BurstZ: A Bandwidth-Efficient Scientific Computing Accelerator Platform for Large-Scale Data

Gongjin Sun
Department of Computer Science
University of California, Irvine
gongjins@uci.edu

Seongyoung Kang
Department of Computer Science
Kookmin University
ksy9164@kookmin.ac.kr

Sang-Woo Jun
Department of Computer Science
University of California, Irvine
swjun@ics.uci.edu

ABSTRACT

We present BurstZ, a bandwidth-efficient accelerator platform for scientific computing. While accelerators such as GPUs and FPGAs provide enormous computing capabilities, their effectiveness quickly deteriorates once the working set becomes larger than the on-board memory capacity, causing the performance to become bottlenecked either by the communication bandwidth between the host and the accelerator. Compression has not been very useful in solving this issue due to the difficulty of efficiently compressing floating point numbers, which scientific data often consists of. Most compression algorithms are either ineffective with floating point numbers, or has a high performance overhead.

BurstZ is an FPGA-based accelerator platform which addresses the bandwidth issue via a novel hardware-optimized floating point compression algorithm, which we call sZFP. We demonstrate that BurstZ can completely remove the communication bottleneck for accelerators, using a 3D stencil-code accelerator implemented on a prototype BurstZ implementation. Evaluated against hand-optimized implementations of stencil code accelerators of the same architecture, our BurstZ prototype outperformed an accelerator without compression by almost 4×, and even an accelerator with enough memory for the entire dataset by over 2×. BurstZ improved communication efficiency so much, our prototype was even able to outperform the upper limit projected performance of an optimized stencil core with ideal memory access characteristics, by over 2×.

CCS CONCEPTS

• **Computer systems organization → Architectures**; • **Hardware → Hardware accelerators**; • **Theory of computation →** *Data compression.*

KEYWORDS

FPGA accelerators, Bandwidth, Compression, Stencil

ACM Reference Format:
Gongjin Sun, Seongyoung Kang, and Sang-Woo Jun. 2020. BurstZ: A Bandwidth-Efficient Scientific Computing Accelerator Platform for Large-Scale Data. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392746

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICS '20, June 29-July 2, 2020, Barcelona, Spain
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7983-0/20/06.
https://doi.org/10.1145/3392717.3392746

1 INTRODUCTION

While modern heterogeneous computing systems equipped with application-specific hardware accelerators can achieve high performance and power efficiency, their performance is often limited by available communication bandwidth. For ease of deployment, accelerators such as General-Purpose Graphics Processing Units (GPGPU), Field-Programmable Gate Arrays (FPGA), Tensor Processing Units (TPU), and others, are often packaged as a PCIe-attached expansion card. Such accelerators deliver extremely high performance if the working set fits on their on-board memory resources, but once the working set exceeds their memory resources so that data needs to be dynamically transferred over PCIe, the limited bandwidth of the PCIe link often becomes the critical performance bottleneck [1, 3, 9, 19, 43, 44]. Due to this reason, many existing research on scientific computing accelerators have focused on problem sizes which can fit on the on-board memory resources [6, 8, 17, 42, 46, 51].

Compression is a traditional solution to the interconnect bandwidth issue, but its use has been limited for scientific computing acceleration because of the high performance overhead of floating-point compression algorithms. General-purpose lossless compression schemes such as DEFLATE [14] and LZW [48] are typically very inefficient with floating poing data, which often make up a large part of scientific datasets [15, 30]. Floating-point specific lossy compression algorithms such as ZFP [16, 30] and SZ [15] are widely used to compress scientific data, due to their very efficient compression as well as their capability to limit the error bound of each data element. However, such complex algorithms also have high performance overhead compared to LZ4 or LZO, making their demonstrated performance insufficient to keep up with the internal computation capabilities of scientific computing accelerators.

1.1 BurstZ Platform

This paper presents BurstZ, which addresses the communication bandwidth issue of scientific computing accelerators, by providing the computation engine with a platform which communicates with the host over an efficient hardware implementation of a novel hardware-optimized floating point compression algorithm. BurstZ supports large-scale data processing by storing data in either the memory or storage of the host server in a compressed, randomly accessible format, and decompressing it piecemeal within the accelerator side when required.

Our novel compression algorithm, a variant of the ZFP algorithm we call sZFP, is capable of providing wire-speed compression and decompression of floating point data while using only a small fraction of on-chip resources. sZFP modifies ZFP and introduces a new embedded coding scheme which allows very efficient hardware

implementation. The new coding scheme trades a small amount of compression ratio for an order of magnitude performance improvement. In fact, our sZFP implementation is efficient enough to remove not only the PCIe performance bottleneck, but also alleviate the on-board DRAM performance bottleneck by storing compressed data even in on-board DRAM and decompressing them on the fly.

1.2 Example application: 3D Stencil

As a motivating example, we use 3D stencil computation as an example computation engine to illustrate BurstZ's feasibility and efficiency. Stencil computation is a popular method of scientific computing commonly used in many areas including climate and seismic simulation, as well as approximate solutions of partial differential equations. Stencil computation operates on a multidimensional array by updating each cell according to some fixed pattern, called a stencil, which takes as input the value of cells in a small number of immediately surrounding cells. This makes the computation pattern very regular, and theoretically easily parallelizable.

Stencil computing acceleration has been already researched extensively on various technologies including FPGA, GPU and CPU, and have produced efficient implementation techniques including architectural optimizations, performance modeling, and cache-optimization techniques [8, 10, 11, 33, 34, 36, 46, 47]. However, many previous work on stencil accelerators tend to focus on highly optimizing the stencil computation unit implementation, and do not directly address the bandwidth issue between the accelerator and the host.

Algorithms such as temporal blocking help circumvent the communications bandwidth issues by improving the data movement to computation ratio. However, these solutions **still suffer linear performance degradation as problem size becomes larger** [18, 43]. Furthermore, they are orthogonal solutions to directly removing the communications bottleneck such as what BurstZ aims to do. All ideas related to caching and temporal blocking can also be applied to the BurstZ platform to achieve synergistic results.

1.3 Prototype Implementation and Evaluation

We have implemented BurstZ on a Xilinx VC707 FPGA development board, with a PCIe Gen2 x8 link to host with a maximum bandwidth of 4 GB/s duplex. The accelerator card includes a Xilinx Virtex 7 FPGA chip, as well as an on-board DDR3 DRAM card capable of measured performance of over 11 GB/s. On the BurstZ prototype, we have implemented a 7-point 3D heat-transfer stencil as a motivating example.

In this environment, our BurstZ platform was able to deliver almost 32 GB/s of effective, steady-state bandwidth to our stencil core while streaming large-scale data from the host over PCIe. Such a bandwidth is sufficient to supporting the peak computation capability of our stencil core. This is almost 4× the performance compared to the same hardware platform without BurstZ, where the performance is restricted by PCIe communication and memory access overhead. Even compared to a platform with enough on-board DRAM to hold all required data, our prototype still achieves over 2× the performance. This is especially impressive because a system with enough on-board DRAM requires no communication over slow PCIe. BurstZ is able to achieve higher performance by improving even the effective bandwidth of the on-board DRAM module via wire-speed compression.

The stencil core implementations internally can use various optimizations such as temporal blocking to achieve higher performance within the memory bandwidth budget. However, as long as the performance is being limited by communication bandwidth such optimizations are equally beneficial to all above configurations. As a result, the performance relations between them will show similar patterns regardless of applied optimizations.

1.4 Contributions

We claim the following contributions of this work.

- A bandwidth-efficient scientific computing accelerator architecture that removes the PCIe bandwidth bottleneck using hardware-accelerated compression.
- A modified ZFP algorithm for high-performance error-bound lossy compression in hardware.
- Performance analysis of our architecture demonstrating that the solution is scalable in relation to PCIe and on-board DRAM bandwidth, as well as FPGA capacity.

