Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads

Vijay Sathish[†], Michael J. Schulte[‡], Nam Sung Kim[†]

[†]The University of Wisconsin-Madison, WI, U.S.A. [‡]Advanced Micro Devices, TX, U.S.A.

sathish@wisc.edu, michael.schulte@amd.com, nskim@engr.wisc.edu

Abstract

State-of-the-art graphic processing units (GPUs) provide very high memory bandwidth, but the performance of many general-purpose GPU (GPGPU) workloads is still bounded by memory bandwidth. Although compression techniques have been adopted by commercial GPUs, they are only used for compressing texture and color data, not data for GPGPU workloads. Furthermore, the microarchitectural details of GPU compression are proprietary and its performance benefits have not been previously published. In this paper, we first investigate required microarchitectural changes to support lossless compression techniques for data transferred between the GPU and its off-chip memory to provide higher effective bandwidth. Second, by exploiting some characteristics of floating-point numbers in many GPGPU workloads, we propose to apply lossless compression to floating-point numbers after truncating their least-significant bits (i.e., lossy compression). This can reduce the bandwidth usage even further with very little impact on overall computational accuracy. Finally, we demonstrate that a GPU with our lossless and lossy compression techniques can improve the performance of memory-bound GPGPU workloads by 26% and 41% on average.

Categories and Subject Descriptors

C.1.2 **[Multiple Data Stream Architectures (Multiprocessors)]**: Single-instruction-stream, multiple-data-stream processors (SIMD)

General Terms

Performance, Design

Keywords

Graphics processing units, lossless and lossy data compression.

1. Introduction

To reduce the performance impact of long latency off-chip memory accesses, GPUs rely on many threads in flight to hide memory latency. A GPU's hardware scheduler simply switches to a different thread group when the current group is waiting for data to return from its off-chip memory. Since these context switches are lightweight and involve practically no overhead for GPUs, the memory access latency can be effectively hidden for workloads with many threads. However, as the ratio of memory instructions to compute instructions increases, this technique becomes less effective. Workloads exhibiting such a behavior can be classified as memory-bound workloads.

Clearly, one way to improve the performance of these workloads is to improve the memory bandwidth by integrating more memory channels (i.e., memory controllers (MCs) and their memory I/O links (i.e., I/O circuits and pins, and interconnects)) and/or increasing their frequencies. The technical challenges, however, include limits in off-chip memory speeds and the number of I/O pins, neither of which scale well with technology scaling for a given chip package. Moreover, individual memory channels constitute a notable fraction of the power consumption in computing platforms (e.g., 25W for four DDR3 memory channels for a 4GB total capacity [1]), which can limit bandwidth and frequency increases under a given platform power constraint.

Considering such challenges, we can improve the memory bandwidth by utilizing the available memory channels more efficiently. As an effective alternative, in this paper, we propose a GPU architecture supporting lossless and lossy compression techniques for data transferred through the memory I/O links. Although compression techniques have been adopted by commercial GPUs [2], they are customized for compressing texture and color data. Furthermore, the microarchitectural details for GPU data compression are proprietary and its performance benefits have not been previously published.

The key contributions of this paper are as follows. First, we demonstrate the potential of a hardware-based lossless compression technique to improve the performance of GPUs by showing the compressibility of the data stored in GPU's off-chip memory (Section 2). Second, we propose necessary microarchitectural enhancement to support lossless and lossy compression for data transferred through the GPU memory I/O links (Section 3). The compression technique and implementation for GPUs are different than those for CPUs [3,4,5,6] because they exploit (i) a programming model and platform architecture unique to GPUs, which allows the compression to be transparent to existing workloads and compilation tools, and (ii) the impact of compression latency on performance is not as critical for GPUs as it is for CPUs. Furthermore, our lossy compression technique exploits the observation that reducing the precision of floating-point (FP) numbers incurs very little accuracy loss for many GPGPU workloads [7,8]; we apply lossless compression after truncating some least-significant bits (LSBs) of FP data to reduce the bandwidth usage even further while producing acceptable results for many GPGPU workloads in recognition, mining, synthesis (RMS) and physics simulation domains. Finally, we demonstrate the performance improvements of the GPU supporting lossless and lossy compression techniques, respectively (Section 4).

2. The Case for GPU Memory I/O Link Compression

GPUs typically provide much higher off-chip memory bandwidth and shorter latency than CPUs by using more memory channels and faster memory (e.g., GDDR); each channel is controlled by a MC. For example, 4-8 MCs are integrated in NVIDIA GPUs. However, many GPU workloads can still benefit from more memory channels (i.e., higher bandwidth). Figure 1 shows the performance speedup of 16 GPU workloads when the number of MCs (i.e., memory

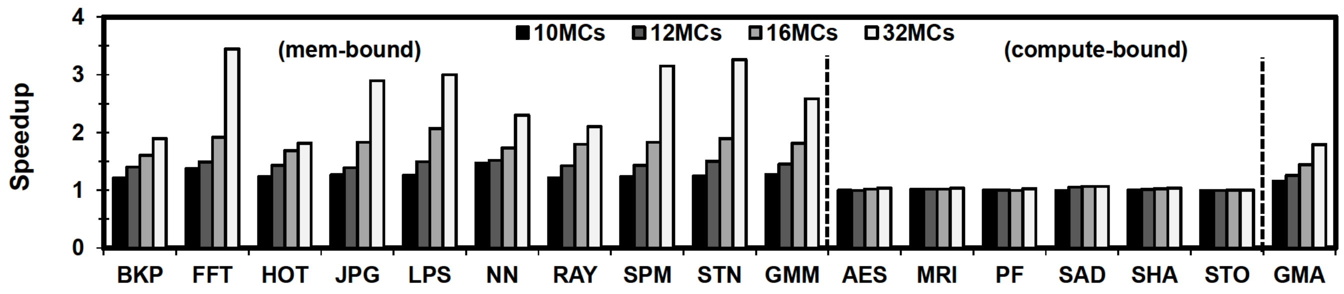


Figure 1: Speedup when varying the number of MCs, relative to a baseline GPU with 8 MCs.

channels) is increased from 8 to 10, 12, 16, and 32. While compute-bound workloads (i.e., AES, MRI, PF, SAD, SHA, and STO) show little or no performance improvement, memory-bound workloads (i.e., BKP, FFT, HOT, JPG, LPS, NN, RAY, SPM, and STN) significantly benefit from higher memory bandwidth provided by more meory channels; GMM and GMA denote the geometric mean speedup of memory-bound and all workloads, respectively. Doubling the number of memory channels increases the geometric mean of the performance of memory-bound workloads by 81%. See Section 4.1 for the benchmark descriptions, properties, and simulation methodology.

Figure 2 shows pseudo code for (a) an NVIDIA CUDA application, and (b) the corresponding data flow between a CPU (i.e., host) and a GPU (i.e., device). The CPU initializes the data for the GPU, stores them in the CPU DRAM (i.e., CPU main memory), and initiates a memory copy transaction by invoking the "CUDAMemCpy" function. Once the GPU acknowledges the transaction, the GPU's DMA controller begins to copy the data, which go through the GPU's on-chip interconnect and MCs to the GPU DRAM; see the dotted arrow lines in the "HostToDevice" direction.

