A Scalable Multi-Engine Xpress9 Compressor with Asynchronous Data Transfer

Joo-Young Kim
Microsoft Research
Redmond, WA, USA
jooyoung@microsoft.com

Scott Hauck
University of Washington
Seattle, WA, USA
hauck@ee.washington.edu

Doug Burger
Microsoft Reserach
Redmond, WA, USA
dburger@microsoft.com

***Abstract*—Data compression is crucial in large-scale storage servers to save both storage and network bandwidth, but it suffers from high computational cost. In this work, we present a high throughput FPGA based compressor as a PCIe accelerator to achieve CPU resource saving and high power efficiency. The proposed compressor is differenciated from previous hardware compressors by the following features. 1) targeting Xpress9 algorithm, whose compression quality is comparable to the best Gzip implementation (level 9), 2) introducing scalable multi-engine architecture with various IP blocks to handle algorithmic complexity as well as to achieve high throughput, 3) supporting a heavily multi-threaded server environment with asynchronous data transfer interface between the host and the accelerator. The implemented Xpress9 compressor on Altera Stratix V GS performs 1.6-2.4Gbps throughput with 7 engines on various compression benchmarks, supporting up to 128 thread contexts.**

***Keywords*-FPGA; data compression; LZ77; Huffman encoding; hardware accelerator; Xpress; high throughput;**

I. Introduction

Modern server and storage systems handle peta-byte scale data with multiple compute and storage nodes connected to each other via high speed networks. Data compression plays an essential role in achieving a cost effective system by reducing the size of data to be stored or transmitted.

For this general purpose compression domain, multi-stage lossless algorithms that combine dictionary based method such as LZ77 [1] and statistical coding scheme such as Huffman encoding [2] are widely used. The best example is well-known Gzip compression [3]. Another example is LZMA [4], which claims the best compression ratio among others but is very slow due to its heavy optimizations.

In this paper, we present a high throughput Xpress9 compressor on reconfigurable devices. The Xpress9 algorithm is an advanced branch of Microsoft's Xpress compression algorithm family [5], targeting superior compression quality. We propose a multi-engine architecture that scales with the engine number, while each engine parallelizes Xpress9's algorithm features. We also provide an asynchronous data transfer interface to the host, which makes the proposed compressor useful under multi-threaded server environment.

II. Xpress9 Compression Algorithm

Algorithm 1 depicts the Xpress9 pseudo-code. The LZ77 process achieves compression by replacing a whole set of current data with a single reference to the repeated occurrence in the past, representing the result with a pair of numbers, *length-offset*. To find matches, it keeps the most recent data in a buffer called *window*, and slides forward by half the window length when it hits the end. The Xpress9 uses a 64KB window size rather than commonly used 32KB to increase the chance for matching. It performs LZ77 processing until it fills an internal buffer storing length-offset results, and streams them out packed in the bit-level by Huffman re-compression. The algorithm iterates these two stages to the end of input data.

Hash insertion, the first step of the LZ77 process, is a process of building a linked chain of matching candidate positions that have the same hash value. It can be effectively done with a head and prev table. The head table holds, for each set of 3 characters, the most recent position in the incoming data where that hash value has been calculated. When it encounters the same hash value, it retrieves the head position in that hash value (=head[hash value]) and store it to the current position of the prev table (=prev[pos]), which indicates the previous position that has the same hash value for a given position in the window. Then it updates the head table to the current position for the next insertion. Thus, to find all possible previous matches for a given position, we need to walk the prev table (e.g., prev[pos] gives the first candidate, prev[prev[pos]] gives the next previous, and so on) until the linked-list for that hash value is terminated by a NULL entry. The result of matching is represented with the following three packet types.