The rest of this paper is organized as follows: Section 2 presents some existing work relevant to this one. Section 3 presents a detailed description of stencil computation and factors which affect its performance. Section 4 describes the ZFP algorithm and its performance bottleneck, and then presents the design of our sZFP algorithm. Section 5 explains the architecture of BurstZ. Performance and efficiency evaluations are presented in Section 6, and we conclude with discussion in Section 7.

2 BACKGROUND AND RELATED WORK

2.1 Stencil Computing and its Acceleration

Stencil computing is an iterative computing method, which operates on a multidimensional grid representation of data. Computation is expressed in terms of *stencils*, which update a cell in the grid based on the values of a small number of cells in the immediately surrounding area. Figure 1 shows a graphical representation of a 2-dimensional 5-point stencil and a 3-dimensional 7-point stencil. A wide variety of stencils have been designed depending on the application, such as 9-point 2D stencils, and 25-point 3D stencils. At each time step, the stencil code *sweeps* across the entire grid, updating each grid value. There is no dependency between each stencil operation within a single sweep, a characteristic which allows straightforward parallelization.

Stencil codes are important tools in many scientific computing problems. One important example of stencil computation, the Lattice Boltzmann Method (LBM), is a popular method to solve industrial-scale fluid dynamics and heat transfer problems without having to solve computationally-intensive partial differential equations [13, 23, 28, 35, 40].

Due to their importance in many scientific applications, there have been a great amount of previous work on its optimization and acceleration on various computation platforms such as multi-core CPUs, GPUs and FPGAs. Both FPGA and GPU-based accelerators have demonstrated very high performance, but here we focus

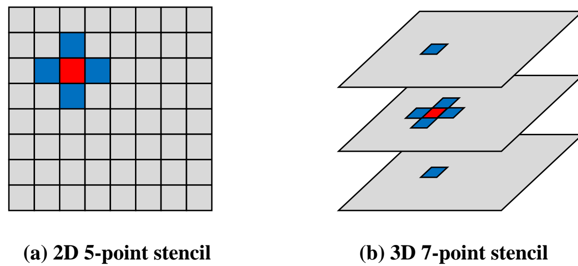

Figure 1: Example 2D and 3D stencils

on FPGA-based acceleration as they often demonstrate very high power efficicy [26, 38, 52].

Thanks to the simple nature of individual stencil code and ease of parallelization, the performance of stencil code accelerators are typically not bound by their computational capacity, but by the speed in which grid data can be accessed [11, 23, 35]. As a result, a large amount of work has focused on memory access and re-use methodologies, aiming to improve the ratio between the amount of memory access and computation.

Improving Memory Re-Use: Two major methods of improving memory re-use is (1) *tiling*, which improves spatial re-use, and (2) *temporal blocking*, which improves temporal re-use. Tiling loads and processes data in units of multi-dimensional tiles which can fit in on-chip memory, allowing most cells in a tile to be loaded once, except for a relatively small number of cells located at the edge of each tile, which requires data from neighboring tiles to compute. These cells are called the *halo*. Tiling in the stencil context is analogous to tiling for cache efficiency in matrix multiplication [2, 5, 41]. Temporal blocking performs multiple sweeps of computation on a tile while they are loaded on on-chip memory, before the results are written back to large main memory. One caveat of temporal blocking is that the size of the halo becomes larger with more sweeps, as illustrated in Figure 2. This is because with each sweep, more cells near the edge of each tile depend on the updated data of the original halo in the first sweep. This limits the use of temporal blocking, especially with high dimensions or high-order stencils which depend on a relatively large number of neighbor cells.

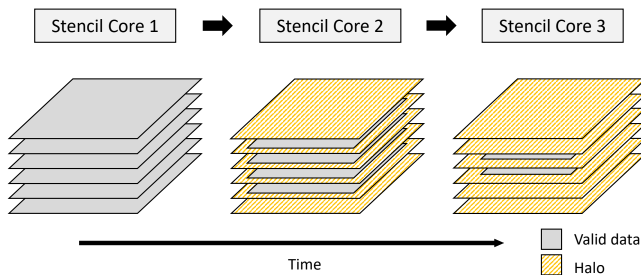

Figure 2: Deep temporal blocking increases the size of the *Halo* , reducing the amount of valid data

Most modern stencil accelerator designs take advantage of both tiling and temporal blocking, and more [6, 17, 19, 42, 46, 51]. A large body of work has focused on determining an optimal tiling and temporal blocking methods given the accelerator platform [8, 11, 42], as well as devising performance models and characterization methods about various memory optimizations [12, 17]. There has been research into efficient generation of stencil accelerator on FPGAs using high-level languages such as OpenCL [46, 51, 52].

Communication Bottleneck: Most of existing research on stencil accelerators have focused on problem sizes which can fit in the fast on-board memory capacity available on the accelerator device. Once the problem size becomes too large, data access starts spilling over into host-side memory or storage over a relatively slow interconnect such as PCIe, which immediately becomes the bottleneck of performance.

While the same tiling and temporal blocking optimizations be applied at the scale of the on-board memory to make the problem less bandwidth-bound, the same problem still exists as the problem sizes become larger. This is because issues including the aforementioned halo growth limits the effectiveness of temporal blocking. As a result, it has been shown that even temporally blocked kernels **suffer linear performance degradation** as the problem size becomes much larger than on-board memory capacity [18, 43]. This is the situation we are interested in.

In this work, we mainly focus on the issue of removing the communication bottleneck, as it impacts both temporally blocked and non-blocked implementations. To the best of our knowledge, BurstZ is the first system which completely removes the host-side interconnect bottleneck using fast compression.

2.2 Error-Bounded Lossy Compression of Scientific Data

A traditionally effective method for reducing the overhead of data movement is compression. Lossless compression methods including DEFLATE [14] and LZW [48] have been very effective in compressing enterprise data. High-throughput compression algorithms such as LZO [37], LZ4 [7] and Stream VByte [29] sacrifice a small amount of compression efficiency for speed, and has been been useful in many high-performance processing environments, in applications including compressing network traffic [20, 49] and operating system swap space compression [27] for distributed processing. Stream VByte, for example demonstrated over 16 GB/s decompression throughput on a 3.4 GHz Haswell processor.

However, such data-oblivious lossless algorithms cannot efficiently compress scientific data, which often consists largely of floating point numbers [15, 30]. Floating point encoding can incur a large entropy (i.e., irregularity), which general-purpose pattern-matching compression methods struggle with. Tested on real-world data, effective lossless compression schemes such as gzip struggle to achieve even 2-to-1 compression [30].

An effective class of compression algorithms for floating point values is lossy compression algorithms such as ZFP [16, 30] and SZ [15]. If the domain expert knows that the data and application can tolerate a certain amount of precision loss, such lossy algorithms can achieve extremely high compression efficiency while ensuring the user-defined error bound on each value. This error bound guarantee makes lossy compression much more desirable compared to simple quantization of values to 32-bit or 16-bit floating point values, may have accuracy losses oblivious to the actual

scale of the individual data elements, leading to large, unexpected errors. Under realistic levels of error tolerance for HPC scientific data, these lossy algorithms regularly achieve compression ratios of over 10x [32]. As a result, such algorithms have been used in a wide array of applications including medical image reconstruction [21], extreme weather simulation [39], extreme-scale scientific frameworks [22], and many more [31].

Performance Overhead: While lossy compression algorithms can achieve very efficient compression, they are not immediately applicable to the task of removing the link bandwidth bottleneck, due to their performance overhead. Evaluation of single-thread ZFP and SZ implementations on the Argonne FUSION cluster server with 2.6 GHz Xeon Nahelem processor measured less than 300 MB/s for compression and decompression for both algorithms. Running enough threads to saturate the PCIe link with such algorithms would be too computationally expensive.

There exist GPU-accelerated implementations of both ZFP and SZ, which create massively many instances of the algorithms to parallelize computation [25]. These implementations use the thousands of computation units available on modern GPUs to achieve dozens of GB/s of compression and decompression, and are capable of saturating the PCIe bandwidth between the GPU and host.

