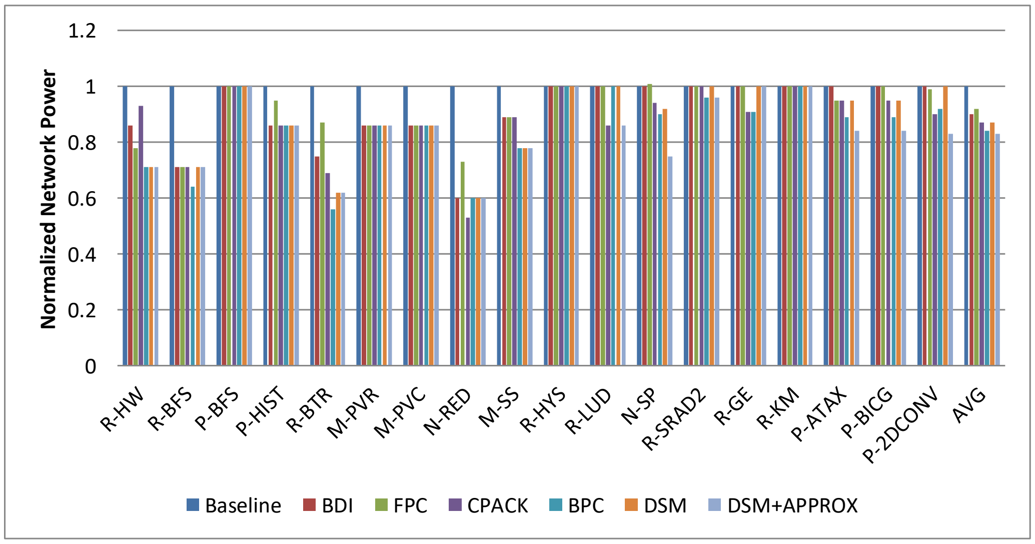

Figure 6.10: Network Power

flits injected into the network. The number of buffer reads and writes are reduced which contributes to reduction of power. The benchmarks which gave higher compression ratio resulted in higher reduction in network power. Overall, on an average DSM+APPROX shows 17% reduction in power whereas BDI, FPC, CPACK BPC and DSM shows 10%, 8%, 13%, 16% and 13% respectively. The gate count for BDI, FPC, CPACK, BPC, DSM and DSM+APPROX is approximately around 30k, 150k, 40k, 68k, 29k and 53k respectively. BDI, DSM and CPACK have lower area overhead and will result in lesser combinational power followed by DSM+APPROX, BPC and FPC. Since the area overhead of compressor in comparison with the GPU is not much, the additional hardware should not have much impact on overall power.

7. CONCLUSION AND FUTURE WORK

In this thesis, we present a new low latency compression scheme, DSM with two-level data preprocessing. Data remapping clusters nibbles at same positions from different data elements so as to create CSN patterns. Then, data approximation sets partial mantissa bits of FP data to zeros within the maximum error range, which is predicted by our neural network model to satisfy a user-defined application error constraint. Finally, DSM compressor compresses CSN patterns into small bits. We applied DSM compressor to packet compression. We achieved 12% IPC improvement with compressibility 50% and 35% in INT type and FP type benchmarks, respectively. The prediction model that estimates the approximation precision to satisfy user-defined error limit with 90% accuracy.

Due to low latency overhead of DSM compressor, we will apply this in caches to store compressed data. Motivated from our analysis of Kmeans application, we will extend our work for more machine-learning applications as future work. The approximation on FP data is indiscriminately considered as bringing minor error, but we identified that there is a class of applications sensitive to the FP approximation. Due to the limited machine-learning benchmarks, an important part of the future work will include developing more benchmarks using machine-learning algorithms.

REFERENCES

[1] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009.

[2] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," 2012.

[3] NVIDIA, "NVIDIA GeForce GTX 750 Ti," 2012.

[4] I. Advanced Micro Devices, "AMD Graphics Cores Next (GCN) Architecture," Jun 2012.