After the DMA controller finishes the copy transaction, the CPU launches a GPU kernel and the GPU begins computations. During the kernel execution, streaming multiprocessors (SMs), each of which is comprised of 8-32 CUDA cores, read/write data from/to the GPU DRAM through the on-chip interconnect and MCs; see the solid arrow lines in the "MemRdWr" direction. After the GPU finishes the kernel execution, it invokes another "CUDAMemCpy," requesting the DMA controller to copy the computed results back to the CPU DRAM; see the dotted arrow lines in the "DeviceToHost" direction. During the memory copy transactions from the CPU DRAM to the GPU DRAM, the data can be compressed on the fly if the compressors, which are integrated with the GPU MCs, provide sufficiently high bandwidth.

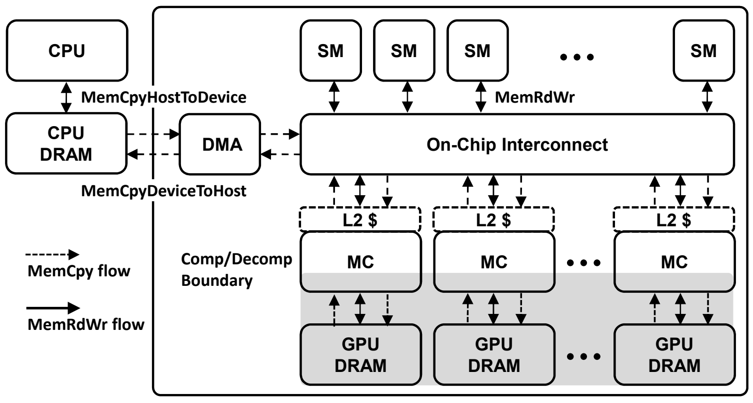
When an SM requests a memory read (i.e., a 128-byte block per request), an MC begins to fetch 16-byte compressed data chunks from the GPU DRAM in burst mode (i.e., 4 bytes per transfer). Since the number of transferred 16-byte chunks for the compressed 128-byte block is fewer than for the uncompressed block, the effective bandwidth and latency between the GPU and its off-chip memory are improved; the compressors and decompressors should support higher throughput than the memory channels. Then, the compressed block can be decompressed in the MC using the decompressors, and the decompressed 128-byte block is sent back to the SM; the gray-shaded region in Figure 2-(b) represents the region where the data resides in compressed form.

Figure 3 plots the average compression ratio of data copied from the CPU DRAM to the GPU DRAM; a single block is considered as a single independent unit for compression, and the size of a single block is denoted by block size. We applied a compression algorithm presented by Chen *et al.* [9] to every block transferred to the GPU DRAM. On average, the volume of the data can be compressed to approximately a half of the uncompressed size for 128-byte blocks. In turn, this can improve the effective (or perceived) bandwidth by a factor of two.

Finally, integrating the CPU and GPU on the same chip can enable them to share the same memory and may not require data transfers between them. However, high-end GPUs will still continue to be discrete devices (i.e., they are not likely to be integrated with CPUs in the same chip) because of power and thermal constraints.

```
//main routine that executes on the CPU
int main(void){
   .
   .
   // initializes data
   .
   .
   // initiates the data transfer to the GPU
   cudaMemcpy(..., MemcpyHostToDevice);

   // invokes the kernel
   foo <<<n_blocks, block_size>>> (...);

   // initiates the data transfer to the CPU
   cudaMemcpy(...,MemcpyDeviceToHost);
   .
   .
```

(a)



(b)

Figure 2: (a) Pseudo code using CUDA. (b) The data transfer flows between a host CPU and a GPU device; some GPUs have on-chip L2 caches.

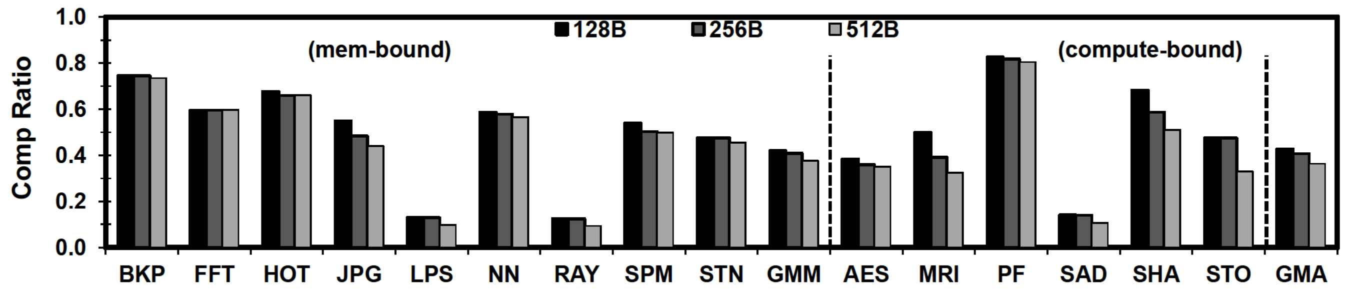


Figure 3: Average compression ratio, which is defined as the size of the compressed blocks over the size of the uncompressed blocks, when varying unit block size over which the compression algorithm is applied.

3. Microarchitectural Enhancement for Memory I/O Link Compression

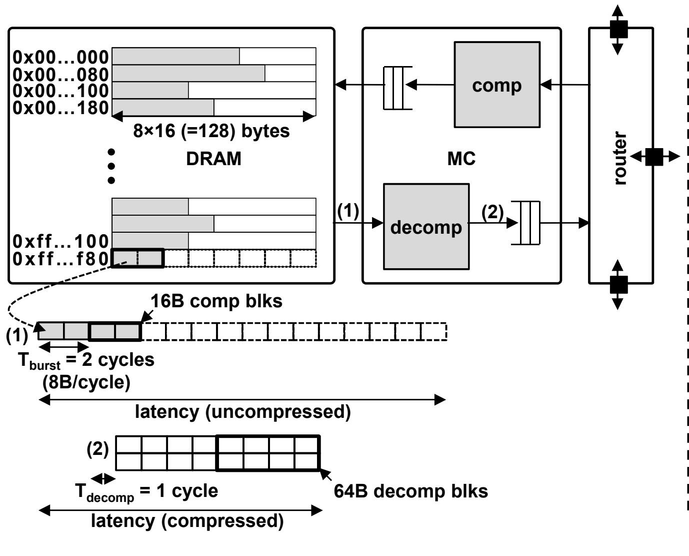
There are two important observations for supporting hardware-based memory I/O link compression to mitigate the memory bandwidth and latency issues for GPUs. First, the data values for some GPU workloads are likely to be highly correlated, resulting in high compression ratios as shown in Section 2. This can potentially improve the perceived bandwidth and latency. Second, threads running on an SM are executed as a group (i.e., warp) in a single instruction multiple thread (SIMT) fashion. Consequently, each warp, which is comprised of 32 threads, generates 128-byte memory accesses. Thus, GPUs allow a larger block size for compression than CPUs, potentially leading to a higher compression ratio.