FPGA Implementations: However, this same approach is not very efficient on FPGAs, which have much lower clock frequencies compared to GPUs. Furthermore, the FPGA chip space limitations prevent fitting too many instances of compression/decompression cores on chip. For example, a single pipeline of GhostSZ [50], an FPGA implementation of SZ, significantly outperformed software with over 800 MB/s of compression throughput on an Intel Arria 10 FPGA. While 8 pipelines of GhostSZ is projected to deliver over 6 GB/s of bandwidth, this would already consume over 40 % of the chip. This is still not enough to saturate the PCIe link in compressed form. In a previous work, we have explored optimized implementations of the ZFP algorithm on an Arria 10 FPGA [45], and arrived at similar results: Achieving 1–2 GB/s of bandwidth while consuming 30% of chip space. In the same work, we introduced some hardware-optimized algorithmic optimizations to the ZFP algorithm which almost doubled the performance while maintaining similar compression efficiency and on-chip resource utilization.

In Section 4, we analyze the source of the high performance and chip space overhead of the ZFP algorithm, and describe our modifications which improve on our previous work to achieve an order of magnitude performance improvement with less than 10 % of a Xilinx Virtex 7 chip.

3 PERFORMANCE ANALYSIS OF STENCIL ACCELERATION

Let's assume a system configuration with a host server and a stencil accelerator device plugged into its PCIe port. The accelerator will have a certain amount of on-board memory, as well as a much smaller amount of fast, on-chip memory. If the dataset for stencil computation is very large, it will not fit on the on-board memory of the accelerator, and will be held at the host, either in-memory or in-storage.

Assuming an ideal scenario where tiling doesn't have halo overhead, all of the stencil data needs to be streamed from host to the accelerator and back, exactly once. Unless this data rate is too fast for the stencil implementation on the accelerator to handle, the ideally achieveable maximum performance will be limited by the this data movement rate.

Under this model, we perform a simple roofline analysis to illustrate the theoretical upper bound of performance achievable under various system configurations. We compare five following scenarios, which are described in Table 1. The baseline performance numbers are modeled after our prototype FPGA environment, the Xilinx VC707 FPGA development board. *Largemem* and *Largemem2* assume the data size is small enough to fit in the on-board memory capacity, and therefore is not effected by PCIe performance limitations. *compress4* assumes the existence of a wire-speed compression accelerator, which can alleviate the performance bottleneck of both PCIe and memory.

Configuration	Description
PCIe4	4 GB/s PCIe, 10 GB/s DRAM
PCIe8	8 GB/s PCIe, 20 GB/s DRAM
Largemem	Large capacity DRAM at 10 GB/s
Largemem2	Large capacity DRAM at 20 GB/s
compress4	4 GB/s PCIe, wire-speed 4x compression

Table 1: Different configurations for roofline analysis

Figure 3 shows the roofline analysis of these configurations. Even as the peak internal performance of the accelerator grows, the performance of each configuration is either limited by PCIe bandwidth, or by on-board memory bandwidth. While one could of course use newer accelerator cards with faster PCIe or memory, the performance characteristics will remain similar, as demonstrated in many previous work on out-of-core stencil acceleration [18, 43].

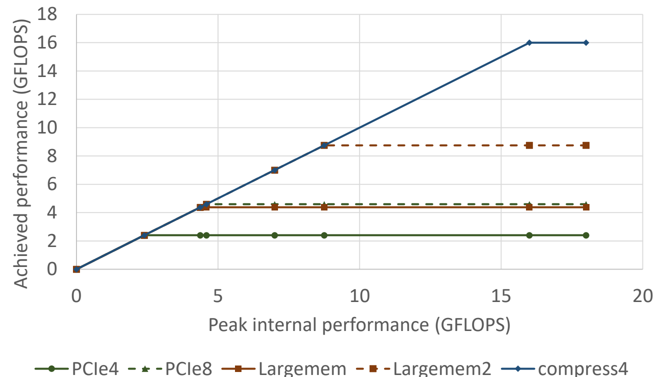

Figure 3: The stencil accelerator's performance is limited by both PCIe and DRAM's bandwidth

The analysis presented in Figure 3 shows that a system with fast, efficient compression can be an attractive solution to the bandwidth issue, as it can circumvent the communication bottleneck and achieve much higher performance even compared to systems on more capable platforms. The question now becomes, can we implement a floating-point compression accelerator with high compression ratio (ideally 4x or more), capable of achieving wire-speed.

4 HARDWARE-EFFICIENT COMPRESSION OF SCIENTIFIC DATA

BurstZ implements a hardware-optimized version of the ZFP algorithm, which we call sZFP, to achieve both efficient compression as well as high throughput. The ZFP algorithm is based on block transforms, similar to the JPEG image compression algorithm, as opposed to SZ which is prediction-based. ZFP was chosen over SZ because most components of ZFP were readily parallelizable numerical operations. The ZFP compression algorithm works in block units, each consisting of 4^d values, where d is the dimension of the block. The dimension of the block doesn't have to match the original dimensions of the data, and our implementation uses 1-dimensional blocks, where each block has 4 double-precision floating point values.

ZFP compresses data in four stages: (1) Fixed-point conversion, where all values are normalized to the largest exponent and cast to fixed-point, (2) Block transform, which allows spatially correlated values to be mostly decorrelated, for efficient compression. (3) Sequency ordering, which maps high-dimensional blocks into a 1-dimensional array such that the numbers are roughly sorted. This step is not required for 1-dimensional blocks. (4) Embedded coding, where the array is encoded one *bit-plane* at a time, until either the error bound is hit, or the provided bit budget is depleted.

The first three stages of the algorithm can be very efficiently implemented on an FPGA to support deterministic wire-speed operation, except for embedded coding, which is what our sZFP algorithm modifies. The original embedded coding stage extracts and encodes each *bit-plane*, where the Nth bit plane is constructed by gathering the Nth-bits of each element in a block. There are 64 bit planes in a block consisting of double precision floating point numbers, each consisting of d bits. A pseudocode representation of ZFP's original embedded coding algorithm can be seen in Algorithm 1. Group testing works well for compression for ZFP, having roughly sorted numbers thanks to sequency ordering can result in an early exit after emitting all 1 bits in the lower bits.

```
Data: d-bit bitplane
while bitplane != 0 do
    emit 1 ;                          ▹ Emit not-done bit
    while True do
        lsb ← bitplane[0];
        emit lsb ;                    ▹ Emit data bit
        bitplane ← bitplane >> 1;
        if lsb == 1 then break;
    end
end
emit 0 ;                              ▹ Emit done bit
```

Algorithm 1: ZFP's original embedded coding algorithm emits bits one-by-one from each bitplane

While this algorithm often results in efficient encoding, it has a high performance overhead in a hardware implementation. Because each loop iteration depends on the results of its previous iteration, and because each iteration emits only one bit of data, in the worst case a hardware implementation may require d cycles to emit a single bit-plane. The problem becomes worse for decompressors, because the offset of the next encoded bit plane depends on the encoding results of the current plane, we cannot start decoding the next bit plane until the current one is completely decoded. Even with a hypothetical encoder which can process one bit plane per cycle, for a 2-D block it can handle only 16 bits per cycle, and only 4 bits per cycle for a 1-D block. For reference, 16 bits per cycle running at 250 MHz results in 500 MB/s, which would be an order of magnitude slower than a typical PCIe link to an accelerator.

Our sZFP algorithm solves this issue in two ways: (1) A coarse-grained, header-based encoding scheme, and (2) enabling parallelism by organizing compressed data into aligned chunks, each of which can be processed independently.

4.1 sZFP modification 1: Header-Based Encoding

sZFP is a variant of the 1-dimensional ZFP algorithm, meaning it compresses floating point numbers in 4-element units. Compared to a 2-dimension or 3-dimension algorithms, an accelerator for 1-dimensional compression requires much less on-chip resources. Also, sZFP replaces the group testing-based encoding scheme to a coarse-grained header-based scheme. The design of our coarse-grained header scheme is based on two observations: (1) Putting a header per bit plane is too expensive with the 1-dimensional algorithm, because each bit plane has only four bits, and (2) After block transform and sequency ordering, the first 64-bit element of the four has a very high MSB (Most Significant Bit) index.

