Future Generation Computer Systems 30 (2014) 170–178

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Optimizing LZSS compression on GPGPUs

Adnan Ozsoy*, Martin Swany, Arun Chauhan

School of Informatics and Computing, Indiana University, Bloomington, IN 47405, USA

ARTICLE INFO

Article history:
Received 1 January 2013
Received in revised form
16 April 2013
Accepted 17 June 2013
Available online 17 July 2013

Keywords:
Lossless data compression
LZSS
GPU
CUDA

ABSTRACT

In this paper, we present an algorithm and provide design improvements needed to port the serial Lempel–Ziv–Storer–Szymanski (LZSS), lossless data compression algorithm, to a parallelized version suitable for general purpose graphic processor units (GPGPU), specifically for NVIDIA's CUDA Framework. The two main stages of the algorithm, substring matching and encoding, are studied in detail to fit into the GPU architecture. We conducted detailed analysis of our performance results and compared them to serial and parallel CPU implementations of LZSS algorithm. We also benchmarked our algorithm in comparison with well known, widely used programs: GZIP and ZLIB. We achieved up to 34× better throughput than the serial CPU implementation of LZSS algorithm and up to 2.21× better than the parallelized version.

1. Introduction

Lossless data compression can be used to reduce storage requirements as well as improve data communication performance by reducing the amount of data that needs to be transferred over an I/O channel. These improvements in storage and communication come at the cost of increased processing requirements in compressing and decompressing the data. Lossless compression algorithms are computationally intensive, and thus while they may reduce the volume of data, they may not improve overall I/O performance. However, with a growing number of applications working with huge amounts of data, the problem of scaling data movement has gained critical importance, and compression at the producers and consumers of the data may be the only viable option. In this paper, we present an investigation of using GPU cards designed for general purpose computing (GPGPUs) to accomplish fast data compression and decompression, freeing the CPU to perform the main computation.

Several compelling reasons make GPGPUs attractive for performing data compression. First, GPGPUs are asynchronous coprocessors, able to run user tasks concurrently with the CPU, enabling latencies of compression tasks to be hidden through careful pipelining. Second, large memory bandwidth with direct memory access (DMA) makes it possible to transfer large amounts of data between the CPU main memory and the GPU memory. Third, direct channels between multiple GPGPUs and between GPGPUs and I/O devices on emerging systems provide a potential to improve I/O performance further by offloading I/O communication to the coprocessors. Finally, emergence of supercomputing architectures based on hybrid CPU–GPU nodes has increased the availability of GPGPUs, making an effort to leverage them worthwhile.

Unfortunately, lossless compression is not inherently parallel, creating a challenge in porting the compression algorithms to GPGPUs that achieve high performance primarily through large-scale fine-grained data parallelism. In this paper we present a pipelined parallel lossless compression algorithm that can attain rates suitable for data compression *on the fly*. We achieve high performance by creating a software pipeline between CPU and GPU and by tuning the algorithm for the hardware idiosyncrasies of current GPGPUs, including their non-traditional memory hierarchy and hardware coalescing of regular vectorized reads from memory.

One of the most popular lossless data compression algorithms is Lempel–Ziv–Storer–Szymanski (LZSS), which belongs to the LZ family of algorithms. The core idea in the LZ family of compressors is in finding redundant strings and replacing them with references to previously visited copies. LZSS is a variation of LZ77 that uses fewer bits to indicate encoding [1,2]. We chose the LZSS compression algorithm for its one-pass characteristic, as unlike several other compression multi-pass algorithms [3,4] a single pass allows stream processing. Our algorithm is based on the preliminary work published earlier on LZSS, called CULZSS [5].

We make the following contributions:

1. Identification of several optimizations to improve the performance of LZSS on GPGPUs.
2. A software-pipelined version of LZSS, suitable for leveraging GPGPUs for data compression in a streaming data scenario.

* Corresponding author. Tel.: +1 3026908685.
E-mail addresses: aozsoy@indiana.edu, adnanozsoy@gmail.com (A. Ozsoy), swany@cs.indiana.edu (M. Swany), achauhan@cs.indiana.edu (A. Chauhan).

0167-739X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.future.2013.06.022

3. Making the GPU-enabled LZSS tunable by exposing a selectable trade-off between compression rate and compression ratio.
4. A fine-grained overlapping version to fully exploit the GPU and CPU hardware resources.
5. Performance evaluation of the algorithm, compared to CPU-based parallel and sequential implementations of LZSS algorithm, as well as with GZIP and ZLIB—two well known compressors that use LZ-based algorithms.
6. CPU-based Huffman coding extension to the GPU software-pipeline for compression ratio improvement.

We begin with an overview of prior and related research work.

2. Related work

Lossless data compression is a broad field that has been investigated by many research communities. There is a large body of work on compression algorithms and their applications. In this section we only focus on GPU-based efforts on lossless compression.

Balevic investigated run-length encoding (RLE) on GPUs [6]. They got improved performance on GPU, although, compared only to a serial version of RLE on the CPU. Our goal has been to achieve better performance in our GPU implementation than multi-threaded compression running on a modern multi-core CPU.

Some previous efforts have proposed GPU-based lossless compression targeting specific types of data. Fang et al. proposed RLE-based compression for compressing databases, leveraging certain redundancy elimination techniques, on GPUs [7]. O'Neil et al. introduced an algorithm specifically for compressing double-precision floating point data, called GFC [8]. Our algorithm works on raw bytes and we present evaluation results on data from a variety of sources.

Cloud et al. studied Huffman encoding on GPUs [9], and while Patel et al. presented the steps needed to port the BZIP2 algorithm to GPUs, they achieved no performance improvement even over the serial CPU implementation [10]. In general, compression algorithms relying on statistical modeling, such as Huffman coding, seem to be harder to optimize for GPU architectures than dictionary-based approaches, such as LZSS. In LZSS, compression proceeds in chunks that are independent, unless the chunks reside in the same buffer. This allows the chunks to be distributed and compressed concurrently. Moreover, LZSS does not require multiple passes over the data, which is the case for statistical encoding, making LZSS a good candidate for building a streaming compressor.

The work in this paper builds on CULZSS, by Ozsoy and Swany, which is a parallel LZSS algorithm for GPUs, using CUDA [5]. We substantially improve the basic CULZSS algorithm by reducing control-flow divergence, making better use of the CUDA memory hierarchy, and overlapping CPU and GPU execution. We explore several versions that fit different possible scenarios. Finally, we transformed the algorithm to work in a software pipeline across the CPU and the GPU, enabling it to operate on a stream of data.

