LLNL-TR-811940

ZFP Hardware Implementation

G. S. Lloyd, P. G. Lindstrom

June 24, 2020

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

ZFP Hardware Implementation

Scott Lloyd, Peter Lindstrom

January 22, 2020

1 Motivation

As core counts increase in new HPC systems with comparatively little increase in memory bandwidth, the trend is an effective decrease in memory bandwidth per core. Other bandwidth limitations in HPC systems exist between CPU and GPU memory, between system nodes, and between node memory and storage. Compression of floating-point data has the potential to reduce data movement and pressure across these communication channels. Furthermore, it has the potential to reduce the footprint of floating-point arrays stored in memory. ZFP, implemented in software, is gaining traction as an effective method in floating-point compression; however, performance gains are limited to the spare compute cycles available before reaching the bandwidth limitations of the communication channel. A hardware implementation of ZFP has the potential to raise the bar on performance. From the inception of ZFP, it was designed to accommodate a hardware implementation [1].

2 Goals

Vendor supplied acceleration of floating-point compression could benefit many HPC applications. Ideally, this work will facilitate the introduction of ZFP hardware into future systems through fundamental research that shows the benefit to HPC applications. Researchers and application developers working together can help answer key questions for vendors seeking to justify the expense of new hardware.

3 Description of the approach

The hardware implementation of ZFP is sourced in SystemC to facilitate its evaluation in various architectures. Figure 1 shows the encode pipeline where uncompressed blocks of floating-point numbers in IEEE format are streamed into the unit and a compressed bitstream flows out from the unit. Figure 2 shows the decode pipeline which is the inverse of the encode pipeline. A modular design enables number formats other than IEEE (such as posits) to be considered with minor adaptations. Best performance will be realized from the hardware ZFP unit when batches of blocks are processed at a time. The current implementation supports 1-D, 2-D and 3-D blocks of floating-point numbers. C++ template parameters are used to specify the bit width of floating-point numbers and the array dimension of the encoder.

A test bench program has been created with several test cases, some with continuous data that is ideal for ZFP, and others with extreme cases containing numbers near or at the maximum or minimum values supported by the number format. The hardware implementation of ZFP has been validated with the software implementation of ZFP [2].