3.1 General Requirements

In this paper, we make the compressed data residing in the GPU's DRAM transparent to the GPU's SMs by compressing and decompressing data with dedicated hardware in the GPU's MCs. Moreover, the host CPU is also completely isolated from the compressions and decompressions. Thus, this architectural change places no need for modifications to the rest of the system. The objective of the proposed compression technique for GPUs is not to increase the effective capacity of the GPU DRAM, but to improve the effective bandwidth and/or latency. Thus, our design still allocates 128 bytes per block in the GPU's DRAM, even though compressing a 128-byte block results in fewer than 128 bytes, as illustrated in Figure 4(a); the space saved due to compression is left unused to eliminate the complications often associated with space compaction and writing to compressed data.

3.2 Impact on Memory Bandwidth and Latency

In Figure 4(a), for the purpose of illustration, we assume that (i) GDDR3 has a burst length of four with a 4-byte bus width (16 bytes per burst) [10], and (ii) the throughput of our decompressor is 16 bytes per cycle with a 1-cycle latency [9]. Although the frequency of the decompressor can be 1.2GHz in a 65nm technology [9] and the frequency of GDDR3 and the MCs is 800MHz, we assume that the decompressor is operating at 800MHz. For example, when a 128-byte block is compressed to two 16-byte chunks (i.e., compression ratio = 0.25), a compressed 16-byte chunk from each burst read yields 64 uncompressed bytes on average. In this case, the bandwidth usage and latency are reduced to 1/4 and 2/3 of fetching a 128-byte block. While a compressed data block from a context is being decompressed, a memory access by another context can be interleaved to maximize the usage of the memory I/O links (and thus the bandwidth). Since these two data blocks are independent, they can also be decompressed in parallel with an additional decompressor.



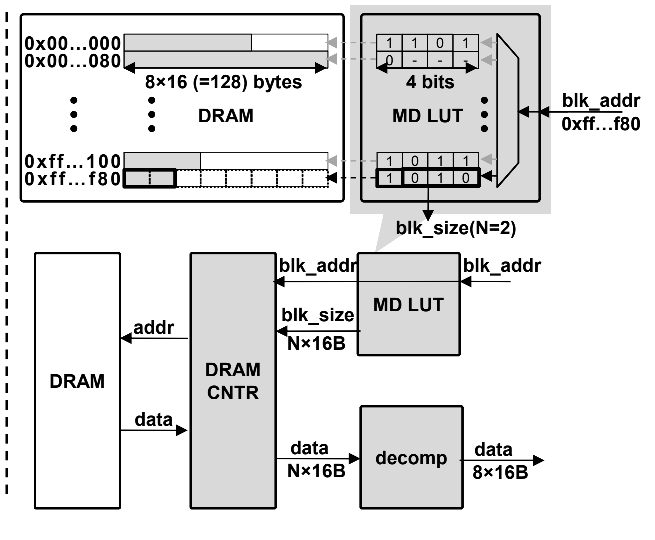


Figure 4: (a) Data flow for compression and the corresponding DRAM read and decompression transaction for 128 bytes compressed to two 16-byte blocks. In the figure, GDDR3 has a burst length of 4 with a 4-byte bus width (16 bytes per burst) [10]; and the throughput of our compressor and decompressor is 4 bytes/cycle and 16 bytes/cycle, respectively with a 1-cycle latency [9]. (b) Interaction between DRAM, DRAM controller, metadata LUT, and decompressor, where N is the number of compressed 16-byte chunks to fetch, instead of the full 8 16-byte chunks.

3.3 Metadata Cache

Need for compression metadata: There are two important reasons to maintain metadata information for every original 128-byte block stored in the GPU's DRAM. First, we need to store 1 bit per original 128-byte block to indicate whether or not the block is compressed; it is not always possible to compress every original 128-byte block to a block with fewer than 128 bytes. Second, we need to fetch only the compressed 16-byte chunks from the corresponding 128-byte block space. Although fetching the empty unused chunks in a full 128-byte block space is not harmful, it will nullify the bandwidth and latency benefit that we are seeking to maximize. Since the number of 16-byte compressed chunks per 128-byte block can vary from 1 to 8, we use three more bits per 128-byte block to indicate the number of 16-byte chunks in each block.

The need for metadata raises the question of how and where the metadata can be allocated, stored, and accessed. When the data blocks are being copied for the first time from the CPU's DRAM to the GPU's DRAM by the GPU's DMA controller, look-up-table (LUT) entries that contain metadata information for all the transferred 128-byte blocks, are populated on the fly during the compression process. The entries will be updated later if writes change the number of 16-byte compressed chunks. We also need to ensure that each MC only holds metadata information for the blocks belonging to that particular MC's address space; the consecutive LUT entries within a MC correspond to consecutive original 128-byte blocks belonging to the MC's address space.

Naïve implementation of metadata LUT: For the naïve implementation of the LUT, we assume the presence of a perfect cache to store all the metadata information. In Figure 4(b), for all read requests, the LUT returns the number of compressed 16-byte chunks (*N*) for a given block address (blk_addr). Then, the DRAM controller fetches *N* 16-byte chunks instead of 8 16-byte chunks. No matter what the value of *N* is, *N* 16-byte chunks are always decompressed into 8 16-byte chunks (i.e., the original 128-byte block). However, the GPU's DRAM can be up to 4GB for 32-bit addressing. Thus, we need 16MB (= 4GB / 128B × 4b) for LUT storage considering 4 bits of metadata information per original 128-byte block. Although the LUT size is just 0.4% of the total GPU memory space, it is very large for an on-chip memory, and thus not practical. On the other hand, if the LUT is located in the GPU's DRAM, the effective latency of memory accesses will increase considerably since each read or write requires two memory accesses, first for the metadata and then for the actual data.

Metadata cache: In this paper, we devise a microarchitectural technique to allocate the metadata information in the GPU DRAM, but cache the most-recently-used metadata information on the chip. This reduces the on-chip memory usage and the latency penalty of accessing metadata stored in the GPU's off-chip memory. Since we adopt a static one-to-one mapping of metadata information based on the data block's address, we reserve 16MB (this is sufficient for 32-bit addressing) of the GPU's DRAM from a known base address during the GPU initialization. When a particular data request address is provided to the MC, the metadata information corresponding to the block always maps to a location in the reserved 16MB space.

MSHR for metadata cache: Caching metadata utilizes a dedicated small two-way set-associative cache and a miss status handling register (MSHR) tables per MC. The metadata requests are placed by the MCs on behalf of the compressors or decompressors to the GPU's DRAM. Returning metadata responses from the GPU's DRAM are used to update the metadata caches and are not forwarded to any of the SMs. The hardware overhead of adding such a cache and a MSHR table per MC is negligible compared to storing all the metadata on chip; we provide a detailed analysis showing the cycle time, area and power consumption of a metadata cache and a MSHR table later in Section 4.1. Whenever a request for metadata misses, an MSHR entry, which is used to prevent duplication of metadata requests in flight, is allocated (if it has not already been allocated previously) and the request is sent to the GPU's DRAM.

Technically, the original 128-byte block request associated with the missed metadata must stall until the metadata returns and is parsed by the *request size modifier* that converts 8 16-byte accesses to *N* 16-byte access. This is to ascertain whether or not a block is compressed, and if so, how many 16-byte chunks need to be fetched from the GPU's DRAM. However, stalling memory requests for a metadata cache miss wastes precious memory bandwidth. Thus, the read request instead proceeds as a request for a full 128-byte block. This conservative sizing of data requests on a metadata cache miss ensures that memory bandwidth is never wasted due to stalling behind metadata requests. However, we note that the received 128-byte block has to stall at the decompressor until the metadata returns to verify whether or not the data block was compressed. Since metadata requests are always given priority over read/write requests, this stall period should be short. We also note that metadata responses returning from the GPU's DRAM require a separate virtual network to ensure that they never get blocked behind stalled data responses that are waiting for their corresponding metadata responses, as this would lead to a deadlock scenario.