3. Background

3.1. LZSS algorithm

The original LZSS algorithm has two main stages that are visited repeatedly for every character in the file until the file is completely encoded.

The first stage finds the matching information for a given string. It works by searching the previously encoded data, called sliding history buffer. Initially there is no history and the buffer is filled with predetermined data. The character at the current position and the subsequent uncoded characters are also put into another buffer, called uncoded lookahead buffer. For each character in the

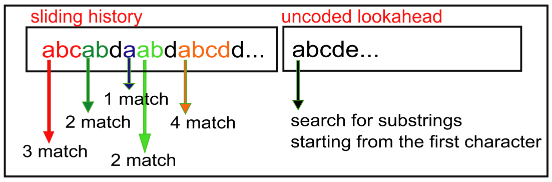

Fig. 1. An example of the matching stage.

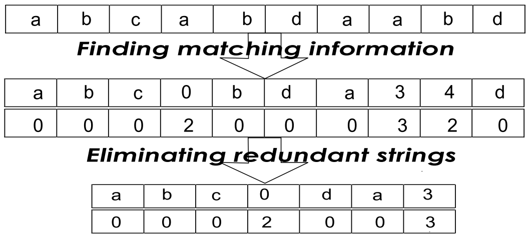

Fig. 2. An illustration of eliminating the redundant data.

file, the searching process looks for the longest substring in the sliding history buffer that matches the uncoded lookahead buffer, starting with that character.

Fig. 1 shows an example string and how this stage finds matches. In the example, an uncoded lookahead buffer has character 'a' at the beginning and search looks into the sliding history buffer for substrings starting with 'a'. The first match is the first character of sliding history and the following 'bc' characters also match with uncoded lookahead. We call the three-character match a 3-matching. The algorithm proceeds along the buffer to find all matches and keeps track of the longest one. In the example, the longest one is a 4-matching.

The second stage stores the encoding information. The match needs to be long enough to amortize the cost of encoding; otherwise, the character is written as it is.

To parallelize these stages, two versions have been proposed in CULZSS by Ozsoy and Swany [5]. The first stage is parallelized by dividing the file into chunks and distributing them to CUDA thread blocks. In each block, the thread reads one character of the uncoded lookahead buffer at a time and looks for matching substrings that start with that character. After these stages, the generated data consists of each character's matching information. However, there is redundant information that needs to be eliminated. Fig. 2 illustrates the elimination step. Our previous work describes the strategy in greater detail [5]. We discuss some of the details relevant to this paper in Section 4.

3.2. Related GPU architecture

GPU architectures have been widely studied and described in several published documents [11–14]. Here, we only discuss the key GPU capabilities and characteristics that influenced our approach.

One of the main important architectural components that needs special attention in CUDA programming is memory. CUDA exposes five memory types: global, constant, texture, shared memory and registers. Global memory is relatively large, but slow. Texture and constant memories are cached by the hardware and are read-only. These three memory types are accessible by all threads. Shared memory is relatively fast, but only accessible by the threads in the same block. The limited size of shared memory as well as thread-private registers makes it necessary to pay special attention to their usage [12].

Another critical component is thread scheduling. In the CUDA architecture, threads are scheduled and executed in groups of

32, called *warps*. The thread blocks are partitioned into warps and then scheduled for execution by a warp scheduler. The SIMD architecture requires that each thread in a warp executes the same instruction at a given time. A thread-dependent control-flow divergence, such as one caused by a data-dependent conditional branch, causes certain threads in a warp to be temporarily disabled. When the threads exit the branch, the paths merge and the threads resume on the same execution path [12]. Understanding this behavior is critical in optimizing code on GPUs.

The CUDA framework supports asynchronous concurrent execution between CPU and GPU. This provides the opportunity to overlap computation on the CPU and GPU, letting the GPU be treated as a coprocessor on which portions of computational work may be offloaded. It also supports asynchronous execution inside the GPU for memory copies and kernels. The necessary properties in the GPU that are needed to have concurrent execution are *asyncEngineCount* and *concurrentKernels*. Concurrency is managed through *CUDA streams*. The sequence of commands that execute in order forms a CUDA stream for the same data. On the other hand, different CUDA streams may follow different execution orders with respect to one another [12]. The use of the word "stream" was appeared in two different concepts throughout the paper. One usage is in the concept of continuously upcoming, flowing data. The other is in the CUDA framework where streams are used to manage concurrent execution. We refer to the former as "streaming data", and to the latter as "CUDA stream" to avoid the confusion in different sections.

These properties can be queried through the CUDA API. For devices that have a value greater than zero for asyncEngineCount can overlap data transfer and kernel execution. The number of engines can be one or more depending on the product line. Asynchronous engine count of one shows that the device can only do overlapped data transfer and kernel execution, where two engines show the ability to concurrent data transfers to and from GPU as well. The concurrentKernels feature shows the ability to run multiple kernels at the same time.

The given properties of the device and the order of asynchronous commands determine the possible amount of execution overlap. CUDA Framework defines implicit synchronization rules when two different CUDA streams are able to run concurrently. The CUDA programming guide lists a number of operations that prevent two commands from different CUDA streams from running concurrently if any of them is called in between. The listed operations are page-locked host memory allocation, device memory allocation, a device memory set, a memory copy between two addresses to the same device memory, any CUDA command to the default CUDA stream, or a switch between the L1/shared memory configurations [12].

The CUDA development environment also includes a CUDA profiler. We used the CUDA profiler to get feedback for memory-access efficiency and per-function performance metrics, which we discuss in Section 6.

4. Optimizing GPU performance

This section describes our improvements to Ozsoy and Swany's CULZSS algorithm [5], driven by optimizations for memory hierarchy and eliminating control-flow divergence.

4.1. Algorithmic improvements

As discussed in Section 3.2, a warp of threads executes in lockstep; therefore, the execution is most efficient when all threads of a warp agree on their execution path [12]. Divergent control flow in the threads of a warp in CULZSS reduces the efficiency of its execution. This occurs mainly in the matching phase of CULZSS.

If there is a match in the first character, then CULZSS reads the subsequent characters from the uncoded lookahead buffer in a loop until it reaches a non-matching character. After the process finishes, the encoded information is recorded. Algorithm 1 outlines the steps. However, this approach results in divergent paths in the execution of individual CUDA threads, since if one thread has a match, it needs to continue executing the loop to find the maximal match. The threads that do not continue are disabled by the hardware until the looping threads converge back by eventually exiting the loop. In the worst case, only one thread might be in the matching loop for every run and all other threads might be idle, resulting in significant performance loss.