In order to address the first issue of header overhead, sZFP uses a coarse unit of encoding, and attaches a 2-bit header per 6 bit-planes instead of attaching a header per each bit-plane. The six bit-planes are simply concatenated, to keep the nonzero bits in the lower bits as much as possible.

However, the second issue of a high MSB in the first element harms the compression effectiveness of this scheme, because almost all sub-groups would have nonzero bits in upper bits because of the first element. To solve this second issue, sZFP treats the first element specially, and encodes its bits separately. Only the remaining three elements, which often have many leading zeros, are encoded using the coarse-grained header.

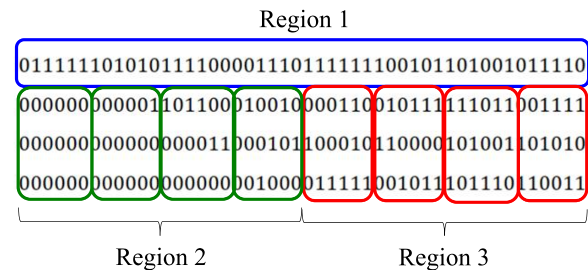

Figure 4: Three different encoding schemes are used for three different regions (Blue, green, red)

Figure 4 shows the different regions of the four-element unit which has different encoding methods, using an example block after sequency ordering. The first element, marked with a blue box, is first encoded separately. The number of bits from the first

element that is encoded depends on the requested error margin. The remaining three elements are divided into two groups (green and red), which are in turn divided into four sub-groups. Each sub-group is assigned a two-bit header. If the desired error margin is achieved within the first (green) group, encoding can stop after encoding the first element (blue) and valid sub-groups of the green group. If the error margin requires more bit planes to be encoded, the red group is encoded as well.

In most cases, we only encode up to the first 48 bit-planes in order to ensure efficient compression, as seen in Figure 4. In extremely rare cases when more than 48 bit-planes need to be encoded, we simply encode the whole block uncompressed, in order to simplify the compression accelerator.

The design of sZFP encoding ensures that one block of four elements can be encoded in at most three cycles, where one cycle is spent for each of the blue, green, and red regions. This fact, coupled with pipelining, allows sZFP to achieve very high throughput with very small on-chip resources.

4.2 sZFP modification 2: Independent Aligned Chunks

Despite the increased performance thanks to the header-based encoding scheme, a single-pipeline performance of sZFP would not be enough to keep up with PCIe or memory performance. This is especially true for decompression, because the starting offset of the next encoded block is dependent on the decompression results of the previous one, making pipelined implementation difficult. For the same reason, parallelizing decompression of a single compressed stream is also difficult.

sZFP solves this issue by organizing compressed data into independent, aligned chunks. For example, in our implementation of BurstZ, sZFP uses chunk sizes of 6 KBs. Each chunk is independent because compressed data is aligned and padded such that compressed data is aligned to the beginning of the chunk, and no block is encoded across the boundary of two chunks. Padding results in a negligible amount of wasted space (less than 32 bytes per 6 KBs), but allows simple parallelism of compression and decompression of a single stream of data. This design allows our sZFP core to achieve high enough performance to saturate even on-board DRAM performance.

5 BURSTZ ARCHITECTURE

5.1 Overall Architecture

Figure 5 shows the overall architecture of the BurstZ platform. The key point of BurstZ is that the **data exists in compressed form both on the host-side, as well as on the on-board device DRAM.** Compressed data is only decompressed on the fly when the computation engine requires it, and generated data is compressed immediately before it is stored in memory.

A BurstZ implementation consists of a host server, as well as an FPGA accelerator connected to the host server over PCIe. The BurstZ platform implementation inside the FPGA includes functionalities including PCIe and on-board memory access, as well as access arbitration for both PCIe and memory. The platform also includes multiple pipelines of compressor and decompressors, through which the computation engine can read and write data to on-board memory as well as host.

Thanks to the low on-chip resource overhead of our compression/decompression accelerators, we can afford to deploy many compressor/decompressor accelerators depending on both the bandwidth requirements as well as the access characteristics. For example, if a particular computation engine naturally has an access pattern of multiple input streams and multiple output streams, BurstZ can deploy multiple compressors and decompressors corresponding to each input and output streams, instead of the computation engine having to include logic to multiplex a single input/output stream. Similarly, if the computation engine internally has multiple pipelines for parallel performance, each pipeline can have a pair (or more) compressor and decompressor accelerators assigned to it.

5.2 Memory Arbiter

One important module in the BurstZ platform is the memory arbiter, which provides convenient shared access to the on-board DRAM while assuring high performance. As multiple entities access memory, including multiple compressor and decompressor pipelines as well as the host software via PCIe, some arbitration of memory resources is absolutely required in the platform for ease of development.

The issue is aggravated by the fact that the on-board DRAM performance is effected heavily by the access pattern. Due to architectural characteristics such as row buffers and burst lengths, memory access is typically much faster for sequential accesses compared to random access. This is the case not only for accelerator memory but for general server memory as well, and many high-performance software systems try their best to optimize their memory accesses to the underlying architecture. On our prototype hardware platform, we measured an order of magnitude performance difference between 64-byte accesses (minimum burst length) and 8 KB accesses (row buffer size). When there are multiple entities accessing memory at the same time, even if each entity's access pattern is sequential, their interleaved access patterns may be very random, harming memory performance.

To achieve high performance, our memory arbiter exposes a burst interface, where each endpoint must first send a burst request before reading or writing data. The scheduler inside the memory arbiter performs memory access in burst units, so that high performance can be achieved as long as burst sizes are relatively large. The internal architecture of the memory arbiter can be seen in Figure 6.

The arbiter is parameterized so that the number of endpoints can be configured at compile time. Each endpoint interface also includes enough buffer space to ensure deadlocks cannot happen by an endpoint's mistake. The scheduler will only start a burst only when there is enough read buffer space to either accommodate a read request, or enough data in the write buffer to finish a write burst.

5.3 Stencil Core Architecture

To evaluate BurstZ, we implement a typical 3D 7-point stencil computation core on our prototype BurstZ platform. Figure 7 shows the view of the dataset from the accelerator point of view. A 3D stencil operates on a 3-dimensional grid of values, as seen in Figure 7(a).

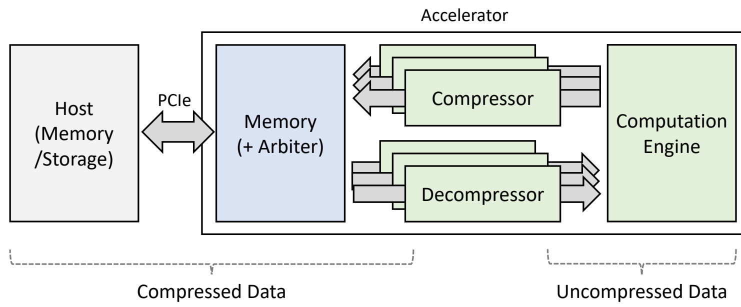

Figure 5: The overall architecture of BurstZ. Data is stored compressed until it is used by the computation engine

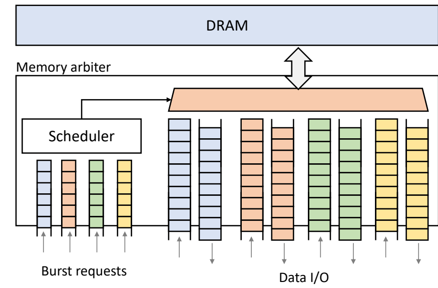

Figure 6: The memory arbiter provides high-performance multiplexing to multiple end-points

We use nX, nY and nZ to denote the number of values in the dimensions x, y and z, respectively. A 2-D space of size $nX \times nY$ is called a "plane". There are nZ planes in total. As seen in Figure 7(b), a 7-point 3D stencil reads three planes (e.g., z = 0, 1, 2) from the on-board DRAM, in order to update plane 1 point by point. While this processing is ongoing, we can load a new plane (e.g., z = 3) to the space used by plane 1. Once plane 1 is done, we can begin to update plane 2, and so on.