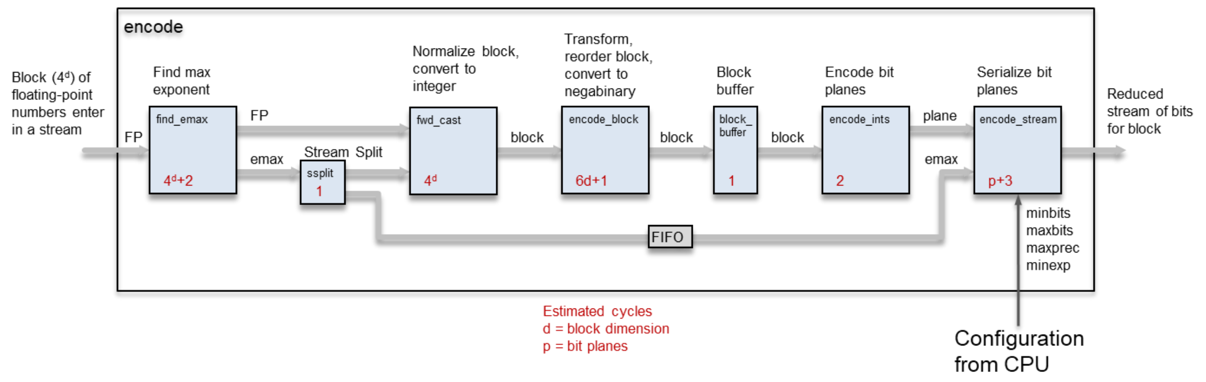


Figure 1: ZFP encode pipeline

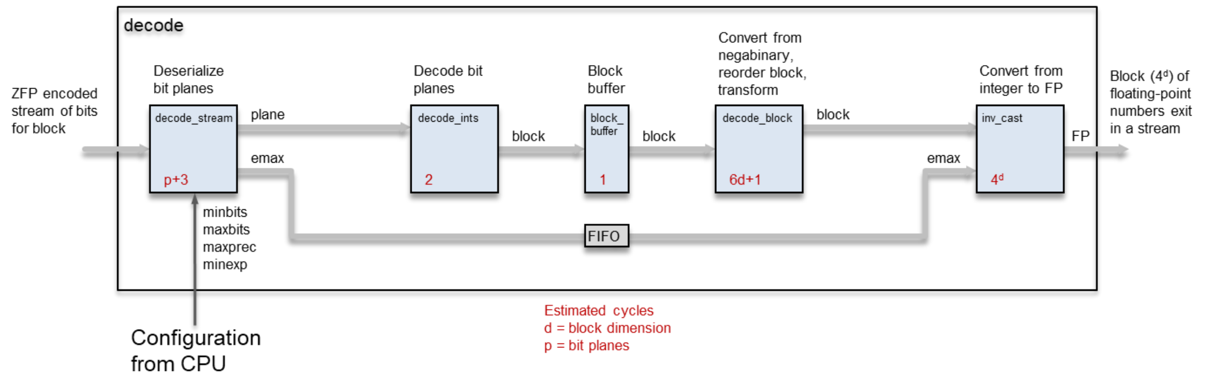


Figure 2: ZFP decode pipeline

The encode and decode pipelines consists of several stages connected by streams of data. A ready-valid handshake is employed in each stream which can apply back pressure to upstream modules if a module is not ready for data. Downstream modules will also wait until data is available from an upstream module. Flits are the unit of data passed between modules and can move between modules at a rate of one per clock cycle if no back pressure is applied. The width of a flit in bits may be different between modules depending on the type of data. For example, between the find_emax module and the fwd_cast module, the flit width is a single floating-point number. Between fwd_cast and encode_block the flit width is 16 floating-point numbers for the 2-D case. An alternate implementation of find_emax could output 4 floating-point numbers in a flit, but this would not improve performance unless the input to find_emax was also a vector of floating-point numbers.

Sub-modules can be categorized into three major groups: 1) FP Input/Output, 2) Transform, and 3) Serialize/Deserialize. Each of these groups function in parallel with a small amount of latency between the groups.

The first stage, find_emax, finds the maximum exponent in a block of numbers (16 for the 2-D case) and then forwards each number on to the second stage, fwd_cast, which normalizes the numbers relative to the maximum exponent. This normalizing stage cannot start until the maximum is found in the first stage. Find_emax contains a FIFO large enough to buffer a block of floating-point numbers until the maximum is found.

Transformation on a block occurs through a pipeline of substages, one for each dimension. As soon as a substage finishes processing one dimension, the next substage can begin processing the next dimension. Each substage consists of one or more transformation modules (fwd_lift) that operate on 4 numbers. In the 2-D case, a substage consists of 4 transformation modules operating in parallel to process the 16 numbers in the block. As shown in Table 1, transformation module processing completes in 6 cycles and occurs in parallel on each variable. At most one arithmetic operation and a shift is executed for each variable in a clock cycle, which allows for a higher speed design. After the transformation, the numbers are reordered and converted to negabinary in one clock cycle.

Table 1: fwd_lift transform module operation

Cycle	X	Y	Z	W
1	x1 = (x0 + w0) >> 1;	y1 = y0;	z1 = (z0 + y0) >> 1;	w1 = w0;
2	x2 = (x1 + z1) >> 1;	y2 = y1 - z1;	z2 = z1;	w2 = w1 - x1;
3	x3 = x2;	y3 = y2;	z3 = z2 - x2;	w3 = (w2 + y2) >> 1;
4	x4 = x3;	y4 = y3 - w3;	z4 = z3;	w4 = w3;
5	x5 = x4;	y5 = y4;	z5 = z4;	w5 = w4 + (y4 >> 1);
6	x6 = x5;	y6 = y5 - (w5 >> 1);	z6 = z5;	w6 = w5;

An alternate transform is shown in Table 2 that only requires 4 cycles to compute; however, the results may differ from the software version of ZFP in the last bit.

Table 2: alternate transform in 4 cycles (not identical with software ZFP)

Cycle	X	Y	Z	W
1	x1 = (x0 + w0) >> 1;	y1 = (y0 - z0) >> 1;	z1 = (z0 + y0) >> 1;	w1 = (w0 - x0) >> 1;
2	x2 = (x1 + z1) >> 1;	y2 = (y1 - w1) >> 1;	z2 = (z1 - x1) >> 1;	w2 = (w1 + y1) >> 1;
3	x3 = x2;	y3 = y2;	z3 = z2;	w3 = w2 + (y2 >> 1);
4	x4 = x3;	y4 = y3 - (w3 >> 1);	z4 = z3;	w4 = w3;