Algorithm 1 Matching - CULZSS

```
while $i < buffer\_length$ do
    if $history[i + tid] == lookahead[j + tid]$ then
        while MATCH do
            $j = j + 1$ //Check for more matches
        end while
        Update MAX matching
    end if
    $j = 0$ , $i = i + 1$ //Reset matching information
end while
Store matching information
```

We eliminate the divergence in control flow by using a strategy that we call *state-based matching*. Each thread can be in either a matching or a non-matching state. If there is a match for the given character and the thread is not in matching state, it enters the matching state and records the location of the matched character. If the thread is already in the matching state, it increments the length of the match. If the given character is not a match and the thread is not in the matching state, the thread continues the search. If a thread is in matching state and hits a non-matching character it indicates the end of the match. The thread records the length and location information, if the just-concluded match is longer than the earlier longest match. Algorithm 2 outlines these steps.

This design eliminates the inner loop when there is a match and greatly reduces control-flow divergence. The shared-memory access is improved for each thread by accessing only one character per step. It also improves data access times since it reads a block of data once, which is then consumed by all threads.

Algorithm 2 Matching - Optimized

```
while $i < buffer\_length$ do
    if $history[i + tid] == lookahead[j + tid]$ then
        $j = j + 1$ //Increment match length
    else
        Update MAX matching
        $j = 0$ //Exit matching state
    end if
    $i = i + 1$
end while
Store matching information
```

4.2. Architecture-oriented optimizations

Memory choice. We utilize texture memory in order to leverage hardware caching. This results in significant performance improvement. Unfortunately, the maximum width for a 1D texture reference bound to linear memory is limited; usually much smaller than total GPU global memory available—for the GPU cards that we used (Compute Capability 2.0), it is up to 2^{27} bytes (128 MB) [12]. This limits its use to relatively small data sizes. However, Section 5 describes how software pipelining can get around this limitation.

Data structure choice. Compression rate (throughput) and compression ratio (amount of compression) are two opposing goals in most compression algorithms. A network that needs to keep up with a fast communication channel may use an algorithm optimized for high compression rate, at the cost of lower compression ratio. In another scenario, a dedicated backup engine may have space constraints, but could tolerate longer execution times and would opt for a better compression ratio.

We have designed our algorithm to make these tradeoffs available to users. In order to achieve higher compression ratio, we expand the search space, which in return increases the chance of finding a longer match. This is implemented by increasing the size of the history buffer. Our implementation provides three versions with history buffer sizes of 512, 256, and 128 bytes.

The other direction of the tradeoff is increasing compression rate by potentially giving up some compression ratio. The straightforward approach of decreasing the search space by decreasing the history buffer size turns out to be inadequate. If the buffer size is too small the compression algorithm fails to achieve any significant compression because there is a limited chance of finding enough matches to amortize the cost of encoding. We found that we needed a history buffer at least 128 bytes long in order to achieve compression. Standard techniques to improve search time, such as sorting and hashing, are not useful on GPUs due to prohibitively high cost of dynamic allocation in CUDA. These techniques did not result in any performance improvement in our experiments.

We observe that a match only counts as a match if there are enough characters to amortize the cost of the encoding. In our case, the minimum count is two bytes. Thus, a single character match is inconsequential. Nevertheless, the algorithm searches for matches on single characters. Instead, if we search on two bytes (`short int` type instead of a `char` type) we expect the search time to be reduced by half. However, we lose some opportunities for matches that start with the second character of the two-byte groups. For example, if the history buffer is {abcdabdeac} and the lookup buffer is {abdeadde}, the algorithm searching on substrings of two characters sees the history buffer to be {ab, cd, ab, de, ac} and the lookup buffer to be {ab, de, ad, de}. A thread could find the match for {abde} in two lookups, instead of four. The next thread would start the search from the next substring {de}, not the next character {b} after {a}. The problem with this approach is that the best match is not four characters long, but five: {abdea}. The search performed on two characters at a time fails to catch this longest match.

This can be prevented by having substrings starting with each character, e.g., {ab, bc, cd, da, ab, bd, de, ea, ac}. This change can be applied to either one or both of the history and the lookahead buffers. We evaluated three additional versions of our algorithm that differ based on which buffers are changed: both buffers changed (128_2), one buffer changed (128_1) and none changed (128_0).

5. Pipelined parallel LZSS

We have extended our optimized parallel LZSS algorithm to work in a software pipeline across the CPU and GPU. This section describes the extension that leads to a fully pipelined LZSS suitable for compressing streaming data.

5.1. Computation overlap

In the CULZSS algorithm, a portion of the work remains that must be handled by CPU after the GPU kernel executes on a given piece of data. This provides the opportunity to overlap GPU and CPU computation to better utilize the existing hardware in the system. The problem, however, is that the CPU computation is dependent on the GPU-generated data. In the GPU, the encoding information for the data needs to be identified and afterwards the data can be compressed with that information on the CPU. Our approach changes this limitation to an opportunity by dividing the data into chunks and processing them in a pipelined fashion. After the GPU finishes processing a chunk the post-CPU computation for that chunk is overlapped with the GPU computation on the next chunk, as illustrated in Fig. 3. The analysis of the multiple kernel invocation overhead is given in Section 6.

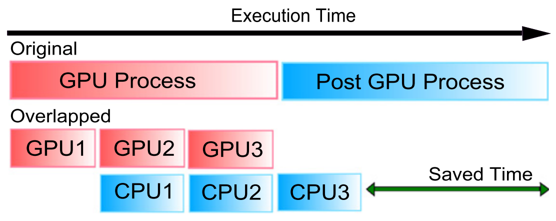

Fig. 3. Overlapping GPU and CPU computation.

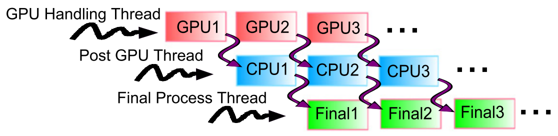

Fig. 4. GPU processing pipeline for streaming data flows.

5.2. The GPU processing pipeline

In order to fully utilize the cores on multi-core CPUs we use thread-level parallelism for the CPU portion of the pipeline. The first idea is dividing the data into equal portions and assigning a thread for each portion. This data-parallel approach requires each CPU thread to access the GPU. Even though multiple concurrent CPU threads may access the GPU on the newer CUDA compute-capable devices [12], it can result in significant performance degradation [15]. Additionally, this approach requires a synchronization step to serialize the GPU access by multiple threads, with a consequent negative performance impact.