3.4 Lossy Compression Technique

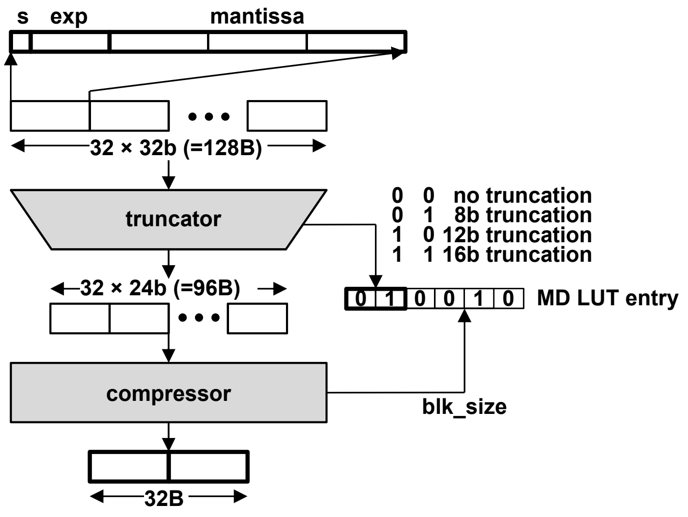
The impact of reducing the precision of FP data on computing accuracy has been widely studied for media, RMS, and physics simulation domain workloads [7,8]; truncating some LSBs of FP data before computing with them results in very little loss in overall accuracy while reducing a considerable amount of chip area and/or power consumption of FP units (FPUs). Furthermore, GPUs already support two different FP computation modes: IEEE compliant and non-compliant FP computations where the latter is much faster than the former at the cost of a small accuracy loss [11]. Since the LSBs in the data transferred from the GPU's off-chip memory are going to be truncated before computations in such a computing method, we observe that the GPU does not need to bring in the LSBs of the FP data. We hypothesize that this can also reduce the memory bandwidth usage significantly with a negligible loss in accuracy.

The main challenge, however, is that many GPGPU workloads use a mix of both integer (INT) and FP data and truncating INT data, unlike FP data, may have detrimental effects on accuracy (and sometimes may lead to erroneous execution states). Therefore, we propose to apply lossy compression only to FP data. Considering the CUDA programming style, which often transfers a set of data with the same data type from/to the GPU's off-chip memory using the CUDAMemCpy function as shown in Figure 5(a), we can selectively apply lossy compression only to FP data. For a programmer to specify the number of LSBs to be truncated for a set of FP data, we modify the *CUDAMemCpy* function interface; in Figure 5(a) we truncate the 8 LSBs of array "bd" copied to the GPU off-chip memory.

Figure 5(b) illustrates the hardware support for lossy compression. For example, if we truncate 8 out of 24 mantissa bits in every 32-bit single precision FP datum in a 128B block, effectively we shrink it to 96B, reducing the bandwidth usage by 25%; two extra bits in each MD LUT entry record the number of truncated LSBs, which is needed for decompression later. In addition to the truncation, we apply the lossless compression technique to the remaining 96B. This results in a 32B compressed block in this example. Note

```
int   *a = new int[N*N];
float *b = new float[N*N];

for(int i = 0; i < M*M; ++i)
    a[i] = i; b[i] = 1.0f;

int *ad; float   *bd;
const int intsize = N*N*sizeof(int);
const int fpsize = N*N*sizeof(float);

cudaMalloc((void**)&ad, intsize);
cudaMalloc((void**)&bd, fpsize);

//existing cudaMemcpy code
//cudaMemcpy(ad, a, intsize, HostToDevice);
//cudaMemcpy(bd, b,  fpsize, HostToDevice);

//new cudamMemcpy to support lossy compression
cudaMemcpy(ad, a, intsize, 0, HostToDevice);
cudaMemcpy(bd, b,  fpsize, 8, HostToDevice);
```

(a)



(b)

Figure 5: (a) cudaMemcpy example code and (b) hardware support for lossy compression.

that some applications require greater accuracy and use double-precision FP arithmetic. In this case, the proposed lossy compression can be an optional feature that programmers can turn on (e.g., to accelerate initial solution searches for large problem spaces).

3.5 MC Microarchitecture

Figure 6 illustrates the microarchitectural changes required for each MC to support a hardware-based memory-link compression technique. The GPU's DRAM (using NVIDIA's architectural convention) is partitioned into four different address spaces depending on the properties and caching patterns of the underlying data. The *global address space* stores the data arrays used for computation and can be both read from and written into. The read-only *texture address space* stores graphics textures and read only data structures. The read-only *constant address space* stores constants required during computation while a *local address space* is used to manage register spills (register data sent back from the SMs to the GPU's DRAM). Our proposed architecture only supports compression of the global and texture spaces, which represent the bulk of the memory traffic. Therefore, only requests prefixed with a "*g*" (global or texture space) in Figure 6 need to pass through the compressor and decompressor.

The requests prefixed "*l*" represent traffic to local and constant address spaces and safely bypass the compression scheme. The abbreviations "*rd*," "*wr*," "*req*," and "*resp*" refer to read, write, request, and response. For example, "*g_wr*" refers to a global write initiated by an SM. If a write is to a full 128-byte block, it passes through the compressor before getting written to the GPU's

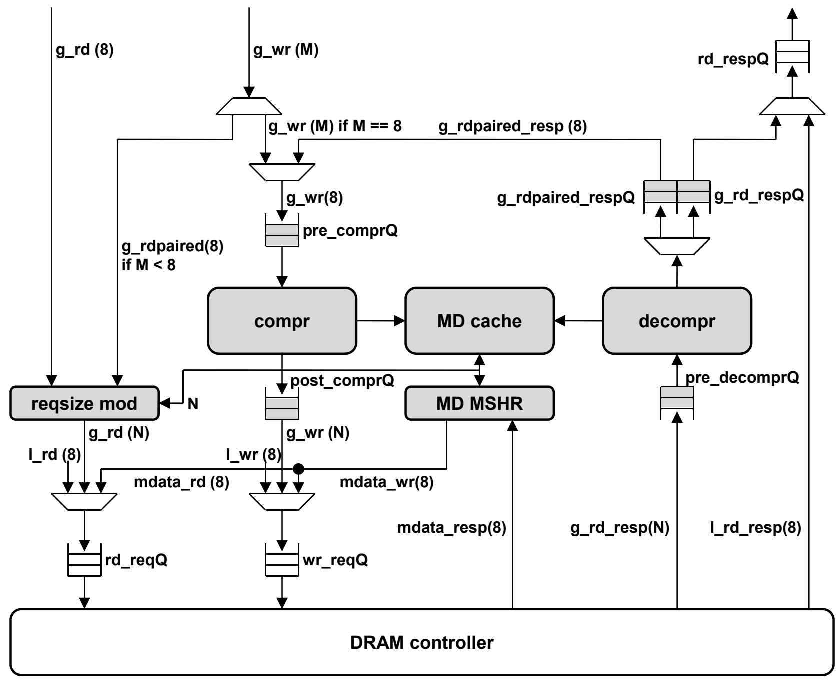


Figure 6: MC microarchitecture to support compression and decompression. The gray-color blocks are components added to an existing MC. The numbers in the parentheses represent the number of 16-byte chunks.