[5] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, "High-throughput sequence alignment using graphics processing units.," *BMC Bioinformatics*, vol. 8, 2007.

[6] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, Mar. 2008.

[7] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective on-chip networks for manycore accelerators," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*, pp. 421–432, 2010.

[8] 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*, PACT '12, (New York, NY, USA), pp. 377–388, ACM, 2012.

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

[10] A. Arelakis and P. Stenstrom, “Sc2: A statistical compression cache scheme,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA ’14, (Piscataway, NJ, USA), pp. 145–156, IEEE Press, 2014.

[11] 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*, MICRO-48, (New York, NY, USA), pp. 38–49, ACM, 2015.

[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, pp. 1196–1208, Aug 2010.

[13] J. Kim, M. Sullivan, E. Choukse, and M. Erez, “Bit-plane compression: Transforming data for better compression in many-core architectures,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 329–340, June 2016.

[14] 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 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT ’12, (New York, NY, USA), pp. 325–334, ACM, 2012.

[15] T. Y. Yeh, G. Reinman, S. J. Patel, and P. Faloutsos, “Fool me twice: Exploring and exploiting error tolerance in physics-based animation,” *ACM Trans. Graph.*, vol. 29, pp. 5:1–5:11, Dec. 2009.

[16] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, “Reducing power by optimizing the necessary precision/range of floating-point arithmetic,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 8, pp. 273–285, June 2000.

[17] C. Denis, "Numerical verification of large scale cfd simulations: One way to prepare the exascale challenge," in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on*, pp. 1255–1258, Aug 2014.

[18] Z. Chen, "Optimal real number codes for fault tolerant matrix operations," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–10, Nov 2009.

[19] S. Z. Gilani, N. S. Kim, and M. Schulte, "Energy-efficient pixel-arithmetic," *IEEE Transactions on Computers*, vol. 63, pp. 1882–1894, Aug 2014.

[20] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A case for toggle-aware compression for gpu systems," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 188–200, March 2016.

[21] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098–1101, September 1952.

[22] T. Moreau, J. San Miguel, M. Wyse, J. Bornholt, L. Ceze, N. Enright Jerger, and A. Sampson, "A taxonomy of approximate computing techniques," tech. rep., UW CSE Technical Report UW-CSE-2016-03-01, 2016.

[23] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, (New York, NY, USA), pp. 25–36, ACM, 2013.

[24] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," *SIGPLAN Not.*, vol. 46, pp. 213–224, Mar. 2011.

[25] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, "DoppelgÄnger: A cache for approximate computing," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 50–61, ACM, 2015.

[26] J. San Miguel, J. Albericio, N. Enright Jerger, and A. Jaleel, "The bunker cache for spatio-value approximation," in *In Proceedings of the International Symposium on Microarchitecture*, 2016.

[27] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2010. Version 3.2.

[28] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[29] R. Das, A. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. Yousif, and C. 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*, pp. 215–225, Feb 2008.

[30] Y. Jin, K. H. Yum, and E. J. Kim, "Adaptive data compression for high-performance low-power on-chip networks," in *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pp. 354–363, Nov 2008.

[31] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 163–174, April 2009.

[32] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of International Symposium on Workload Characterization (IISWC 2010)*, pp. 1–11, 2010.

[33] 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," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[34] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 608–620, April 2011.

[35] NVIDIA, "Cuda c/c++ sdk code samples," 2011. Version 3.2.

[36] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–10, May 2012.

[37] Synopsys, "Design compiler-rtl synthesis," 2010.

[38] Nangate, "45nm technology library," 2011. Version v2010 12.

[39] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient gpus through register compression," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 502–514, June 2015.

[40] C. Sun, C. O. Chen, G. Kurian, L. Wei, J. E. Miller, A. Agarwal, L. Peh, and V. Stojanovic, "DSENT - A tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling," in *2012 Sixth IEEE/ACM Interna-*

tional Symposium on Networks-on-Chip (NoCS), Copenhagen, Denmark, 9-11 May, 2012, pp. 201–210, 2012.