An alternative approach is assigning different tasks to CPU threads, instead of using all threads for each CPU task, effectively using a task-parallel strategy instead of a data-parallel approach. One thread is dedicated to handle communication with the GPU (GPU handler), another for post-GPU work (Post-GPU) that needs to be done on CPU, and a third for dispatching the compressed data to its destination (Final Process). A thread pool is used to reduce overheads. Fig. 4 illustrates this design, which establishes a processing pipeline capable of compressing data arriving in a stream.

The pipeline can be refined based on available CPU cores. For example, we can add another thread to receive uncompressed data. We can also dedicate multiple threads to perform a single task, thus using a mixed data- and task-parallel approach, if needed. In our performance evaluations we use multiple data-parallel threads to perform the post-GPU task.

5.3. Fine grained overlapping

The asynchronous calls on GPU can help to hide the kernel and memory copy latencies. It can also lead to concurrent executions of CPU–GPU workloads and also two GPU workloads that belong to two different steps of the streaming pipeline. In the streaming data scenario, each chunk of data is being processed by GPU and later moved to CPU processing while next chunk will be loaded to GPU for processing. In a fully asynchronous approach, we can combine

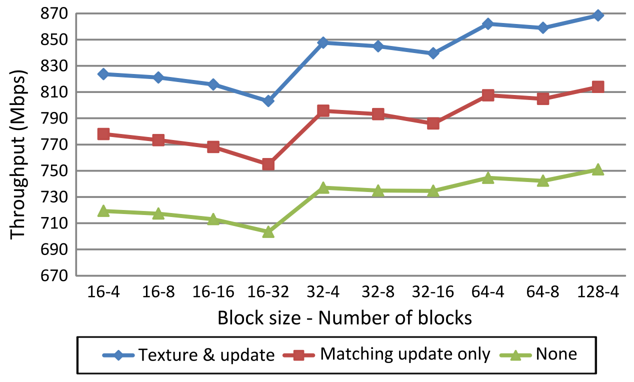

Fig. 5. Texture memory usage and updated matching step results.

the overlapping of the different chunk processing, the concurrent execution of the GPU *memcopy* calls for different chunks, and processing the same chunk in smaller pieces to overlap multiple kernel runs.

5.4. Multiple GPUs

Multiple GPUs on the same machine are supported by NVIDIA through the Scalable Link Interface (SLI) technology.[1] Multi-GPU nodes are increasingly common on supercomputers. Our algorithm lends itself easily to such configurations. The pipeline described above can be easily scaled to communicate with multiple GPUs. Our evaluation of the algorithm with two GPU cards showed nearly linear performance improvement, as Section 6 discusses.

6. Performance evaluation

6.1. Testbed configurations

To evaluate how well our implementations work, we used both the "Delta" GPU cluster in FutureGrid [16,17] and standalone machines in our lab. Each Delta cluster node has two Tesla C2075 cards and two Intel X5660 processors (6 physical, 12 logical cores) at 2.8 GHz, 192 GB of memory, running Red Hat RHEL 6.2. The Tesla C2075 has 448 cores at a GPU clock speed of 1.15 GHz with a 5375 MBytes of global memory. They are running CUDA version 4.1.

The other machines consist of a GeForce GTX 480 card with CUDA version 4.1 installed on each machine with one AMD Phenom II X6 1100T processor (6 cores) running at 3.3 GHz. These machines are used to test the implementation on the GTX card, which has 480 GPU cores (slightly more than the Tesla C2075) and a faster clock speed (1.4 GHz), but smaller global memory (1535 MBytes).

6.2. Parallel implementations on CPU

Our serial CPU implementation of LZSS was mainly adapted from Dippersteins work [18]. We implemented a parallel CPU version of the LZSS algorithm using POSIX threads for comparison. We note that the algorithm does not necessarily have to be better on the GPU for it to be useful in a streaming situation. As long as it delivers adequate throughput to sustain the pipeline, it frees up the CPU for other tasks, enabling the GPGPU to be used as a coprocessor.

Along with serial GZIP and ZLIB, we also used a parallel implementation of GZIP, called PIGZ, in our evaluation [19].

[1] For more information www.geforce.com/hardware/technology/sli.

6.3. Datasets

In our evaluation we used five datasets with different characteristics. The first is a collection of C source files. The second is maps of the state of Delaware. The third set is an English dictionary. The fourth is a tar file of the Linux 2.6.39 kernel. The fifth is a data dump from the Iperf network performance tool, used to test streaming.

6.4. Results and analysis

6.4.1. Algorithm improvements

The results for matching function improvement and texture memory usage are shown in Fig. 5. The improved matching function gave an 8.86% average increase in the throughput. With the use of the texture memory, the increase in the throughput becomes 15.67%. When we look at the specific buffer sizes and buffer counts, fewer large buffers lead to better throughput. The main reason for the better performance with larger buffers is that they give the GPU more data to consume and more opportunity to make use of the many-threaded architecture.

One interesting result is that the running time differences for different datasets in CULZSS [5] are eliminated by the use of state-based matching. Since the number of comparisons is fixed with our matching algorithm, the datasets with similar compression ratios give similar throughputs. The improvements show an average of 5.63× better performance compared to CULZSS with the same datasets and buffer configurations.

Fig. 6 shows the results of our search space and data structure choice to favor either the speed or the compression ratio. The figure consists both the FutureGrid Tesla experiments and our lab setup using GTX 480.

The upper left group of bars are single GTX 480 results, the upper right group of bars are double GPU projected results for GTX 480. The below group of bars show the FutureGrid Tesla results for single and double GPUs, respectively. The leftmost version uses a search buffer size of 512 bytes, which provides the highest compression. From left to right, the buffer size decreases to 128 bytes to favor faster compression. In each group, the rightmost three versions (128_2, 128_1, 128_0) use 128-byte buffers and two-byte search steps, as described in Section 4.2.

For the more compressible datasets (Map and Iperf), the GTX 480 results show 3507 Mbps average throughput on double GPU and 1753 Mbps on single GPU with a compression ratio (compressed/uncompressed) of 0.43. With the best compression ratio of 0.26, the throughput goes down to 505 and 252 Mbps for double and single GPU, respectively. The other datasets (C files, Dictionary and Linux tarball) result in a throughput of 1950 Mbps with 0.81 average compression ratio on double GPUs. With the best compression ratio of 0.51, the throughput goes down to 498 Mbps.

The Tesla C2075 gives a throughput of 2488 Mbps for double GPU and 1249 Mbps for single GPU on the fastest option over the more compressible datasets. The best compression ratio version gives 334 and 172 Mbps for double and single GPU, respectively, with the same ratio as GTX 480.