Table 1: Description of abbreviations used in Figure 6.

g_rd/wr (l_rd/wr)	global (local) read/write request by SM
g_rdpaired	global read request by either SM or MC as part of read-modify-write operation
md_rd/wr	MD line read/write request by MD MSHR in response to MD cache miss/dirty line eviction
g_rd_resp (l_rd_resp)	global data (local data) from DRAM in response to g_rd (l_rd) or g_rdpaired
g_rdpaired_resp	global data after decompression in response to g_rdpaired
md_resp	MD line from DRAM in response to md_rd
rd/wr_reqQ	A physical queue staging all read/write requests to the DRAM controller
pre/post_compQ	A logical queue staging g_wr(16) and g_wr(N) data before/after compression
pre _decompQ	A physical queue staging g_rd or g_rdpaired response data before decompression
g_rd/rdpaired_respQ	A physical queue staging g_rd/g_rdpaired response data after decompression
rd_respQ	A physical queue staging all read response data returning to SMs from MC

DRAM. However, a partial write, which is smaller than 128 bytes, needs to be converted to a read request (marked "*g_rdpaired*") followed by an update and write back with decompression and compression operations in between. Such an operation can be costly and is the main reason for potential performance degradations with a larger block size than the L1 cache line size (i.e., 128 bytes). The compressor also needs to update the corresponding metadata entry in the LUT after compression for the write. The request size modifier ("*reqsize mod*" in Figure 6) block is used to modify the read request size by reading the metadata cache to ascertain the number of compressed 16-byte chunks that need to be fetched from the GPU's off-chip memory, and thus plays an important role in the memory traffic reduction. Table 1 describes the abbreviations used in Figure 6.

4. Evaluation

4.1 Methodology

GPU architecture simulation: In this study, we analyze the performance of a GPU using GPGPU-Sim, which was validated against NVIDIA's Quadro FX 5800 and showed more than 90% performance correlation [12]. Our baseline GPU is configured to simulate NVIDIA's Quadro FX 5800 [13] with an enhanced number of streaming multiprocessors (SMs) and MCs to model recent high-end GPUs like the GTX580; see Table 2 for detailed simulation parameters. We modify GPGPU-sim to model a GPU architecture supporting our compression technique. Note that NVIDIA's GTX580 has 16 SMs operating while our baseline GPUs have 42 to 60 SMs, but GTX580 has 32 CUDA cores per SM while Quadro FX5800 has 8 CUDA cores per SM. That is, GTX580 has 512 CUDA cores, while our GPUs have 336 to 480 CUDA cores. We use 16 benchmarks from the GPGPU-Sim [12], Rodinia [14], Parboil2 [15] and ERCBench [16] benchmark suites. Their characteristics are summarized in Table 3.

Compressor/decompressor throughput, power, and area models: We assume that the architecture uses the compressor and decompressor hardware design proposed in [9], and our proposed microarchitectural enhancements described in Section 3. Table 4 summarizes the frequencies, throughputs per cycle, power consumption, and area of a compressor and a decompressor designed with a 65nm technology. The compressor is pipelined with a 3-cycle initial latency and it can compress 64 bits per cycle. The decompressor is not pipelined and it can decompress 128 bits per cycle. The combined power consumption of 8 decompressor and compressor pairs (one pair per MC) is 0.24% of the maximum power consumption of a Quadro FX 5800 that typically consumes 187W. Similarly, the area overhead is negligible (the Quadro FX 5800 die area is close to 500mm^2).

Metadata cache and its MSHR timing, power, and area models: We use CACTI 6.5 with the 65nm technology itrs-hp option to evaluate the cycle time, power, and area overhead of metadata caches and their MSHR tables. The area per 32-entry metadata cache with 128-byte line size is less than 0.05mm^2. The maximum cycle time is less than 800ps and the total power consumption is roughly 10mW at 800MHz.

4.2 Results

Performance improvement with compression /decompression using perfect LUT: Figure 7 shows the performance speedup trend as the throughput per cycle of decompressors is varied from 4 to 32 bytes per cycle; the maximum throughput per cycle of decompressors from [9] is 16 bytes per cycle at 1.2GHz while the MCs operates at 800MHz. Assuming a perfect metadata LUT and

Table 2: Simulation parameters (see [12] for details).

# of SMs	42~60	# of Memory Channels	8
SM Freq	1.30GHz	Memory Freqquency	800MHz (GDDR3)
On-chip Interconnect Freq	0.65GHz	Memory Bandwidth	102.4 GB/s
Warp Size	32	Bus Width per Channel	4 (Bytes/Cycle)
SIMD Width	8	Memory Controller	FR-FCFS
Max # of Threads per SM	1024	Branch Divergence	Immediate post dominator
Max # of CTAs per SM	8	Warp Scheduling	Round Robin
# of Registers per SM	16384	Const. Cache Size per SM	8 KB
L1$ Memory per SM	32KB	Texture Cache Size per SM	8 KB
Metadata cache	32 entries (128 bytes/entry)	Metadata MSHR	10 entries
pre _compQ	32 entries (8 bytes/entry)	pre/post_decompQ/ *_resQ	32 entries (16 bytes/entry)

Table 3: Benchmark summary.

Benchmark	Acronym	Property	Working Set Size	Working Set Data Composition
Back Propagation	BKP	memory-bound	55MB	single-precision FP
Fast Fourier Transform	FFT		165MB	single-precision FP
HotSpot	HOT		13MB	single-precision FP
JPEG encoder/decoder	JPG		7MB	single-precision FP
Leukocyte	LKT		20MB	single-precision FP / INT
3D Laplace solver	LPS		33MB	single-precision FP
Neural Network	NN		11MB	single-precision FP
Ray Tracing	RAY		76MB	single-precision FP / INT
Sparse Matrix Vector Multiplication	SPM		40MB	single-precision FP / INT
3D Stencil Operation	STN		328MB	single-precision FP
AES encryption/decryption	AES	compute-bound	4MB	INT
Magnetic Resonance Imaging Q	MRI		3MB	FP
Particle Filter	PF		116MB	double-precision FP
Sum of Absolute Difference	SAD		18MB	INT
SHA 1 encryption	SHA		7MB	INT
StoreGPU	STO		1MB	INT

Table 4: Hardware parameters for compressor/decompressor designed with a 65nm technology [9].

Compressor		Decompressor	
Maximum frequency	1.25GHz	Maximum frequency	1.2GHz
Throughput/cycle	64 bits	Throughput/cycle	128 bits (16 bytes)
Power consumption	32.63mW	Power consumption	24.14mW
Area	0.043mm^2	Area	0.043mm^2

16 bytes per cycle for the throughput of decompressors, the proposed technique improves the geometric mean of performance for memory-bound workloads by 26%. FFT and STN show good initial compression ratios (i.e., the compression ratio achieved during the initial data transfer from the CPU's DRAM to the GPU's DRAM) in Section 2, but their compression ratios become close to 1 as the workload progresses, which means little or no compression is achieved, as the execution of workloads progresses. This results in little or no improvement in performance for such workloads. The proposed compression technique does not improve the performance of compute-bound workloads notably, but it does not negatively impact their performance either; PF was categorized as a compute-bound workload in Section 2 where the bandwidth was increased to identify compute- and memory-bound workloads, but the compression technique improves its performance. This is because the compression technique also reduces DRAM access latency; we observe that PF performance increases when the frequency of MC is increased.