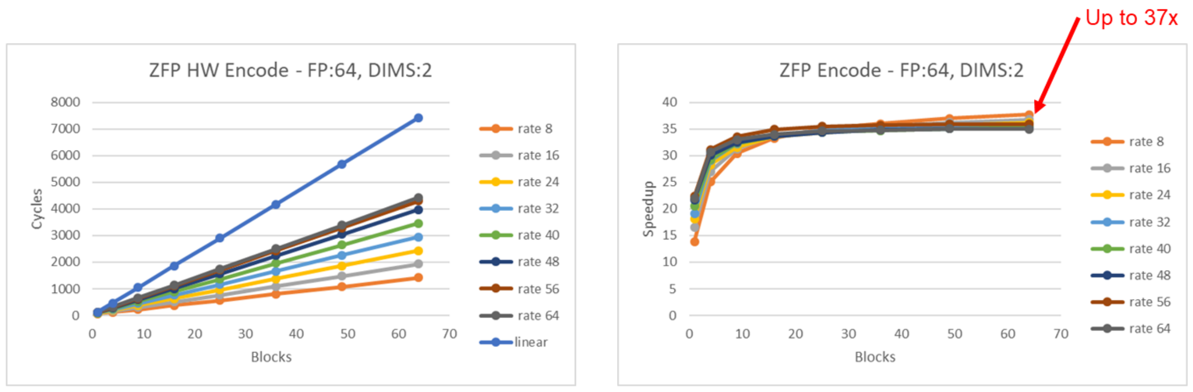
The last two modules encode and serialize the result of the transform. A buffer stage resides between the transformation and encode stages to improve the throughput. The encode stage cannot start until a complete block is ready for processing since it remaps the floating-point numbers into bit planes. With the buffer, transformation can continue while the encode stage is processing, otherwise the transformation stage would block waiting for the encode stage to finish. The final stage encodes the exponent in the output stream and serializes bit planes received from the encoding stage. This final stage also has access to configuration variables from a managing process such as a CPU that dictates the compression rate and other parameters for the compressed output.

4 Important Results

Demonstration of the ZFP hardware implementation under simulation shows that:

- ZFP IP can target an FPGA or be added to a new chip design
- Latency for compressing one block ranges between 34-84 cycles in the 1-D case, 64-118 cycles in 2-D, and 172-247 cycles in 3-D
- Speedup of the hardware implementation over the software ranges from about 15x for 1-D arrays to over 200x for 3-D arrays

Using a continuous-data test case with a sweep of parameters varying the dimension, the number of blocks encoded, and the compression rate; cycle times were recorded for both the hardware and software implementations. For the hardware cycles, the integrated SystemC event simulator was used. For software cycles, the x86 cycle counter was accessed with inline assembly instructions using the public release of ZFP from GitHub [2]. The "linear" data set in Figure 3 extrapolates the cycle time of encoding one block (rate 64) to many blocks, all encoded one at a time. It shows the benefit of encoding batches of blocks through the hardware pipeline. Batch timing starts when the first floating-point number enters the unit and stops when the last word (flit) of compressed data leaves the unit. The benefit of encoding blocks in a batch can be more than a factor of 2 in the 2-D case. Speedup of the hardware implementation over the software ranges from about 15x for 1-D arrays to over 200x for 3-D arrays.



"linear" extrapolates the cycle time for one block (rate 64) to many blocks. It shows the benefit of encoding batches of blocks through the hardware pipeline.

Compared with single x86_64 core, Intel i7, Linux

Figure 3: ZFP performance for 64-bit, 2-D blocks

5 Research artifacts

The artifacts in the ZHW collection [3] include:

- SystemC source code for a hardware implementation of ZFP and a test bench
- A presentation summarizing the ZFP implementation and results
- Simulation results from the test bench

6 Next steps

Only the encoding or compression half of ZFP is implemented in SystemC. The other half, decompression, can be realized as the inverse of compression. We intend to evaluate ZFP using LiME [4], our FPGA emulation platform, by translating SystemC into Verilog or VHDL and integrating it with our existing design.

The current implementation of ZFP in hardware is sufficient to begin performance studies for applications of interest. Furthermore, it can support work as a component in a larger framework to evaluate IP blocks at different locations within an HPC system.

7 Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LDRD 17-SI-004, LDRD 19-ERD-004, LLNL-TR-811940.

References

[1] Peter Lindstrom. *Fixed-Rate Compressed Floating-Point Arrays*. IEEE Transactions on Visualization and Computer Graphics 20(12):2674-2683, December 2014. doi:10.1109/TVCG.2014.2346458

[2] ZFP on GitHub, http://github.com/LLNL/zfp

[3] ZHW on GitHub, http://github.com/LLNL/zhw

[4] Abhishek Jain, Scott Lloyd, and Maya Gokhale. *Microscope on Memory: MPSoC-Enabled Computer Memory System Assessments*. FCCM, Boulder, CO, 2018, pp. 173-180. doi:10.1109/FCCM.2018.00035