Fig. 7 shows the results for the computation overlap approach. For all the datasets, 16 overlapped version is the most beneficial configuration. After a pipeline depth of 32, run time for each chunk becomes too short to hide the overhead of multiple kernel invocations. After that point, overhead dominates which lead to decrease in performance.

Fig. 8 shows the percentage increase in the throughput and ratio of all the versions, fixing the buffer size 512 bytes version as the base case. It is clear that the percentage increase in the throughput is significantly faster than the increase in the ratio.

For both cards, the 128_2 version has reduced performance with the same compression ratio as the 128 buffer version. The

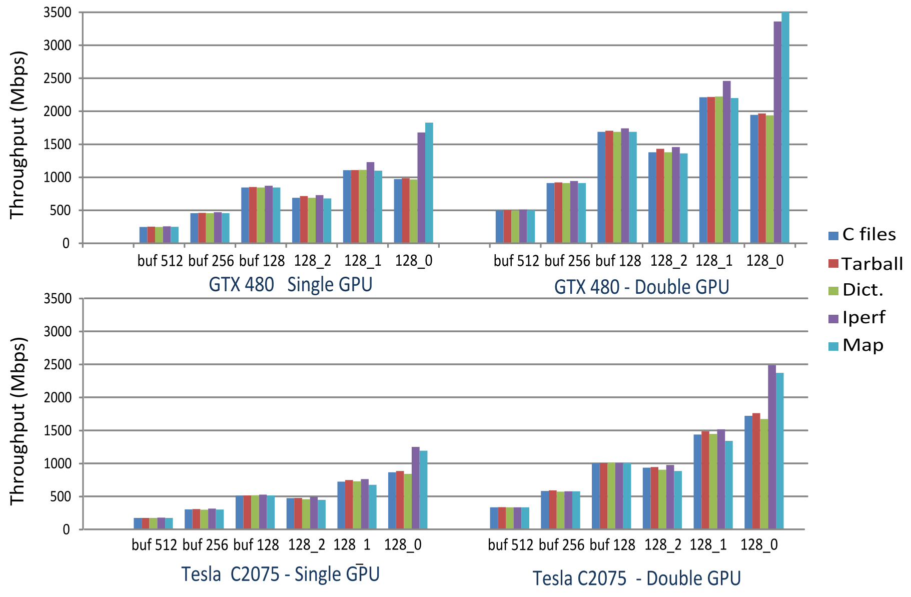

Fig. 6. Comparison of GTX 480 and Tesla C2075 benchmark results over five datasets and all six different CUDA versions.

reason is that the one-step increment two-byte substring fix, as described in Section 4.2, is done on both buffers in that version. That fix generates the same number of searches as the original 128 bytes buffer search version. The additional cost of generating two-byte substrings for each character leads to a loss in performance, while the number of searches stays the same.

6.4.2. Fine grained overlapping

In Section 3.2, we described the implicit synchronization rules and the overlapping behavior of CUDA framework under different command order scenarios. We tested GPU overlapping side of our code with two different order of commands. The pseudocode for each case is given at Algorithms 3 and 4. In Algorithm 3, which is one loop case, all the commands for a chunk are called before any of the next chunks' commands. CUDA streams are used to manage the concurrency between commands, assigned a unique id for each data chunk. In Algorithm 4, the nested loop case, the memory copies from CPU to GPU for all chunks are called before any kernel executions. The same way, all the kernel calls are made before executing any memory copy back from GPU.

Algorithm 3 Loop formation-One loop case

```
for each chunk $c_i$ do
    asyncMemcpy-CPU-to-GPU($c_i$)
    kernel($c_i$)
    asyncMemcpy-GPU-to-CPU($c_i$)
end for
```

The combination of the fine-grained overlapping on GPU kernel executions and pipelined approach for streaming data has taken the implementation into extreme overhead hiding and boost performance. The granularity of the overlapping individual GPU kernel executions and memory copies shown to be most beneficial at a level of 16 overlapping as experimented in Section 5.1. The results of the given combination for buf 128 version are shown at Fig. 9. The fine-grained overlapping is applied to buf 256 and buf 512 versions as well to improve the compression rate. On GTX 480 cards, the performance gain is by 6.78% on average for one loop case, and by 10.9% on average for processing the chunks by pieces as shown for nested loop case. The throughput values increased to 905 Mbps for first case and to 938 Mbps for latter case from the

Algorithm 4 Loop formation-Nested loop case

```
for each chunk $c_i$ do
    for each smaller chunk $c_{ij}$ in $c_i$ do
        asyncMemcpy-CPU-to-GPU($c_{ij}$)
    end for
    for each smaller chunk $c_{ij}$ in $c_i$ do
        kernel($c_{ij}$)
    end for
    for each smaller chunk $c_{ij}$ in $c_i$ do
        asyncMemcpy-GPU-to-CPU($c_{ij}$)
    end for
end for
```

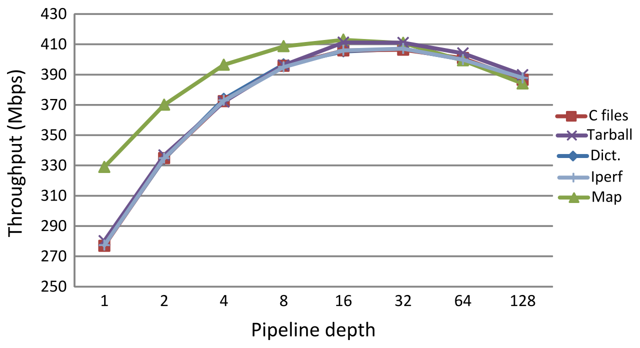

Fig. 7. Overlapped computation results.

previous synchronous version result of 850 Mbps. The Tesla C2075 has an increase of 42.9% on average on both cases. The throughput is increased to 750 Mbps from 516 Mbps.

The GTX 480 performance gain for the one loop case (Algorithm 3) comes solely from using the page-locked memory. It is because the GTX 480 series has only one concurrent copy engine which allows overlapping of the kernel execution and memory copy but does not allow concurrent data transfers between different CUDA streams. Each CUDA stream consists of three events: CPU-to-GPU memory copy, GPU kernel execution, and GPU-to-CPU memory copy back. In this case, the kernel execution of stream$_{i-1}$ can be overlapped with the memory copy from CPU to GPU issued to stream$_i$, since they are independent. However that memory copy is issued after the memory copy from GPU to CPU is issued

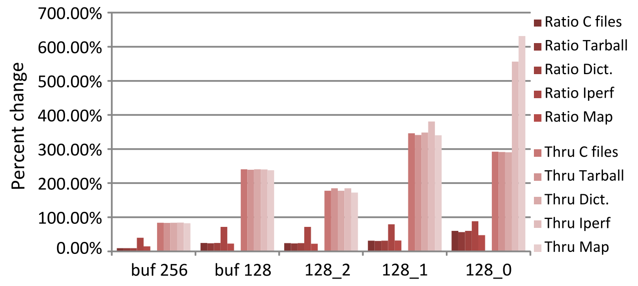

Fig. 8. Throughput and ratio change by different versions.