Algorithm 1 Xpress9 Algorithm

```
1: while (processed<input) do
2:     LZ77 process
3:     repeat
4:         Hash chain build;
               Prev[pos] ← Head[hashVal];
               Head[hashVal] ← pos;
           Xpress9 matching;
               LIT, MTF, PTR packet match
               frequency histogram;
5:     until fills internal buffer
       Huffman encoding
           Create Huffman code;
           Output bitstream;
6: end while
```

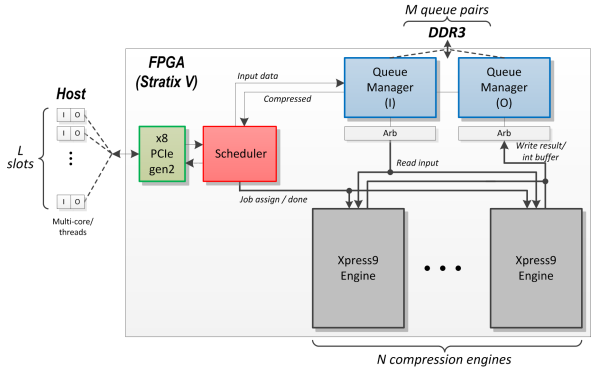

Figure 1. System architecture

- Literal (LIT): LIT emits current byte as it is since it could not find any good matches from the past.
- Pointer (PTR): PTR is a match between current and a previous point, containing the offset and length.
- Move-to-Front (MTF): As a special feature, Xpress9 stores the offsets of the 4 most recent matches, and searches those points prior to normal hash search.

MTF improves compression quality by stripping the offset field needed in PTR. Another feature of Xpress9 is local search optimization. It runs MTF and hash matching not only at the current position but also at the next two and picks the best overall result. This can be seen as an extended feature of Gzip's lazy evaluation that looks only the next position.

III. System Architecture

Figure 1 shows the system architecture of proposed FPGA based Xpress9 compressor. We used the Altera Stratix V development kit [6] that provides PCIe x8 interface to host PC and 1GB DDR3. The proposed architecture involves 3 key components to handle heavily multi-threaded compression workloads on the FPGA: a custom PCIe interface to support L communication channels, a queue management system to hold up to M different compression contexts, and a hardware scheduler to utilize N compression engines. The number L, M, and N can be selected for system requirements.

A. Host interface

Data transfer between the host and FPGA is accomplished by PCIe data channel called *slot*. Each slot includes an input and output pinned memory for sending/receiving data to/from the FPGA, respectively, with the memory size (=unit transfer size) of an integer power of two from 8KB to 128KB. Communication between multiple slots and the FPGA is done similarly as in circuit switching: it guarantees the channel until it ends the unit data transfer. We allocated 128 slots to support up to 128 threads. With 64KB transfer size, the implemented PCIe core gives 3GB/s bandwidth.

B. Queue management for asynchronous data transfer

Most of hardware accelerators require synchronous operation with their host CPUs. Since the host always waits for the FPGA to finish its processing, the synchronous operation can harm system throughput in practice. To prevent this, we introduced a queue management scheme that removes timing constraint between the host and the FPGA by buffering multiple compression contexts using DDR3 memory. The host can push/pull data to/from the FPGA regardless of its operation status, namely, asynchronously.

Due to the PCIe slot size, the host evenly chunks the input data into slot sized segments and send them to FPGA through a slot iteratively. Input queue manager is responsible for storing segments into a single queue so that they can get pulled out together later time. In general, it manages multiple queues in DDR3 and controls enqueueing and dequeueing operation of a requested queue. We employed another queue manager at the output side to enqueue compressed results from the compression engines and dequeue them for the host. We picked the queue number to 128, same as the slot number, to allow direct coupling between slots and queues.

C. Hardware Scheduler

The hardware scheduler manages a queue status table and an engine availability table to assign jobs to the engines. The former stores the compression context information for each queue such as input data size, compressed data size, and unique ID tags while the latter keeps 1 bit busy or idle status for each engine. For job assignment, the scheduler waits until it has an idle engine and sends the queue ID and input size information to the engine and make it busy. When the compression is done by an engine, the scheduler updates the compressed size at the queue status table and notifies the host through an interrupt. The engine becomes available again for the next compression. With this simple job scheduling, the scheduler ensures no engine remains idle when it has input data in the queueing system.