Impact of metadata cache size on performance: To analyze the impact of metadata cache size on the performance of GPUs, we vary the number of entries from 4 to 32. The miss rates are high if the number of metadata cache entries is either 4 or 8. However, the geometric mean of the miss rates become less than 2% if the number of metadata cache entries is 16. When the performance of a GPU using perfect on-chip metadata LUTs is compared to that of a GPU using metadata caches, we observe that the 16-entry metadata data caches result in a geometric mean performance degradation of roughly 1%. Note that most compute-bound workloads have relatively high miss rates. This is because their memory accesses are not frequent enough to exercise metadata caches, resulting in high miss rates caused by compulsory misses. Furthermore, 32-entry metadata caches have a negligible impact on performance relative to perfect on-chip metadata LUTs. This is because (i) data accesses by SMs show very high *spatial* locality, (ii) the metadata cache retrieves 128 bytes of metadata information per metadata cache miss, and (iii) a 32-entry metadata cache per MC exceeds the entire data caching capacity of our baseline GPU. Since we need only four-bit metadata for each 128B data block and the number of 128B data blocks covered by one metadata cache line is 128 × 8 bits / 4 bits = 256, each 128-byte metadata cache line contains the metadata for 256 × 128-byte data blocks (i.e., 32KB worth of data blocks). Thus, a 32-entry metadata cache in each MC can contain the metadata for 32 × 256 × 128-byte data blocks (i.e., 1MB worth of data blocks). Since each MC has one 32-entry metadata cache and our baseline GPU has total eight MCs, eight 32-entry metadata caches contain the metadata for 8MB worth of data blocks. Note that the total L1 capacity in our baseline GPU is at most 1.875MB for L1 caches (i.e., 32KB × 60 SMs) (and another baseline GPU supporting L2 caches has 2MB for L2 caches (i.e., 256KB × 8 MCs)). Thus, a 32-entry metadata cache per MC exceeds the entire data caching capacity of our baseline GPU.

Impact of the number of decompressors per MC on performance: An MC can access multiple DRAM banks. In such a case, multiple decompressors may reduce the latency of decompression. Otherwise, the 16-byte chunks from the second bank must wait until the decompression of all the 16-byte chunks from the first bank is completed. However, our experiments using decompressors with 16 bytes per cycle throughput show that the performance of a GPU with two decompressors per MC shows only marginal performance increase (~1%) compared to a GPU with one decompressor per MC across all the workloads we examine. This is because the throughput (i.e., bandwidth) of a single decompressor

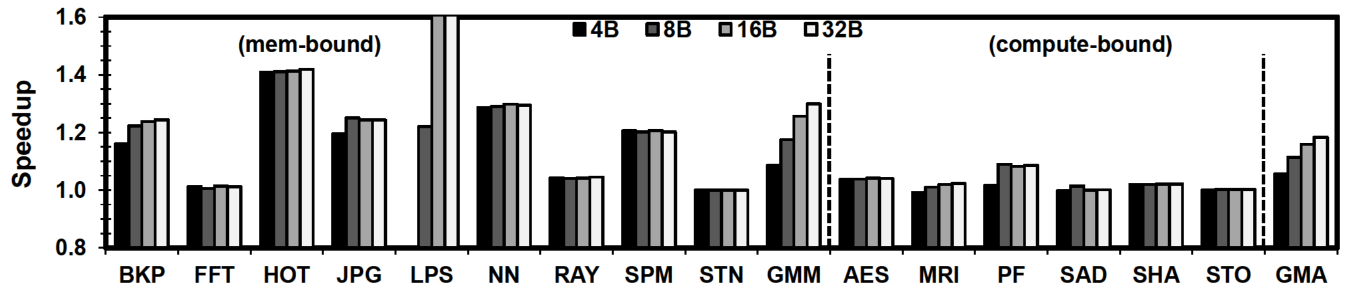


Figure 7. Performance speedup when varying throughput/cycle of decompressors. A perfect metadata LUT and 60 SMs are assumed.

is already twice as high as the memory bandwidth. In situations where the throughput of the decompressor is close to or less than the memory bandwidth, having multiple decompressors may be useful.

Impact of the peak bandwidth to peak compute ratio on performance: The peak bandwidth to peak compute ratio is a key measure to check whether or not the memory bandwidth is balanced with the compute capability. A GTX580 has 512 cores (16 SMs × 32 cores per SM) operating at 1.55GHz and it has a bandwidth over compute ratio (BCR) of 0.24, while our baseline GPUs have 336 to 480 cores (42 to 60 SMs × 8 cores per SM) operating at 1.3GHz and BCR ranging from 0.26~0.16.

Figure 8 shows performance speedup with the compression technique when varying the number of SMs in a GPU when we fix the memory bandwidth to the value shown Table 2; the speedup of 42-, 48-, 54-, and 60-SM GPUs with compression is relative to the performance of 42-, 48-, 54-, and 60-SM GPUs without compression, respectively. The overall relative improvement with 42 SMs is slightly lower than with 60 SMs, which demonstrates that the compression technique can still be effective for a GPU with higher BCR since most memory-bound workloads congest the MCs even with much fewer SMs. These memory-bound workloads can easily congest the MCs even with 48 SMs; Lee *et al.* [17] show that just 32 SMs could generate enough memory accesses to congest the MCs. The geometric mean of speedup for all the memory-bound workloads for 42, 48, 54, and 60 SMs is 22%, 24%, 27%, and 26 %, respectively. Note that the speedup trends of some workloads such as FFT, JPG, and RAY have slight ups and downs as the number of SMs increases. This is because the performance of the baseline 42-, 48-, 54-, and 60-SM GPUs, which do not support compression, does not increase linearly with more SMs. Finally, the BCR is lower for our baseline GPU with 60 SMs than the GTX580, but we expect that future GPUs will have lower BCRs.

Performance improvement with L2 caches + compression/decompression: Figure 9 plots the speedup when varying L2 cache size. We measure the performance speedup of a GPU supporting the proposed compression technique (32-entry MD caches and 16 bytes per cycle decompressor throughput) with 32KB, 64KB, 96KB, and 256KB L2 caches per MC (total 256KB, 512KB, 768KB, and 2MB L2 caches); the NVIDIA Fermi architecture supports a 768KB total L2 cache capacity [11]. Then, we plot the performance speedup of the GPU relative to our baseline GPU that neither has L2 caches nor supports the compression technique. The geometric mean performance speedup of the memory-bound workloads is about 88% when a GPU with a 768KB L2 cache supports the compression technique. For roughly half of workloads we examined (i.e., JPG, NN, BKP, HOT, LPS, and SPM), the performance of a GPU supporting the compression technique with 768KB L2 caches is higher than or comparable to that of a GPU with 2MB L2 caches only. To isolate the performance increase contributed by the compression technique, we stack the speedup numbers shown in Figure 9 on top of the speedup numbers obtained with a GPU that has the same size L2 caches but does not support the compression technique. JPEG, RAY, BKP, HOT, LPS, and SPM show notable improvements over using L2 caches only. The compression technique increases the geometric mean of performance of a GPU with a 768KB L2 cache by 37% for memory-bound workloads.