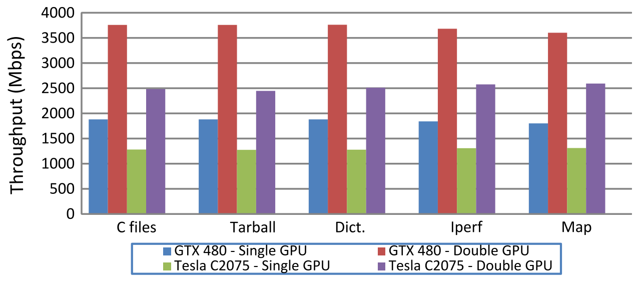

Fig. 10. Decompression results.

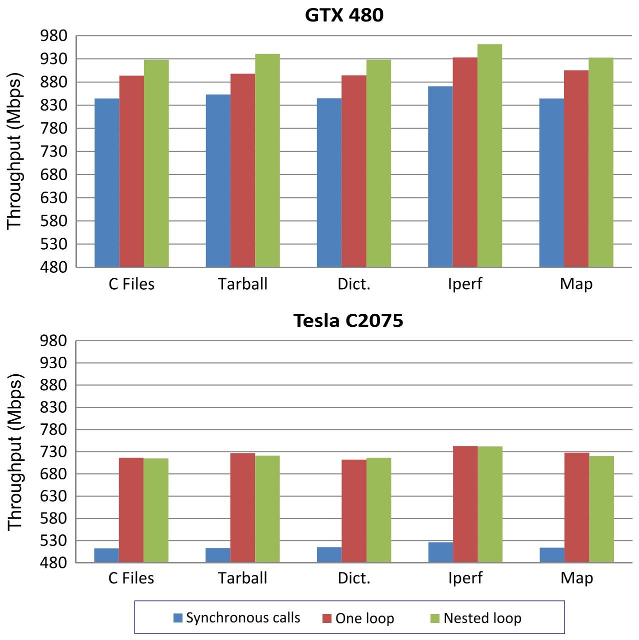

Fig. 9. Fine-grained asynchronous overlapping results.

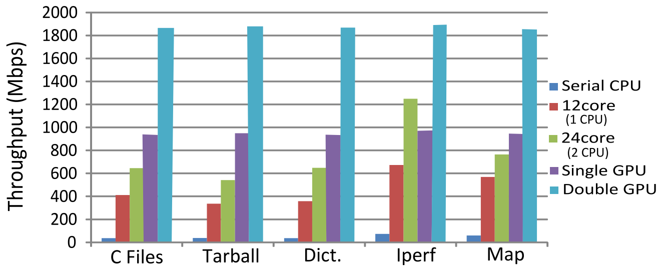

Fig. 11. Serial LZSS, parallel LZSS, and CUDA LZSS comparison.

to stream$_{i-1}$. So there is a memory copy in between that violates the implicit synchronization rules [12]. For that reason, it can only start once the memory copy from GPU to CPU issued to stream$_{i-1}$ has completed. This results in the code given in Algorithm 3 being totally serialized on the GPU side, although the commands are all asynchronous.

The revised code at Algorithm 4 starts all the CPU to GPU memory copy calls before kernel calls, so kernel calls for stream$_i$ can be overlapped with the memory copy from CPU to GPU for stream$_{i+1}$ since there is not a memory copy that violates the synchronization rules by being in the middle to any of the two CUDA streams. This gives an extra 4% boost to the GTX 480 performance results. Tesla C2075 cards, on the other side, has two concurrent copy engines which allows both concurrent data transfers and overlapping of the kernel execution and memory copy. Because of this capability the two cases in Algorithm 3 and 4 can both benefit from the overlapping. Therefore, for the Tesla C2075, both cases show a similar performance (Fig. 9). Having two concurrent copy and execution engines is also the main reason that Tesla C2075 benefits significantly more (45%) than the GTX 480 performance gain achieved by using the fine grained overlapping.

6.4.3. CUDA profiler

We mentioned the limiting factors of shared memory in the GPU architecture section. To verify that our program does not encounter any shared memory conflicts, we used the CUDA Visual profiler. Under memory throughput analysis tab, "l1_shared_bank_conflict" column reports the shared memory conflicts. The profiling of our program gives a zero value for that column, which shows there is no conflict. The profiler reports several metrics and is a good source for performance tuning feedback.

6.4.4. Decompression

The decompression results for each dataset are given in Fig. 10. The decompression algorithm is the same for all versions and adapted directly from the decompression algorithm by Ozsoy and Swany [5]. In a compress–send–decompress scenario, if the decompression rate is not lower than the compression rate, the throughput of the whole compression process will not be bounded by decompression step. The results show that our decompression algorithm runs at an average of 1880 Mbps for single GPU, and 3700 Mbps for double GPU on GTX 480. On the Tesla C2075, the single GPU throughput reaches an average of 1290 Mbps and 2522 Mbps for double GPU. These numbers are enough to sustain the compression rates given earlier. Additionally, the decompression results are very similar for different datasets, which shows that the running time of the decompression algorithm is not directly related to the data type or ratio.

6.4.5. Comparison with parallel CPU version

Fig. 11 shows the comparison between serial CPU, parallel CPU, and CUDA implementations. The serial CPU version of the LZSS algorithm runs at an average throughput of 49.09 Mbps over all datasets with a faster compression option (buffer 128 bytes) and 22.46 Mbps throughput with a better compression ratio (buffer 512 bytes).

The parallelized version achieves the best performance at 24 threads, which is equivalent to the number of underlying CPU cores (a single Xeon 5660 has 6 physical / 12 logical cores). The context switching overhead dominates the performance at higher thread counts. In the best case, parallel CPU LZSS can achieve an average throughput of 769.62 Mbps at faster compression and 351.92 Mbps at higher compression. Comparing the ratios between the CPU and CUDA implementations shows that CUDA versions face a 2%–5% loss. The reason for that is the chunk distribution among CUDA thread blocks requires each block to generate its own history buffer and filling the history buffer for each block causes compression loss. Comparing the throughput shows our double GPU CUDA implementation can go up to 1703.21 Mbps throughput at faster

Table 1
GZIP and ZLIB results.

