Documentation for GZip project

Last Update: 2015.9.3
Bisheng Huang, Tsinghua University
Xinyu Niu, Imperial College
Wayne Luk, Imperial College

1. Overview

Hardware implementation of lossless data compression is important for optimizing the capacity/cost/power of storage devices. GZIP is a file format used for file compression and decompression, which is based on one of the most popular algorithms for lossless storage - DEFLATE. DEFLATE uses a combination of the LZ77 algorithm and Huffman coding. A software version of LZ77 algorithm implementation can be found in [1].

In this work we use the MaxCompiler programming tool suite to implement LZ77 algorithm on a FPGA. At current stage we achieved a throughput of 8 input bytes per cycle clock, a compression ratio of 1.88, and a compression speed of 1.6GB/s on FPGA with a clock frequency of 200MHz. Note that the Huffman coding mentioned in [2] and [3] has not been included in current implementation.

2. Hardware Workflow

First, the host code sends the input data to LMEM via PCI. Then the compression core performs IO with the LMEM. At last the host code gets the outputs from LMEM via PCI again.

The data flow of the compression core can be outlined in Figure1.

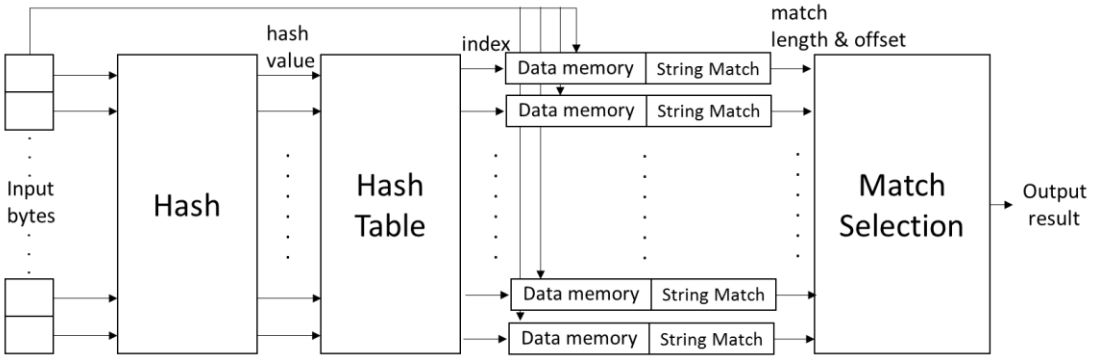

Figure 1. The workflow of the compression core

The compression core consists of four parts, the hashing module, the hash table module, the matching module, and the selection module. For every cycle, the core takes 8 bytes as inputs (called the input window). The function of each module will be discussed in details in the next part.

2.1 Hash module

The hash module performs hash for every three bytes. For an input window of 8 bytes (some bytes are buffered at the end), the module outputs 8 hash values in total. The hash function is the same function used by gzip and may be implemented using the following C fragment described in [1]:

```
key = 0;
for (i = 0; i < (M + 1); i++)
{
    key = (key << d) ^ (string[i]);
    key %= hash size;
}
```

For current implementation, d = 5 and hash size = 1024.

2.2 Hash Table module

The module receives the eight hash values from hash module, update the hash table and then output 8 indices for each hash value. (8*8=64 indices in total)

The hash values are stored in multiple hash tables. The eight input hash values are stored separately in eight hash table.

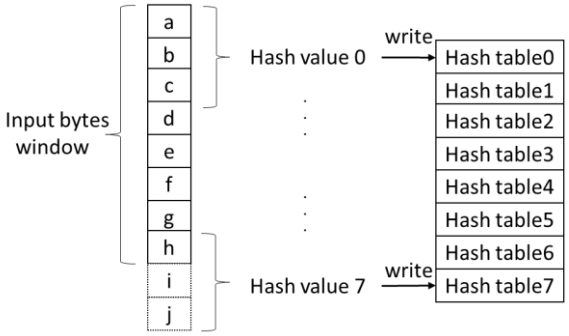

Figure 2. Hash table writing strategy

Then for each hash value, all the hash tables are read to get the indices for the sliding window.