We implement a very simple stencil core without in-memory tiling, which must read all data elements three times, once for each input plane. But thanks to the high memory bandwidth made possible by wire-speed compression, we demonstrate our implementation **outperforms even the projected performance of an ideally tiled accelerator by over** 2×. We emphasize that we do not argue that our stencil core design is superior to existing tiling-based methods. It is merely an example to demonstrate the capabilities of BurstZ with multiple I/O pipelines, and to emphasize that the compression/decompression cores provide such high data bandwidth, they allow us to outperform highly optimized cores even with such a simple design.

In order to improve memory re-use, and improve the memory access bottleneck, we maintain three most recently accessed rows of each plane in fast on-chip memory queues, so that each stencil operation can be done from on-chip memory. The conceptual location of example buffered rows can be seen in Figure 7(c).

Figure 8 shows how we load the plane's contents to on-chip memory row by row. Since we need three consecutive rows to begin the computation, we create two row buffers for each input plane. The two buffers are used as a circular buffer that always hold two most recently input rows in its plane. The two buffers, coupled with the input, are fed into the stencil core, inserting 9 elements into the stencil core every cycle. These 9 elements are the points in each 2-dimensional yz-plane of the 3-dimensional cube bounding the 7-point stencil. The stencil core is designed such that it takes each 2-dimensional yz-plane per cycle in a pipelined manner.

Because our stencil core does not implement in-memory tiling, the three input planes must be read from on-board memory in parallel. BurstZ supports this using three separate decompressor pipelines. The stencil core requires only one compressor pipeline because only one plane is output at once.

In order to facilitate high parallelism and bandwidth, each element in the row buffer actually consists of multiple floating point values. For example, in our prototype implementation the datapath is 32-bytes wide, meaning four double-precision floating point values are entered into the stencil core every cycle, per input element. The internals of the stencil core is designed such that it can achieve wire-speed processing via an array of floating point operators.

5.4 Compression Accelerator Architecture

In order for compression to be useful, it must achieve wire-speed, in order to keep up with the bandwidth of the memory and computation engine. As described in Section 4, a single pipeline of decompressor or compressor is often not enough to keep up with the tens of GB of throughput a stencil core requires. sZFP is designed to support parallelized compression and decompression of a single data stream using aligned chunks. In our prototype implementation, each chunk is 6 KB in size. This section describes

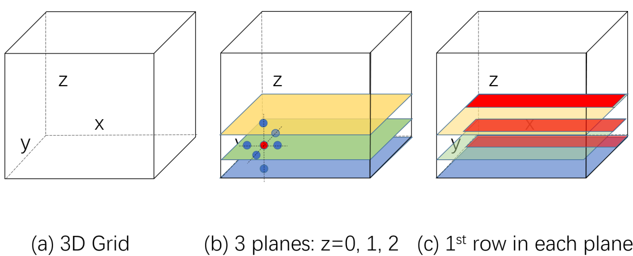

Figure 7: The basic principle of 3D stencil computation

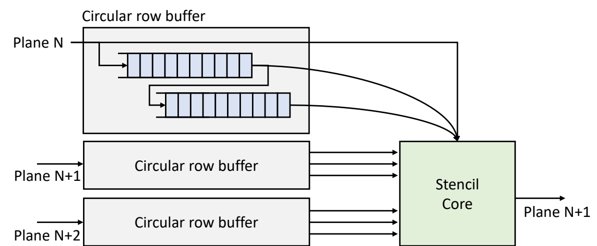

Figure 8: Three sets of two on-chip BRAM row buffers are used by the stencil core

the efficient architecture for parallelized sZFP compression and decompression accelerators.

5.4.1 Decompressor Architecture. The major performance bottleneck of the decompressor is the decoding stage, as described in Section 4. This is because the algorithm can't know the bit offset of the next encoded 4-element block, until the current block is decoded. All other stages of the decompression algorithm, including the block transform and floating-point conversion, can easily support wire-speed processing with a single pipeline.

Instead of simply parallelizing the entire decompressor pipeline, wasting chip space, our implementation only replicates the decoder modules. The internal architecture of a decompressor accelerator can be seen in Figure 9. The input stream is broken into chunks, and distributed in a round-robin fashion to an array of decoders. The decoded results are collected at the block transform stages in-order, after which everything else can be processed at wire-speed.

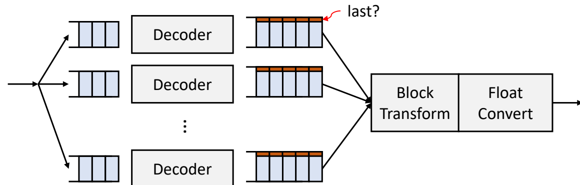

Figure 9: A multi-pipeline sZFP decompressor accelerator

Because we cannot predict how much uncompressed data will be generated from a *chunk*-sized input, the decoder module is programmed to tag each output element with a *last?* flag, telling the block transform stage if this element is the last to be decoded from a chunk. When the block transform stage encounters a last element, it can move on to the next decoder. In order to support high performance not bottlenecked by any particular decoder, each decoder has both a chunk-size input buffer, and an output buffer of size $chunk \times 4$, so that each decoder can work at its own pace without causing head-of-line blocking.

5.4.2 Compressor Architecture. The design of a wire-speed compressor pipeline is much simpler compared to a wire-speed decompressor pipeline. Since encoding each 4-element block still takes up to 3 cycles, the encoder is still the bottleneck, similar to the decompressor. However, because each uncompressed input element can simply be round-robin distributed to each encoder without having to wait until it is encoded, the encoder array does not need to work in terms of aligned chunks, but with individual elements.

Figure 10 shows the internal architecture of the compression module. After block transform, the transformed blocks are distributed round-robin to an array of encoders. After encoding, the encoded blocks are received in a round-robin way by the shuffler, which bit-packs the compressed blocks, and also handles chunk-alignment.

5.5 Implementation Details

We have implemented a BurstZ prototype on a Xilinx VC707 FPGA development board. The VC707 board is not a high-end FPGA by modern server standards, and is equipped with a Xilinx Virtex-7

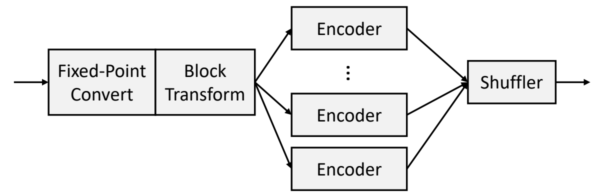

Figure 10: A multi-pipeline sZFP compressor accelerator

FPGA, as well as 1 GB of on-board DRAM capable of up to 11 GB/s of DDR3 bandwidth. The board plugs into the host via a PCIe Gen2 x8 link, which is capable of 4 GB/s duplex bandwidth.

Table 2 shows the breakdown of on-chip LUT resource utilization of various components in the BurstZ platform, including the PCIe, memory, arbiter, three decompressor pipelines, as well as one compressor pipeline. The platform consumes about 40% of the on-chip resources of our prototype platform, and less than **5%** of the on-chip resources of a modern, high-end FPGA such as the Virtex Ultrascale+. Besides LUTs, the BurstZ platform consumes less than 500 KB of on-chip Block RAM resources, leaving the majority of on-chip memory resources to the computation engine.

Accelerators with higher-performance FPGAs will support more, faster compute engines, which in turn will require more compression pipelines. Thanks to the very low resource requirements of BurstZ, we project this platform will be able to scale to the computation capabilities of modern and future accelerator platforms.

6 PERFORMANCE EVALUATION

We demonstrate the effectiveness of our BurstZ platform in two parts: (1) The effectiveness of the sZFP algorithm and its accelerator implementation, and (2) The application performance benefits of BurstZ on a 3D stencil core. The application performance benefit is demonstrated by comparing the measured performance of our prototype implementation against various other, conventional architectures implemented on the same hardware. The comparison includes the projected performance with ideal tiling and caching, which achieves the upper bound performance achievable on the same hardware platform.