Throughput	C files	Tarball	Dict	Iperf	Map
ZLIB	39.02	70.77	37.91	30.95	31.99
GZIP serial	227.15	222.46	221.02	209.62	173.79
GZIP(12)	1190.70	1146.70	1199.06	1228.85	1145.41
GZIP(24)	1296.20	1383.78	1376.34	1131.49	1349.14
CUDA-buf512	263.52	267.43	263.52	272.76	265.81
CUDA-128_0	972.70	984.33	969.17	1679.94	1827.84
CUDA2-512	511.66	529.50	516.98	537.32	510.47
CUDA2-128_0	1888.60	1948.98	1901.31	3309.48	3510.33
Ratio					
ZLIB	0.19	0.21	0.35	0.17	0.14
GZIP	0.19	0.21	0.19	0.17	0.15
CUDA-buf512	0.51	0.51	0.51	0.25	0.28
CUDA-128_0	0.79	0.82	0.82	0.46	0.41

compression and 521.19 Mbps at higher compression, achieving between 34× to 24× faster time than the serial implementation and 2.21× to 1.48× faster than the parallel version.

6.4.6. *Comparison with GZIP and ZLIB*

GZIP and ZLIB are well known compressors [20,21]. Both applications use the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding algorithms. The results are given at Table 1. The table also shows the results for the parallelized version of GZIP, called PIGZ, along with the double GPU CUDA versions (named with CUDA2 prefix).

The closest compression ratio for which we can compare our CUDA implementation to other two compression programs is for the 512-byte buffer version, in which our algorithm achieves the highest compression. Comparing that version to serial GZIP and ZLIB, our code gives 1.26× and 6.33× better throughput, respectively. However, the parallel version of GZIP, PIGZ, can reach an average throughput of 1307 Mbps on two CPUs (24 threads). In comparison, the double-GPU version with higher compression ratio of our algorithm only achieves 521.19 Mbps throughput on average, making it 2.5× slower than PIGZ. On the other hand, opting for faster compression in our algorithm can result in 2× better performance than PIGZan option that does not exist in PIGZ.

Our results show that ZLIB and GZIP can compress any dataset with a better ratio than the LZSS algorithm. This is because both of the algorithms use the LZ77 algorithm along with Huffman coding, which leads to a better compression ratio.

To give a chance to our CUDA code to improve the compression ratio by using statistical approaches like those being used in the ZLIB and GZIP, we also integrated Huffman coding in our CUDA software pipeline. The detail of the Huffman coding is out of scope for this paper. The detailed examination of the algorithm and a possible CUDA ported version are good candidates for future work. Briefly, however, Huffman coding is a statistical approach to the lossless data compression problem. The compression is done by assigning variable length codes to symbols and giving shorter codes to those that appear more frequently in the given data chunk, making those symbols require less space to store.

The integrated Huffman coding step comes at the end of processing, after GPU compression. The implementation is adapted from Dipperstein's work [22] and it runs only on the CPU. The overhead of this step can be hidden in the pipeline as long as it is not the slowest step. The results for a Huffman integrated 512-byte version are shown in Fig. 12. After applying Huffman coding, the compression ratio dropped by 11.16% on average. The Huffman coding is another step in the pipeline and because of the running time of the Huffman coding is not the slowest step in the pipeline, the throughput is not affected.

We extended the Huffman support for the 256-byte and the 128-byte versions as well. The 256-byte version shows on average

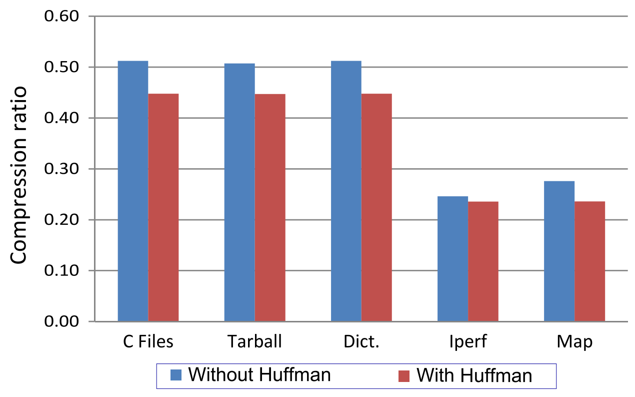

Fig. 12. Huffman coding extension results for 512-byte version.

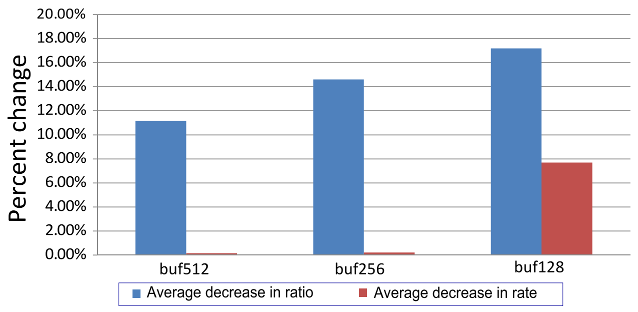

Fig. 13. Effect of Huffman coding on compression ratio and rate.

a 14.60% decrease in the compression ratio. The decrease in the ratio is the result of a larger LZ compressed file that is passed to the Huffman step compared to 512-byte version. Since the 256-byte version produces faster but less compressed data than 512 version, the Huffman coding get a better chance to compress on a bigger input. The throughput is still managed to be kept the same compared to non-Huffman version. The 128-byte version had a 17.20% decrease in the ratio; a bigger decrease compared to the 512 and 256-byte versions for the same reason we explained before — namely having a bigger chunk to compress with Huffman coding. However, the hidden cost of having Huffman coding in the streaming data pipeline stops in this version and the throughput decreases 7.7%. The reason of throughput decrease in 128-byte version is that Huffman coding cannot keep up with the GPU work. The GPU work which is the slowest step in the previous versions now is faster than the Huffman coding running time, and the throughput is determined by the Huffman coding which is the slowest step in this version. Because of the loss of performance in the compression rate for the faster versions, we stopped extending the Huffman coding for the further 128-byte versions which are faster. The three extended version comparisons are given at Fig. 13.

7. Conclusion and future work

We have presented a pipelined parallel LZSS compression algorithm for GPUs which is directed toward compressing streams of large data. Building on our earlier CULZSS algorithm, these GPU optimizations result in a 46.65% performance improvement. We then extended the algorithm by pipelining it, which showed encouraging performance in our experimental evaluation. This pipeline has six different levels, from fastest compression to high compression, which can meet varying requirements. Finally, compared to the serial and parallel CPU implementations of LZSS algorithm, our version performs better while maintaining a similar compression ratio.