IV. Engine Implementation

Figure 2 shows the architecture of compression engine. It consists of several custom IP cores and memory modules, interconnected by a multi-layered bus. For the Huffman encoding part, we employed a NIOS II microprocessor that effectively runs the program with a 16KB instruction and 64KB data memory. To hide the Huffman encoding latency in the system, we perform task-level pipelining between LZ77 and Huffman stage. We utilize the external DDR3 for double buffering of 128KB intermediate results, with having a couple of 16KB buffers on-chip.

A. Hash Insertion

Hash insertion block implements the hash chain build stage with a simple 5 stage pipelined operation: data load, hash value calculation, head table read, prev table update, and head table update. To increase throughput, 2 consecutive bytes go through the pipeline at the same time with even-odd memory banking. With possible bank conflicts in the head table, the block achieves 350MB/s at 200MHz.

B. Multi-Path Speculation

To overcome FPGA's 10x slower clock rate than CPU, the multi-path speculation block parallelizes all the possible

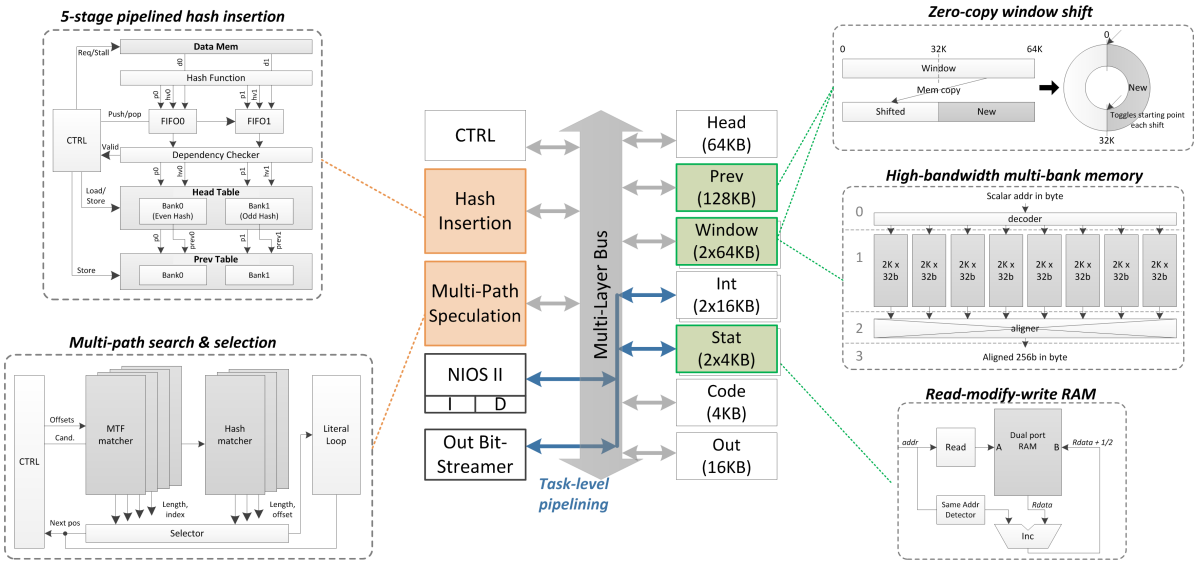

Figure 2. Engine architecture

matching paths in Xpress9 algorithm, i.e., 4 MTF matchings and a hash chain matching for 3 positions, and select the correct result when certain conditions are met. Once the output packet is selected, operations on false paths are terminated and get flushed for the next matching.