6.1 Benchmark Datasets

In order to evaluate our system under realistic scenarios, we use real-world datasets from the *Scientific Data Reduction Benchmarks (SDRBench)* [4], which includes various real-world datasets from fields including climate simulation, molecular dynamics, and cosmology simulations. We selected three datasets from SDRBench

Module	LUTs	VC707%	VCU118%
Platform (PCIe+DRAM+Arbiter)	22K	7%	<1%
1x Decompressor	26K	9%	1%
1x Compressor	25K	8%	<1%
Total	125K	41%	<5%

Table 2: FPGA LUTs usage breakdown of the BurstZ platform for stencil computation

Name	Description
Configurations with sZFP compression	
B'z3	BurstZ with error bound of 1E-3
B'z4	BurstZ with error bound of 1E-4
B'z5	BurstZ with error bound of 1E-5
B'z6	BurstZ with error bound of 1E-6
Configurations with no compression	
Nocomp	BurstZ's stencil core with no compression
Fastmem	BurstZ's stencil with unlimited DRAM bandwidth
Largemem	BurstZ stencil with enough memory to hold dataset
Ideal	Core with ideal tiling and caching
IdealLarge	Ideal with enough memory to hold dataset

Table 3: Evaluated accelerator configurations

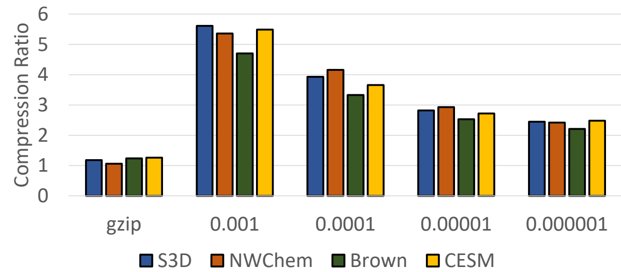

Figure 11: Compression efficiency of sZFP, across four datasets, with varying error bounds

which use double-precision floating point data (S3D, NWChem, and Brown), and selected one which uses single-precision floating point data (CESM-ATM), and cast it to double precision values. When a dataset was too small for realistic evaluation, we simply replicated the whole dataset multiple times to obtain a larger dataset.

6.2 Evaluation Configurations

For evaluation of our BurstZ prototype, we measured the performance of the system with four different configurations where the error bound of the compression algorithm was set to either 1E-3, 1E-4, 1E-5, or 1E-6. These are typical compression parameters used in real-world scientific computing scenarios [32].

We compared the performance of BurstZ against various other accelerator architectures that could be implemented on a hardware platform with the same component performances. Compared results include the ideal, unrealistic systems such as those with ideal tiling and caching, as well as accelerators with large enough memory to always accommodate the whole dataset.

Table 3 lists the system configurations for BurstZ and others. **Ideal** and **IdealLarge** represents performance upper limits a stencil accelerator can achieve on the same hardware platform, when either the dataset is realistically large (Ideal), or if the dataset is smaller than on-board memory capacity (IdealLarge). Both systems assume ideal situations with ideal tiling and caching, as well as no halo overhead, meaning the entire dataset is sweeped by the stencil core exactly once, and this memory movement is the only performance bottleneck. For **Ideal**, the on-board memory bandwidth is shared across PCIe data loading to memory, as well as the stencil core reading the loaded data exactly once.

6.3 Prototype Platform Evaluation

The VC707 FPGA development board on which we implemented our BurstZ prototype, is equipped with a PCIe Gen2 x8 link to the host server, as well as 1 GB of on-board DDR3 DRAM. After protocol and flow control overhead, our PCIe implementation was able to achieve 3.1 GB/s of duplex communication bandwidth between the host and FPGA. As for the on-board memory, our memory controller and arbiter were able to achieve almost full advertised performance of the memory module, which is approximately 11 GB/s, shared between read and write requests. Our compression accelerators and stencil core example were implemented on top of this hardware platform.

6.4 sZFP Evaluation

6.4.1 Compression Efficiency. Figure 11 shows the efficiency of the sZFP compression algorithm across benchmark datasets and error bounds. We also provide comparison against gzip, which is a commonly used integer based compression algorithm. As shown in Figure 11, gzip typically does not work well with floating point numbers. The compression efficiency of sZFP slowly declines, as the error bound becomes stricter, following the pattern shown also by the original ZFP algorithm. Configured with reasonable error margins, sZFP consistently provides 3x – 4x compression. In Section 6.5, we will show that this compression efficiency is sufficient to remove the communication bottleneck.

6.4.2 Accelerator Performance. Figure 12, and Figure 13 show the compression and decompression performance of the sZFP accelerator, respectively, showing an order of magnitude higher performance compared to software, as well as the reported performance numbers of FPGA-based hardware accelerator of unmodified lossy floating point compression algorithms. Performance was measured across the same four benchmark datasets, with the four error bound values. The *Software* region in the figures represents the range of achieved performance by single-thread software implementations provided by the algorithm authors. The *Vanilla FPGA* region presents the best, single-core performance of unmodified algorithm, either published by GhostSZ [50], an FPGA implementation of SZ, or measured average of the FPGA implementation of ZFP from our previous work [45]. The figures show that with lenient error bounds, both the compressor and decompressor achieve almost wire-speed performance with the default setting, which is multiple times faster than the best-effort FPGA implementations of the unmodified algorithms.

While these figures seem to show the compression accelerators failing to achieve wire-speed performance with stricter error bounds, this is not an inherent limitation of the algorithm. As described in Section 5, our compression accelerators are designed to have a configurable number of internal pipelines to achieve parallelism. On the BurstZ platform, the default compressor is configured to have two internal pipelines, and the decompressor is configured to have five internal pipelines, which was just enough to achieve wire-speed in many realistic scenarios. Figure 14 shows the scaling

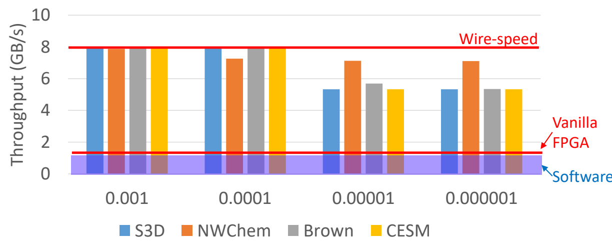

Figure 12: Compression performance of BurstZ's sZFP accelerator

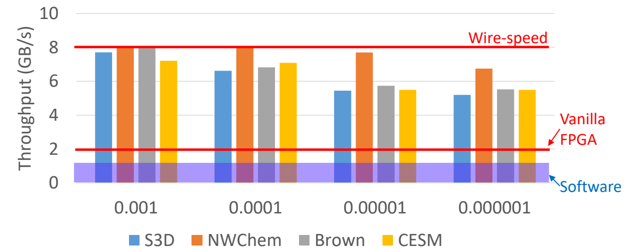

Figure 13: Decompression performance of BurstZ's sZFP accelerator

of the ***worst-case*** performance as the number of internal pipelines increase. Meaning, if the compressor was configured to have three internal pipelines instead of two, and the decompressor was configured to have eight internal pipelines instead of five, the compression and decompression accelerators would have invariantly achieved wire-speed performance without fail. This change would have incurred a proportional increase in on-chip resource utilization, but considering the low resource utilization numbers given in Table 2, even that resource utilization will be quite low.

6.4.3 Comparisons Against ZFP. Compared to the vanilla ZFP, sZFP often achieves a significantly lower compression ratio. However, sZFP has various characteristics that make it a better algorithm for the purpose of BurstZ.

First of all, the encoding scheme allows for extremely efficient hardware implementation, while achieving very high performance. As we show in Section 6.4.2, our sZFP accelerator demonstrates wire-speed performance, which is 8 GB/s in our case, while consuming a fraction of on-chip resources. This makes sZFP quite desirable, considering the on-chip resource utilization presented in Table 2 is less than a third of the available FPGA implementation of unmodified ZFP [45], while achieving multiple times faster performance. This is assuming the chip capacity of the Intel Arria 10 GX 1150 with 427,200 ALMs is comparable to the Xilinx XC7VX485T with 485,760 Logic Cells, but the utilization difference is significant enough to be meaningful.