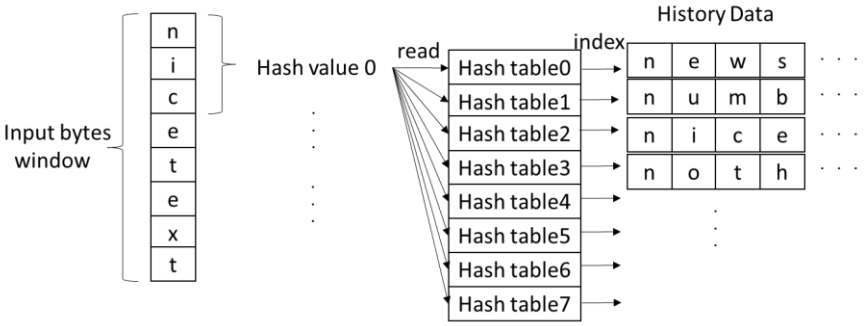

Figure 3. Hash table reading strategy

2.3 Matching module

The module has eight replications. Each is in charge of the eight indices associated with one hash value. For each of the eight indices, we takes out the history data according to the index from the history buffer. Then the history data is aligned using a crossbar. After that the history data is compared with the current input window to see how long the match can be. Since we have eight indices for each hash value, we will get eight match lengths and then pick the longest as output.

2.4 Selection module

The selection module gets a match length for every position of the input window. The module need to decide which match is selected because the matches may overlap with each other. The selection strategy we adopted is the Head-tail algorithm described in [1].

Currently the output is not the final result of the LZ77 algorithm but rather an intermediary. For every "match" in the input window, there are two outputs, the Valid_Bits, and the Data_Bits. The Valid_Bits is a number that indicates how many bits in Data_Bits are valid, while Data_Bits is a fix-sized 32-bits data. There are three possible values for the output Valid_Bits: 9, 16 and 0. 9 means outputting the original byte at this position. 16 means outputting an encoded symbol, and 0 indicates that this position has been covered by a previous match.

The outputs will be further organized in CPU host code. What the host code needs to do is just extract the valid bits in Data_Bits and put them together to generate a continuous data stream.

3. Performance

We tested the design on Calgary Corpus at a clock frequency of 200MHz, and we achieved a throughput of 1.6GB/s and a compression ratio of 1.88. Note that the performance represents all the compute elements described in Figure 1, excluding CPU-to-LMEM time, the File IO time, or the time used by the CPU to organize the results. The resource usage after optimization is 255764/262400 (97.47%) for logic utilization and 1315/2567 (51.23%) for block memory.

4. Limitations

1. The hash table architecture limits the number of indices that can be used.
2. The use of crossbar is expensive when considering logic resources usage. In the current design it's hard for us to expand input window from 8 bytes to 16 bytes or higher because the resources usage has already reached its upper limit.
3. The outputs of the design are not the standard outputs of Lz77 algorithms, and they need to be further organized in the CPU host code to generate the continuous output data stream.

5. Future Work

1. We can try to avoid using crossbars in the matching module by following the memory layout described in [3]. It stored the bytes-to-compare according to the indices. When searching the hash value in the HashTable, the history data that has the same hash value can be directly extracted and thus did not require a crossbar to align the data from the sliding window. In this way we will use more memory resources but save logic resources.
2. A Huffman coding module in the end can be added to increase the compression ratio. But it requires more logic resources too.
3. Since we still have memory resources left, we can increase the sliding window size and the hash table size to see if we can get improvement.
4. It will be better if the compression core can organize the outputs by itself and generates the final, continuous output stream without the help of CPU codes.

6. References

- [1]Michael Dipperstein; *LZSS (LZ77) Discussion and Implementation* http://michael.dipperstein.com/lzss/index.html
- [2]Fowers, Jeremy; Kim, Joo-Young; Burger, Doug; Hauck, Scott; *A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs*. IEEE International Symposium on Field-Programmable Custom Computing Machines, 2015.
- [3]Abdelfattah, M. S.; Hagiescu, A. & Singh; *Gzip on a chip: high performance lossless data compression on FPGAs using OpenCL*. Proceedings of the International Workshop on OpenCL, 2014.