A byte matcher is a basic component in matching that computes the matched number of bytes between current and target position. It maximizes the throughput with 7 stage pipelined operation. For the first and second stage, it fetches 32 bytes of data from the window starting from current and target position, respectively, leveraging full bandwidth without conflicts. Third and fourth stage is waiting for data arrived from the window while it calculates the next two load addresses. At the fifth stage, the fetched two 32 byte streams are set to internal registers. For the last two stages, vector comparator compares two streams and calculates the number of identical bytes. With seamless pipelined operation, we can have 32 byte matching result every two cycles, which is equivalent to 3.2GB/s at 200MHz.

Since MTF search involves 4 different matchings per position, we need 12 byte matchings for 3 consecutive positions overall. However, if we re-partition them with an offset perspective, each offset performs 3 matchings, which are between current and offsetted, current+1 and offsetted+1, and current+2 and offsetted+2, respectively. Thus, only a single byte matcher is required to handle these matchings as the first two can share the result of the last and determine final results based on the first two byte comparison.

For hash chain matching, hash chain walker traverses the prev table from the current to the end of the chain and pushes read candidate positions into candidate queue. Tail checker filters candidates whose possible match length cannot exceed the current best length by comparing two offsetted tails of the current and candidate positions. Since the candidate queue naturally separates the hash chain walker's operation and byte matcher's operation, they can be run in parallel.

C. Zero-Copying Window Shift

To overcome the memory copy problem in window shifting, we made window and prev mem to be functioned as circular buffer. Regarding the physical half point as logical starting base, it fetches new data into physical former half without shifting. The logical starting point toggles every time the window shifts. Simple logical-physical address translation logic wrapped around the memory makes the address space appear to be same as before to outside blocks. Additional subtracting and threshold logic is added for prev mem to apply the offset caused by shift.

D. High-Bandwidth Multi-Bank Memory

To support wide bandwidth requirement of the window memory for parallel matchings by matchers, we employed two replicas of 8 32-bit wide dual port RAMs. In total, the window memory allows 4 simultaneous reads of 256-bit data every cycle, which provides 25.6GB/s at 200MHz.

E. Read-Modify-Write RAM

For frequency histogram of observed literals and matches, we devised a read-modified-write memory utilizing FPGA's dual-port RAM. In the histogram mode, a port reads the existing frequency from the address and writes the incremented value through the other port. A detection unit compensates the incremental value for the consecutive accesses to the same address within the reading latency.

V. Experimental Results

A. Resource utilization

Table I shows the resource utilization of our Xpress9 compressor on Stratix V GS FPGA that includes 172K Adaptive

Table I
STRATIX V GS RESOURCE UTILIZATION

Entity	ALMs	Memory bits
PCIe	4383 (2.5%)	2289632 (5.6%)
DDR3	9693 (5.6%)	313280 (0.8%)
2 x QM	22957 (13.3%)	1573036 (3.8%)
Scheduler	1630 (0.9%)	12160 (0.03%)
7 x Engine	117754 (68.2%)	26401536 (55.3%)
Misc.	1342 (0.8%)	398336 (1.0%)
Total	157759 (91.4%)	30987980 (75.8%)

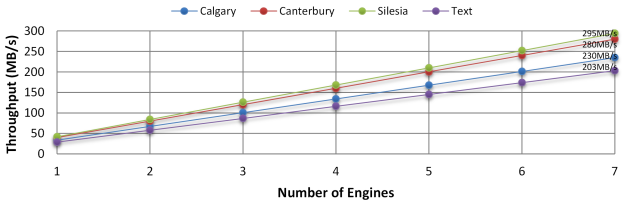

Figure 3. Multi-engine throughput performance

Logic Modules (ALMs) and 39Mbs of memory. System level IPs for host communication and multi-threading queueing accounts for 22% of logic and 10% of memory. For the rest, we successfully fit 7 compression engines with the hardware scheduler, with each engine consumes 9.7% of logic and 9.2% of memory. We have 3 clock domains: 250MHz for PCIe, 166MHz for DDR3, and 200MHz for user domain.