Secondly, sZFP shows very consistent compression ratios, achieving compression ratios similar to the original ZFP algorithm for datasets that are not easily compressed, providing a similar lower bound in efficiency. For example, the original ZFP algorithm was able to reduce the size of the **NWChem** dataset by a factor by over 21× with an error bound of 0.001, which is much better than the 4.7× compression of sZFP. However, for the dataset **Brown**, the original ZFP demonstrated only about 6× compression, while sZFP also achieved about 5×. These characteristics allow BurstZ to use sZFP to achieve consistently high performance across various scientific datasets. Consistently achieving 4–5× compression is sufficient for the purpose of BurstZ, which is to better balance the computation and communication performances.

6.5 End-to-End Application Performance

Figure 15 compares the end-to-end performance of stencil computation, on the various system configurations described in Table 3, with performance numbers normalized to **Largemem**. For each benchmark column, the left five bars represent the BurstZ system using each of the error bounds for compression. The right four bars are different stencil accelerator architectures implemented on the same hardware platform.

In terms of raw performance, Largemem corresponds to 2.4 double-precision GFLOPS, meaning the measured BurstZ systems measure between 5.25 to 7 double-precision GFLOPS. This corresponds to 11 to 14 single-precision GFLOPS as performance is entirely memory bound. Considering that the Intel stencil reference implementation on an FPGA of similar scale demonstrates 7 single-precision GFLOPS with a single pipeline [24], we can be confident our stencil accelerator has a reasonable design. To achieve higher GFLOPS, we can use the same temporal blocking methods the Intel design used, in order to achieve almost 200 GFLOPS. But since these optimizations will affect all compared system configurations similarly, and are orthogonal to the data movement issue we are addressing, we present normalized performance results.

It can be seen that even with the most stringent error bound (B'z6 with error bound of 1E-6), the BurstZ system outperforms all other configurations, and performs on par with **IdealLarge**, which is an unrealistic system with not only ideal tiling, caching, and no halo overhead, but also on-board DRAM large enough to accommodate the entire dataset. When compared against **Ideal**, which is an upper-bound performance projection of a system streaming data from the host, even the slowest B'z6 system consistently achieved almost 2× the performance, with B'z3 achieving almost 3×.

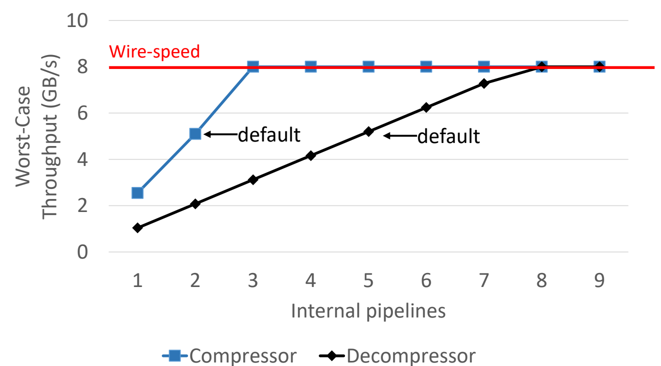

Figure 14: Worst-case performance of the compression accelerator as the number of internal pipelines increase

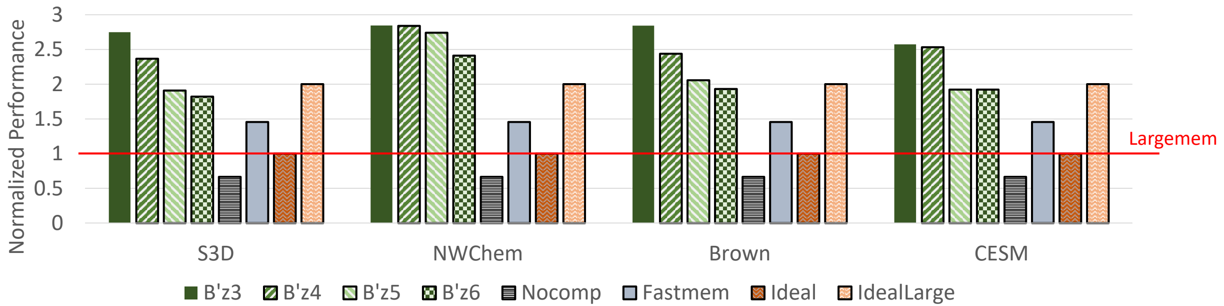

Figure 15: BurstZ application evaluation outperforms even in-memory systems with ideal caching

This is a significant performance improvement, considering that the BurstZ systems have an inherent disadvantage of lacking in-memory tiling, and must read the input data from on-board memory three times, once for each read plane. When compared against systems with similar data access patterns, but lacking compression, all BurstZ configurations achieve over 3× the performance of **Nocomp**, over 2× the performance of **Largemem**, and consistently outperforms even **Fastmem**.

For all measured BurstZ systems, the biggest performance limiting factor is not the PCIe, but the on-board DRAM performance, meaning the problem has now become a more classical scientific computing issue of optimizing memory accesses. The memory arbiter serves six endpoints: PCIe read, PCIe write, three decompressors, and one compressor. All endpoints have roughly the same sustained throughput, which limited each endpoint's throughput to 1.8 GB/s on our platform with 11 GB/s total memory bandwidth. After compression, this translates to over 6 GB/s of throughput per I/O port on the computation engine side, which is lower than the wire-speed of 8 GB/s. A traditional solution of a more optimized stencil with better tiling and caching will reduce the memory pressure, further improving performance.

7 CONCLUSION AND DISCUSSION

We present BurstZ, a bandwidth-efficient scientific computing accelerator platform for large data. BurstZ uses a novel, hardware-optimized compression algorithm sZFP and removes the PCIe bottleneck, which is the primary performance limiting factor of large-scale scientific computing acceleration. In fact, BurstZ's sZFP accelerators are so efficient that it drastically increases the effective on-board memory bandwidth, which allows our example accelerator to outperform even completely in-memory systems.

We believe the impact of a BurstZ-like system on scientific computing will be significant for multiple reasons. First, it will reduce the cost of computation as accelerator performance becomes less bound to expensive on-board memory capacity. Second, it will also allow handling of much larger problems than was possible before, because removing the PCIe bottleneck also means fast secondary storage devices such as NVMe flash can support the full computation performance of an accelerator. Furthermore, we project that improving the effective performance of communication via compression can also remove the network bottleneck of distributed systems.

We have designed BurstZ as a general infrastructure which will be beneficial for not only stencil computation, but also many other data-intensive scientific applications. In the future, we plan to use BurstZ to explore various scientific computing workloads to improve the speed and reduce the cost of scientific discovery.

ACKNOWLEDGMENTS

This work was partially funded by NSF (CNS-1908507)

REFERENCES

[1] Muhammad Shoaib Bin Altaf and David A Wood. 2017. Logca: A high-level performance model for hardware accelerators. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 375–388.

[2] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. 2001. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1033–1051.

[3] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2014. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*. Springer, 1–35.

[4] F. Cappello, M. Ainsworth, J. Bessac, Martin Burtscher, Jong Youl Choi, E. Constantinescu, S. Di, H Guo, Peter Lindstrom, and Ozan Tugluk. Accessed April 2020. Scientific Data Reduction Benchmarks. https://sdrbench.github.io/

[5] Siddhartha Chatterjee, Alvin R Lebeck, Praveen K Patnala, and Mithuna Thottethodi. 2002. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems* 13, 11 (2002), 1105–1123.

[6] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. 2018. SODA: stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[7] Yann Collet et al. 2013. Lz4: Extremely fast compression algorithm. *code.google.com* (2013).

[8] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. 2015. An optimal microarchitecture for stencil computation acceleration based on nonuniform partitioning of data reuse buffers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 3 (2015), 407–418.