Note that the compression technique without L2 caches improves the performance by only 26% while the improvement with the compression technique and L2 caches is nearly 37%. This can be explained by the following observations we made based on simulation statistics. First, the L2 miss rates of some workloads are reduced with the compression technique, because highly compressed blocks arrive earlier than their uncompressed counterparts, and the subsequent memory accesses for the blocks do not experience misses. Second, L2 caches filter some or many accesses to the DRAM, reducing the contentions for accessing the memory channels. This allows the MCs and the compression technique to work

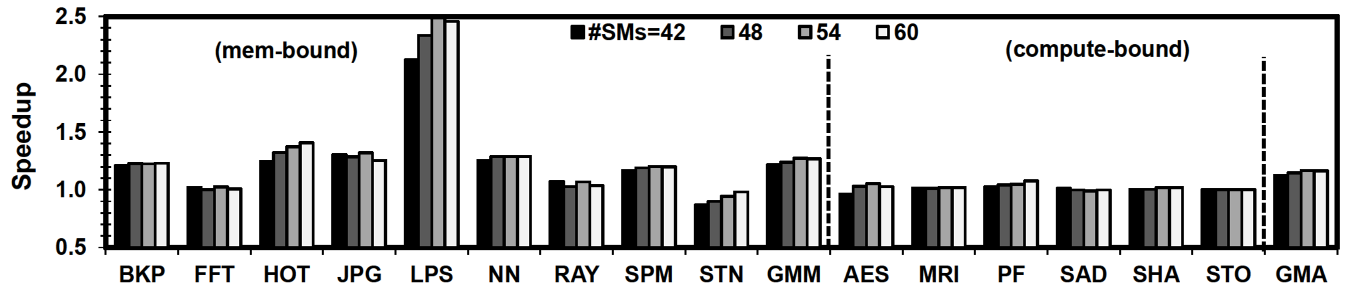


Figure 8. Performance speedup when varying the number of SMs to analyze the effectiveness of the compression technique for different bandwidth-compute ratio values.

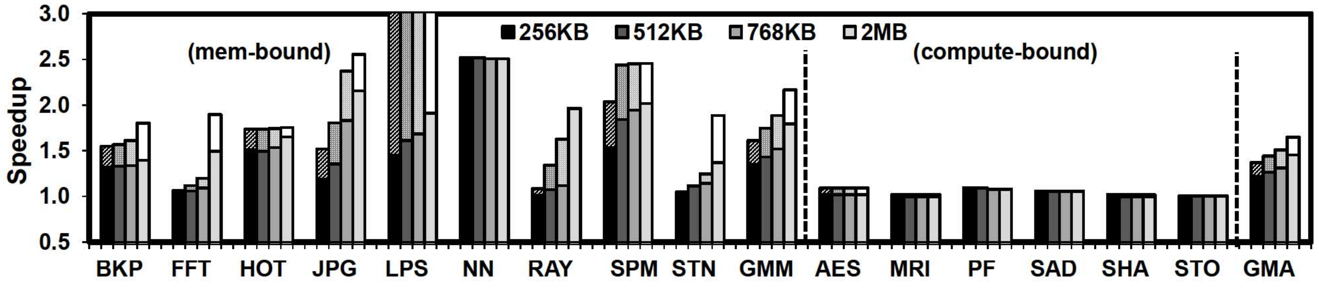
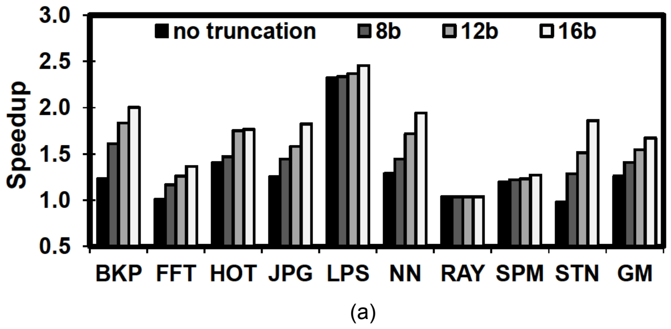


Figure 9. Performance speedup when varying L2 cache size combined with the compression/decompression technique. The results are normalized to a GPU without L2 caches or the compression/decompression technique.



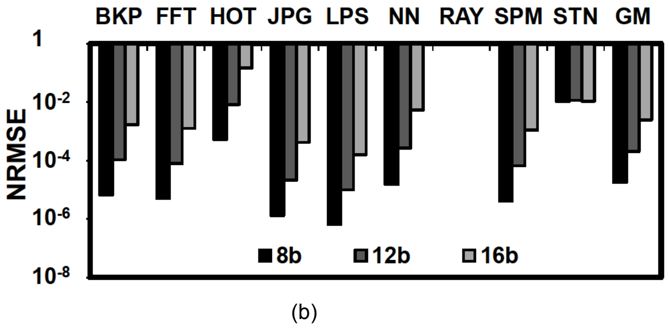


Figure 10: (a) Speedup when varying the number of LSBs truncated and (b) normalized root mean square error (NRMSE) of all the final data transferred back to the CPU at the end of kernel executions.

more efficiently for servicing the same number of accesses. On the other hand, if the MCs are flooded with too many concurrent requests, their service efficiency goes down substantially and the compression technique shows less benefit. Finally, the compression technique improves the effective memory bandwidth and latency. This leads to less memory contention and allows L2 caches to handle misses more efficiently, leading to a synergistic effect of more efficient services for memory requests.

Impact of lossy compression on performance and overall computing accuracy: Figure 10(a) shows the performance speedup as the number of truncated LSBs increases for the memory-bound workloads. The GPU with more LSBs truncated provides higher performance improvement for most workloads except for RAY; the performance of RAY is limited by many simultaneous memory accesses rather than high memory bandwidth usage and the truncation does not improve the performance since the LSBs of the data are already zero. The compression after truncating 8~16 LSBs improves the geometric mean performance by 41%~61%, which is 15%~41% higher than the lossless compression alone. Meanwhile, as demonstrated in Figure 10(b), the impact on overall computing accuracy (i.e., the geometric mean of normalized root-mean-square error (NRMSE)) is 10^{-5} to10^{-3} for truncated single-precision data; we obtain the NRMSE values for all the final data transferred back to the CPU at the end of the kernel executions using double precision arithmetic. Note that RAY shows no error even after the lower 16-bits are truncated since the original input data applied to the FP units have zeros in their 16 LSBs. This is because RAY only utilizes the upper 8 bits of the mantissa for its input data. Thus, RAY is excluded when calculating the geometric mean of the NRMSE values.