B. Experimental Setup

To evaluate the proposed compressor, we chose 4 different data benchmarks covering a variety of data types: Calgary and Canterbury Corpus [7], Silesia Corpus [8] and large text benchmark [9]. For comparison with other LZ77 based algorithms, we chose Gzip level1(fastest), level6(normal), level9(best compression), and LZMA. We used a machine with 2.3GHz Intel Xeon E5-2630 CPU and 32GB RAM.

C. Multi-Engine Scalability

We assume the highest workload scenario to measure the scalability of the proposed architecture. The host threads make 128 compression requests at the same time and we measured the overall spent time. As the graph in Figure 3 shows, the overall throughput scales linearly for all benchmarks as the number of engines increases. This can be achieved because asynchronous processing interface hides most of the data transfer time and the hardware scheduler seamlessly distributes the enqueued jobs to engines. This scalability will continue if the PCIe and DDR3 bandwidth can serve aggregated throughput of the engines.

D. Comparison

Figure 4 shows the compression ratio vs throughput graph for software algorithms as well as our hardware compressor. It is noteworthy that the throughput axis is in log scale. The LZMA keeps the best compression ratio for all the benchmarks but its throughput is limited to 1-2MB/s, proving improving compression quality is very expensive. For the GZIP family, the throughput quickly drops as the optimization level goes up to 9. However, there is no obvious

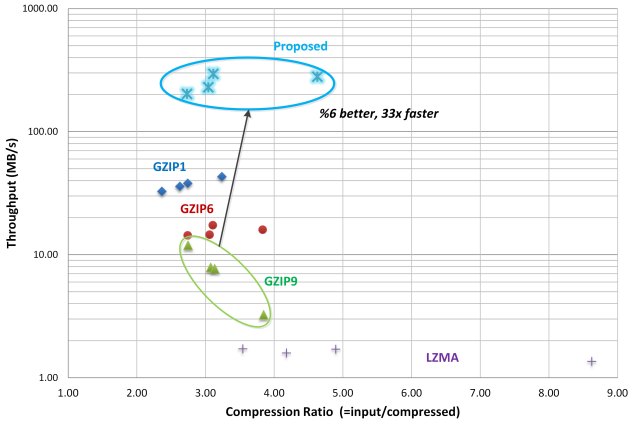

Figure 4. LZ77 based compressor comparison

gain in compression quality from level 6 to 9 although the throughput drops by half. On the other hand, our hardware Xpress9 compressor shows 16x and 33x performance boost from the Gzip level 6 and level 9, respectively, while maintaining 6% better compression ratio on average.

VI. CONCLUSION

In this paper, we presented a high quality and high throughput compressor on reconfigurable devices for storage server application. Unlike most hardware compressors target Gzip algorithm with limited set of features, we fully implemented the Xpress9 algorithm, claiming the best quality compression on the FPGA. With the multi-engine and queueing architecture, our compressor demonstrated scalable performance under heavily multi-threaded environment.

REFERENCES

[1] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

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

[3] J.-l. Gailly. (2013) Gzip the data compression program. [Online]. Available: http://www.gnu.org/software/gzip/manual/

[4] I. Pavlov. (2013) Lzma sdk. [Online]. Available: http://www.7-zip.org/sdk.html

[5] Microsoft. (2014) Ms-xca: Xpress compression algorithm. [Online]. Available: http://msdn.microsoft.com/en-us/library/hh554002.aspx

[6] Altera. (2014) Dsp development kit, stratix v edition. [Online]. Available: http://www.altera.com/products/devkits/altera/kit-stratix-v-dsp.html

[7] M. Powell. (2001) The canterbury corpus. [Online]. Available: http://corpus.canterbury.ac.nz/descriptions/

[8] S. Deorowicz. (2014) Silesia compression corpus. [Online]. Available: http://sun.aei.polsl.pl/ sdeor/index.php?page=silesia

[9] M. Mahoney. (2014) Large text compression benchmark. [Online]. Available: http://mattmahoney.net/dc/text.html