[9] Mayank Daga, Ashwin M Aji, and Wu-chun Feng. 2011. On the efficacy of a fused CPU+ GPU processor (or APU) for parallel computing. In *2011 Symposium on Application Accelerators in High-Performance Computing*. IEEE, 141–149.

[10] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. 2010. Auto-tuning stencil computations on multicore and accelerators. *Scientific Computing on Multicore and Accelerators* (2010), 219–253.

[11] Gaël Deest, Nicolas Estibals, Tomofumi Yuki, Steven Derrien, and Sanjay Rajopadhye. 2016. Towards scalable and efficient FPGA stencil accelerators.

[12] Gaël Deest, Tomofumi Yuki, Sanjay Rajopadhye, and Steven Derrien. 2017. One size does not fit all: Implementation trade-offs for iterative stencil computations on FPGAs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–8.

[13] Cornelius Demuth, Miguel AA Mendes, Subhashis Ray, and Dimosthenis Trimis. 2014. Performance of thermal lattice Boltzmann and finite volume methods for the solution of heat conduction equation in 2D and 3D composite media with inclined and curved interfaces. *International Journal of Heat and Mass Transfer* 77 (2014), 979–994.

[14] Peter Deutsch. 1996. DEFLATE compressed data format specification version 1.3. (1996).

[15] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 ieee international parallel and distributed processing symposium (ipdps)*. IEEE, 730–739.

[16] James Diffenderfer, Alyson L Fox, Jeffrey A Hittinger, Geoffrey Sanders, and Peter G Lindstrom. 2019. Error Analysis of ZFP Compression for Floating-Point Data. *SIAM Journal on Scientific Computing* 41, 3 (2019), A1867–A1898.

[17] Keisuke Dohi, Kota Fukumoto, Yuichiro Shibata, and Kiyoshi Oguri. 2013. Performance modeling and optimization of 3-D stencil computation on a stream-based FPGA accelerator. In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*. IEEE, 1–6.

[18] Toshio Endo. 2016. Realizing out-of-core stencil computations using multi-tier memory hierarchy on gpgpu clusters. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 21–29.

[19] Toshio Endo and Guanghao Jin. 2014. Software technologies coping with memory hierarchy of GPGPU clusters for stencil computations. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 132–139.

[20] Rosa Filgueira, David E Singh, Alejandro Calderón, and Jesús Carretero. 2009. CoMPI: enhancing mpi based applications performance and scalability using run-time compression. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 207–218.

[21] Michael Schacht Hansen and Thomas Sangild Sørensen. 2013. Gadgetron: an open source framework for medical image reconstruction. *Magnetic resonance in medicine* 69, 6 (2013), 1768–1776.

[22] Michael A Heroux, Jonathan Carter, Rajeev Thakur, Jeffrey Vetter, Lois Curfman McInnes, James Ahrens, and J Robert Neely. 2018. *ECP software technology capability assessment report*. Technical Report. Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States).

[23] Minh Quan Ho, Christian Obrecht, Bernard Tourancheau, Benoît Dupont de Dinechin, and Julien Hascoet. 2017. Improving 3D Lattice Boltzmann Method stencil with asynchronous transfers on many-core processors. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 1–9.

[24] Intel. 2018. *AN 870: Stencil Computation Reference Design*. https://www.intel.com/content/www/us/en/programmable/documentation/abw1532533443842.html

[25] Sian Jin, Pascal Grosset, Christopher Biwer, Jesus Pulido, Jiannan Tian, Dingwen Tao, and James Ahrens. 2020. Understanding GPU-Based Lossy Compression for Extreme-Scale Cosmological Simulations. In *2020 The 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[26] Ryohei Kobayashi, Shinya Takamaeda-Yamazaki, and Kenji Kise. 2012. Towards a low-power accelerator of many FPGAs for stencil computations. In *2012 Third International Conference on Networking and Computing*. IEEE, 343–349.

[27] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[28] Pierre Lallemand and Li-Shi Luo. 2003. Theory of the lattice Boltzmann method: Acoustic and thermal properties in two and three dimensions. *Physical review E* 68, 3 (2003), 036706.

[29] Daniel Lemire, Nathan Kurz, and Christoph Rupp. 2018. Stream VByte: Faster byte-oriented integer compression. *Inform. Process. Lett.* 130 (2018), 1–6.

[30] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.

[31] LLNL. Accessed April 2020. ZFP related projects. https://computing.llnl.gov/projects/floating-point-compression/related-projects.

[32] Tao Lu, Qing Liu, Xubin He, Huizhang Luo, Eric Suchyta, Jong Choi, Norbert Podhorszki, Scott Klasky, Mathew Wolf, Tong Liu, et al. 2018. Understanding and modeling lossy compression schemes on HPC scientific data. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 348–357.

[33] Naoya Maruyama and Takayuki Aoki. 2014. Optimizing stencil computations for NVIDIA Kepler GPUs. In *Proceedings of the 1st international workshop on high-performance stencil computations, Vienna*. 89–95.

[34] Richard Membarth, Frank Hannig, Jürgen Teich, and Harald Köstler. 2012. Towards domain-specific computing for stencil codes in HPC. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 1133–1138.

[35] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 2010. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[36] Xinyu Niu, Qiwei Jin, Wayne Luk, Qiang Liu, and Oliver Pell. 2012. Exploiting run-time reconfiguration in stencil computation. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 173–180.

[37] Markus F.X.J. Oberhumer. 2017. *LZO real-time data compression library*. http://www.oberhumer.com/opensource/lzo/

[38] Koji Okina, Rie Soejima, Kota Fukumoto, Yuichiro Shibata, and Kiyoshi Oguri. 2016. Power performance profiling of 3-D stencil computation on an FPGA accelerator for efficient pipeline optimization. *ACM SIGARCH Computer Architecture News* 43, 4 (2016), 9–14.

[39] Leigh Orf. 2019. A Violently Tornadic Supercell Thunderstorm Simulation Spanning a Quarter-Trillion Grid Volumes: Computational Challenges, I/O Framework, and Visualizations of Tornadogenesis. *Atmosphere* 10, 10 (2019), 578.

[40] Liu Peng, Richard Seymour, Ken-ichi Nomura, Rajiv K Kalia, Aiichiro Nakano, Priya Vashishta, Alexander Loddoch, Michael Netzband, William R Volz, and Chap C Wong. 2009. High-order stencil computations on multicore clusters. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–11.

[41] Toomas Remmelg, Thibaut Lutz, Michel Steuwer, and Christophe Dubach. 2016. Performance portable GPU code generation for matrix multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*. 22–31.

[42] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. 2013. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems* 25, 3 (2013), 695–705.

[43] Takashi Shimokawabe, Toshio Endo, Naoyuki Onodera, and Takayuki Aoki. 2017. A stencil framework to realize large-scale computations beyond device memory capacity on GPU supercomputers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 525–529.

[44] Sambit K Shukla, Yang Yang, Laxmi N Bhuyan, and Philip Brisk. 2013. Shared memory heterogeneous computation on PCIe-supported platforms. In *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 1–4.

[45] Gongjin Sun and Sang-Woo Jun. 2019. ZFP-V: Hardware-Optimized Lossy Floating Point Compression. In *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 117–125.

[46] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama. 2016. OpenCL-based FPGA-platform for stencil computation and its optimization methodology. *IEEE Transactions on Parallel and Distributed Systems* 28, 5 (2016), 1390–1402.

[47] Shuo Wang and Yun Liang. 2017. A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[48] Terry A. Welch. 1984. A technique for high-performance data compression. *Computer* 6 (1984), 8–19.

[49] Benjamin Welton, Dries Kimpe, Jason Cope, Christina M Patrick, Kamil Iskra, and Robert Ross. 2011. Improving i/o forwarding throughput with data compression. In *2011 IEEE International Conference on Cluster Computing*. IEEE, 438–445.

[50] Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Anthony Skjellum, and Martin C Herbordt. 2019. GhostSZ: A Transparent FPGA-Accelerated Lossy Compression Framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 258–266.

[51] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 153–162.

[52] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. High-performance high-order stencil computation on FPGAs using opencl. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 123–130.