5. Related Work

Hardware-based memory data compression techniques have been primarily employed either to increase the effective memory size by memory compaction, or to increase the effective cache size. Chen *et al.* used a variant of the Lempel-Ziv (LZ) compression algorithm to propose a scheme that dynamically partitions the cache into sections of different compressibility [3]. Alameldeen *et al.* showed that an adaptive dynamic scheme that chooses between holding compressed or uncompressed data in the L2 can improve the performance of memory-intensive workloads while constraining the performance degradation of other workloads to an acceptable level [4]. Zhang *et al.* show that supplementing a direct mapped cache with a small frequent value cache can greatly reduce the cache miss rate [18]. The frequent value cache is basically a smaller direct-mapped cache dedicated to holding frequent benchmark values. IBM's XMT technology employs a hardware parallelized derivative of the LZ sequential algorithm to achieve on the fly content compression [5]. The input data is divided into 1KB blocks before applying parallelized compression. However, their primary goal was not to improve the memory bandwidth but to increase the main memory capacity due to data compaction.

Some researchers have also employed hardware-based compression to reduce communication bandwidth between general purpose processors and off-chip DRAM. Benini *et al*. proposed a data compression scheme to reduce memory traffic in general purpose processor systems [19]. Data is stored in a compressed form in the main memory but uncompressed in the cache. They use a differential compression scheme to achieve on-the-fly compression (decompression) of data from (to) the cache to (from) the DRAM. Finally, Thuresson *et al.* quantified what type of value locality is exploited by each compression scheme and demonstrated that a new compression scheme exploiting value locality can free up a considerable percentage of the memory bandwidth [6]. Montrym and Moreton described the architecture of the GeForce 6800 [2]. Although they mentioned that compression techniques are used for texture depth and color data, but they neither disclose any microachitecture details for supporting the compression techniques nor provide a performance analysis.

6. Conclusion

Although GPUs support very high memory bandwidth and hide long memory access latency using many in-flight threads, the performance of many GPU workloads is still memory-bound. Furthermore, improving the bandwidth and latency is often limited by package, interconnect, and power constraints. Facing such challenges, in this paper, we propose a GPU architecture supporting hardware-based memory I/O link compression techniques for the data residing in the GPU's DRAM. First, we demonstrate that many GPU workloads are memory-bound but their data exhibit high compression ratios. Second, we show that GPUs assisted by high-throughput and low-latency compressors and decompressors integrated with metadata caches can improve the performance of many memory-bound workloads by 26% on average for GPUs with L1 caches but without L2 caches. We also show that the power and area of the proposed compression techniques are negligible compared to those of a GPU. Third, we evaluate compression support for GPUs with integrated L2 caches. Due to the synergistic effects between L2 caches and the compression technique, the performance of a GPU adopting 768KB L2 caches and the proposed compression technique is (i) an average of 37% higher than that of a GPU with the same size L2 caches but without the compression technique, and (ii) comparable to that of a GPU with 2MB L2 caches and no compression in many memory-bound workloads that we examined. Fourth, we investigate how a lossy compression technique can be applied to GPGPU workloads and evaluate its effectiveness. This technique leverages the observation that the reduced precision of FP data in GPGPU workloads does not notably impact the overall computing accuracy; truncating the 8 LSBs before the compression improves the performance by 41% while the RMSE for the final results is less than 10^{-3}. Finally, although compression techniques have been adopted by commercial GPUs, they are only used for compressing texture and color data, not data for GPGPU applications; the microarchitectural details for those GPU compression techniques are proprietary and their performance benefits have not been previously published.

Acknowledgement

We would like to thank Prof. David Wood (the University of Wisconsin-Madison), Mike O'Connor (AMD), and Jay Owen (AMD) for their valuable comments and supports. We also would like to thank Prof. Robert Dick and Xi Chen who assisted us to evaluate his compression hardware for this study. This work is supported in part by an SRC grant (Task ID: 2080.001), NSF grants (CCF-095360, CCF-1016262, CNS-1217102), Hiran Mayukh Award, IBM Faculty Award, and generous gift grants from AMD.

References

[1] Rambus, "Challenges and Solutions for Future Main Memory," 2009.

[2] J. Montrym and H. Moreton, "The GeForce 6800," *IEEE Micro*, vol. 25, no. 2, pp. 41-51, Mar-Apr 2005.

[3] D. Chen, E. Peserico, and L. Rudolph, "A Dynamically Partitionable Compressed Cache," in *Singapore-MIT Alliance Symp.*, 2003.

[4] A.R. Alameldeen and D.A. Wood, "Adaptive Cache Compression for High Performance Processors," in *IEEE/ACM Int. Symp. on Comp. Arch. (ISCA)*, 2004, pp. 212-223.

[5] B. Abali *et al.*, "Memory Expansion Technology (MXT): Software Support and Performance," *IBM J. Research and Development*, vol. 45, no. 2, pp. 287-301, Mar 2001.

[6] M. Thuresson, L. Spracklen, and P. Stenstrom, "Memory-Link Compression Schemes: A Value Locality Perspective," *IEEE T. on Computers*, vol. 57, no. 7, pp. 916-927, Jul 2007.

[7] T. Yeh, G. Reinman, S. Patel, and P. Faloutsos, "Fool me twice: Exploring and exploiting error tolerance in physics-based animation," *ACM Trans. Graph.*, vol. 29, no. 1, p. Article 5, Dec 2009.

[8] J. Tong, D. Nagle, and R. Rutenbar, "Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic," *IEEE T. on Very Large Scale Integr. (VLSI) Syst.*, vol. 8, no. 3, pp. 273-285, Jun 2000.

[9] Xi Chen, Yang L., R.P. Dick, Li Shang, and H. Lekatsas, "C-Pack: A High-Performance Microprocessor Cache Compression Algorithm," *IEEE T. on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 196-1208, Aug 2010.

[10] GDDR3 Specific SGRAM Functions. [Online]. http://www.jedec.org/standards-documents/docs/sdram-3110507

[11] C.M. Wittenbrink, E. Kilgariff, and A Prabhu, "Fermi GF100 GPU Architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50--59, Mar 2011.

[12] A. Bakhoda, G. Yuan, W W.L. Fung, H. Wong, and T.M. Aamodt," Analyzing CUDA Workloads using a Detailed GPU Simulator," in *IEEE Int. Symp. Perf. Analysis of Syst. and Software (ISPASS)*, 2009, pp. 163-174.

[13] Specification for Nvidia Quadro FX 5800. [Online]. http://www.nvidia.com/object/product_quadro_fx_5800_us.html

[14] S. Che *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE Int. Symp. on Workload Characterization (IISWC)*, 2009, pp. 44-54.

[15] Parboil Benchmark Suite. [Online]. http://impact.crhc.illinois.edu/parboil.php.

[16] D.W. Chang et al., "ERCBench: An Open-Source Benchmark Suite for Embedded and Reconfigurable Computing," in *IEEE Int. Conf. on Field Programmable Logic and App. (FPL)*, 2010, pp. 408 -413.

[17] J. Lee, P. Ajgaonkar, and N.S. Kim, "Analyzing throughput of GPGPUs exploiting within-die core-to-core frequency variation," in *IEEE Int. Symp. on Performance Analysis of Syst. and Software (ISPASS)*, 2011, pp. 237-246.

[18] Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-centric Data Cache Design," in *ACM Int. Conf. Arch. Support for Programming Lang. and Operating Syst. (ASPLOS)*, 2009, pp. 150-159.

[19] 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 *IEEE Int. Conf. on Circuits and Syst (ICCAS)*, May 2002, pp. 866-869.