Our experiments demonstrate that with the GTX 480 cards we can achieve better performance in all metrics compared with the Tesla C2075 cards because of the larger number of cores and higher core clocks of the GTX 480. One advantage that Tesla C2075 has over GTX 480 is its bigger memory capacity. However, our algorithm and benchmarks do not benefit from this additional availability of memory. The hardware prices are also notably different. At the time of writing, the Tesla C2075 is priced at $ 2099 and GTX 480 at $ 200–$ 250.[2] The discussion of the Tesla-specific features and comparison of different NVIDIA product lines are not within the scope of this paper; however, our benchmark results provide an interesting comparison of price-to-performance ratio of these two cards for lossless compression.

The Huffman coding extension gives a better ratio than pure LZSS. The adopted extension only involves the CPU usage and in faster and more data generated cases, the Huffman step can be the bottleneck among the other pipeline steps. To be able to provide full benefit on all different versions proposed in this paper, the parallelization of the code needs to be carefully visited on both the CPU and GPU sides. This is a future direction to investigate. Also, we invested less effort in studying decompression since it was not a limiting factor. However, with further improvements to compression stage, decompression might become the bottleneck and may need further study. Finally, we note that the matching stage of the algorithm is the most time-consuming part. For that reason, we believe that further improvements to that stage are likely to result in greatest overall benefit.

The algorithm and improvements to LZSS lossless compression on GPUs discussed in this paper are promising. Compared to serial CPU implementation, we achieved up to 34× better throughput and up to 2.21× better than the parallelized version. Our algorithm also compares favorably to the well known LZ-based compressors: GZIP and ZLIB. The results demonstrate that GPU-based LZSS lossless compression is effective and ready to leverage GPGPUs as compression coprocessors in streaming data scenarios.

Acknowledgments

This paper was developed with support from the National Science Foundation (NSF) under Grant No. 0910812 to Indiana University for ‘‘FutureGrid: An Experimental, HighPerformance Grid Testbed’’. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–343. http://dx.doi.org/10.1109/TIT.1977.1055714.
[2] J.A. Storer, T.G. Szymanski, Data compression via textual substitution, J. ACM 29 (1982) 928–951.
[3] M. Burrows, D. Wheeler, A block sorting lossless data compression algorithm, Tech. Rep. SRC-RR-124, Digital Equipment Corporation, 1994.
[4] D. Huffman, A method for the construction of minimum-redundancy codes, Proceedings of the IRE 40 (9) (1952) 1098–1101.
[5] A. Ozsoy, M. Swany, CULZSS: LZSS Lossless Data Compression on CUDA, in: Workshop on Parallel Programming on Accelerator Clusters, 2011, pp. 403–411.
[6] A. Balevic, Parallel variable-length encoding on GPGPUs, in: Proceedings of the 2009 International Conference on Parallel Processing, Euro-Par'09, 2010, pp. 26–35.
[7] W. Fang, B. He, Q. Luo, Database compression on graphics processors, Proc. VLDB Endow. 3 (1–2) (2010) 670–680.
[8] M.A. O'Neil, M. Burtscher, Floating-Point data compression at 75 Gb/s on a GPU, in: 4th Workshop on General Purpose Processing on GPUs, 2011.
[9] R.L. Cloud, M.L. Curry, H.L. Ward, A. Skjellum, P. Bangalore, Accelerating lossless data compression with GPUs, CoRR abs/1107.1525.
[10] R.A. Patel, Y. Zhang, J. Mak, A. Davidson, J.D. Owens, Parallel lossless data compression on the GPU, in: Proceedings of Innovative Parallel Computing, 2012.
[11] D.B. Kirk, W.-m.W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, first ed., Morgan Kaufmann Pub. Inc., San Francisco, CA, USA, 2010.
[12] NVIDIA, CUDA C Programming Guide, 2012.
[13] D. Kirk, NVIDIA CUDA software and GPU Parallel Computing Architecture, in: Proceedings of the 6th international symposium on Memory management, 2007, pp. 103–104.
[14] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, W.-m.W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008, pp. 73–82.
[15] S. Han, K. Jang, K. Park, S. Moon, PacketShader: a GPU-accelerated software router, SIGCOMM Comput. Commun. Rev. 40 (4) (2010) 195–206.
[16] FutureGrid, Delta User Manual, Retrieved June 14, 2012 from https://portal.futuregrid.org/manual/delta.
[17] G. Von Laszewski, G.C. Fox, A.J. Younge, A. Kulshrestha, G.G. Pike, W. Smith, J. Vockler, R.J. Figueiredo, J. Fortes, K. Keahey, Design of the futuregrid experiment management framework, in: 2010 Gateway Computing Environments Workshop GCE, 2010, pp. 1–10.
[18] LZSS (LZ77) Discussion and Implementation, Retrieved April 14, 2011 from http://michael.dipperstein.com/lzss/index.html.
[19] Pigz, Retrieved June 20, 2012 from http://zlib.net/pigz/.
[20] Gzip, Retrieved June 20, 2012 from http://www.gzip.org/.
[21] ZLIB, Retrieved June 20, 2012 from http://www.zlib.net/.
[22] Huffman Code Discussion and Implementation, Retrieved September 14, 2012 from http://michael.dipperstein.com/huffman/.

Adnan Ozsoy is a Ph.D. candidate at the School of Informatics and Computing of Indiana University — Bloomington. He received his B.Sc. in Computer Science from Virginia Polytechnic Institute and State University in 2005, and his M.Sc. in Computer Science degree from University of Texas at Austin in 2007. His research interests include parallel programming, high performance computing with GPUs, and application parallelism problems.

Martin Swany is an Associate Professor of Computer Science in the School of Informatics and Computing and Director of the Indiana Center for Network Translational Research and Education (InCNTRE) at Indiana University. He is a 2004 recipient of the US Department of Energy Early Career Principal Investigator award. His research interests include high-performance parallel and distributed computing and networking.

Arun Chauhan is an Assistant Professor of Computer Science in the School of Informatics and Computing at Indiana University. His research interests are in compilers, parallel computing, heterogeneous computing, and high-level programming languages. Prior to joining Indiana University, he was a Ph.D. student at Rice University, where he worked with late Prof Ken Kennedy on compiling MATLAB using an approach called ''telescoping languages''. Arun Chauhan holds a Masters in Computer Science & Engineering and a Bachelors in Electrical Engineering, both from the Indian Institute of Technology, New Delhi.

[2] All prices are from Amazon.com, June 2012.