UNIVERSITY OF CALIFORNIA,
IRVINE

Scalable Scientific Computation Acceleration Using Hardware-Accelerated Compression

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Gongjin Sun

Dissertation Committee:
Professor Sang Woo Jun, Chair
Professor Nikil Dutt
Professor Elaheh Bozorgzadeh

2022

© 2022 Gongjin Sun

DEDICATION

To my parents, my sister, my fiancée and those who support and help my pursuing my Ph.D.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Sang-Woo Jun. You give me a chance to explore the world of FPGA with a totally new and amazing experience. Your brilliant insights, extreme patient guidance, and rich experience and strong expertise make me grow quickly in this area. I feel so honored to become your first student in your professor career and am so grateful to you for all that you have done including writing and polishing code, papers, presentations, a lot of research suggestions, job hunting, and many more.

I also would like to thank Professor Alex Veidenbaum. Your advising makes me get deep understanding in the area of processor microarchitecture, especially in hardware prefetcher. I appreciate so many talks with you and a lot of useful research advice you gave me.

Next, I would like to thank Professor Anton Burtsev for discussing OS knowledge with me, and, thank Professor Alex Nicolau, Professor Nikil Dutt, Professor Elaheh Bozorgzadeh, Professor Ahmed Eltawil for your service, your valuable comments and suggestions in my advancement, defense and dissertation.

I also would like to thank many staff in Computer Science department, Donald Bren School of Information and Computer Sciences, Center for Embedded Cyber-Physical Systems (CECS), International Center, UCI housing office, Verano Place housing office, and other UCI departments. Thank you so much for your assistance during my PhD program.

Then, I thank Jinpyo Kim for your mentoring in my internship in the vmware and your help in my job hunting.

I also thank my friends and colleagues during my PhD life: Zhi Chen, thank you for giving me many useful suggestions in my first few PhD years; Junjie Shen, we've been labmate for a long time and it is so nice to work and discuss with you, and you give me so much help, so many useful suggestions, thank you!; Wentao Zhu, you always give me support, help and encouragement, especially when I was depressed. I value a lot of talk between us. Sajjad Taheri, Aniket Shivam, Biswadip Maity, Tongsheng Geng and others. Thank you all for your friendship, support, suggestions and collaboration!

I thank YoungBong Kim. As a visiting member in our ARDA group, you taught us a lot of useful storage knowledge.

I would like to thank the ARDA group members: Jeffrey Chen, I miss the time we went to the scary farm for fun at Halloween, went to the comic book store and hung out on the streets in San Diego; Seongyoung Kang, I value and miss the time you, me and Sang-Woo work together to complete a lot of meaningful research work! Se-Min Lim, JiyoungAn, Esmerald Aliaj. I'm very impressed by your work and your progress. I believe all of you will make our group better and better in the future!

I sincerely thank the support from Intel's Academic Compute Environment for the HLS part of our ZFP-V work; BurstZ+ was partially funded by NSF (CNS-1908507), as well as with

generous donations from VMware Research. Thank NSF and VMware Research!

My PhD program is financially supported by Computer Science department, Professor Alex Veidenbaum, and Professor Sang-Woo Jun. I really appreciate your support!

Last but not least, I appreciate a lot the support from my parents, my sister, my fiancée and friends during my PhD career.

Thank you to everyone who have helped me during my PhD career!

VITA

Gongjin Sun

EDUCATION

Doctor of Philosophy in Computer Science **2022**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Science **2013**
University of Science and Technology of China *Hefei, China*

Bachelor of Science in Computer Sciences **2006**
Northwestern Polytechnical University *Xi'an, China*

RESEARCH EXPERIENCE

Graduate Research Assistant **2015–2022**
University of California, Irvine *Irvine, California*

Visiting Research Assistant **2011–2013**
State Key Laboratory of Computer System and Architecture, ICT, CAS *Beijing, China*

TEACHING EXPERIENCE

Reader **2015–2018**
University of California, Irvine *Irvine, CA*

REFEREED JOURNAL PUBLICATIONS

BurstZ+: Eliminating The Communication Bottleneck of Scientific Computing Accelerators via Accelerated Compression **2022**
ACM Transactions on Reconfigurable Technology and Systems (TRETS)

REFEREED CONFERENCE PUBLICATIONS

Bandwidth Efficient Near-Storage Accelerator for High-Dimensional Similarity Search **Dec 2020**
International Conference on Field-Programmable Technology (FPT)

ColumnBurst: a near-storage accelerator for memory-efficient database join queries **August 2020**
ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)

BurstZ: A Bandwidth-Efficient Scientific Computing Accelerator Platform for Large-Scale Data **June 2020**
ACM International Conference on Supercomputing (ICS)

ZFP-V: Hardware-Optimized Lossy Floating Point Compression **Dec 2019**
International Conference on Field-Programmable Technology (FPT)

Combining Prefetch Control and Cache Partitioning to Improve Multicore Performance **May 2019**
International Parallel and Distributed Processing Symposium (IPDPS)

ABSTRACT OF THE DISSERTATION

Scalable Scientific Computation Acceleration Using Hardware-Accelerated Compression

By

Gongjin Sun

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Professor Sang Woo Jun, Chair

Hardware accelerators such as GPUs and FPGAs can often provide enormous computing capabilities and power efficiency, as long as the working set fits in the on-board memory capacity of the accelerator. But if the working set does not fit, data must be streamed from the larger host memory or storage, causing performance to be limited by the slow communication bandwidth between the accelerator and the host. While compression is an effective method to reduce data storage and movement overhead, it has not been very useful in solving this issue due to efficiency and performance limitations. This is especially true for scientific computing accelerators with heavy floating-point arithmetic, because efficiently compressing floating-point numbers requires complex, floating-point specific algorithms.

This dissertation addresses the host-side bandwidth issue of accelerators, specifically FPGA accelerators, using a series of hardware-optimized compression algorithms. Since typical compression algorithms are not designed with efficient hardware implementation in mind, we explore and implement variants of existing algorithms for high performance and efficiency. We demonstrate the impact of our ideas using two classes of applications: Grid-based scientific computing, and high-dimensional nearest neighbor search. We have implemented a scientific computing accelerator platform (BurstZ+), which uses a class of novel error-controlled lossy floating-point compression algorithms (ZFP-V Series). We demonstrate that BurstZ+ can

completely remove the host-accelerator communication bottleneck for accelerators. Evaluated against hand-optimized kernel accelerator implementations without compression, our single-pipeline BurstZ+ prototype outperforms an accelerator without compression by almost 4×, and even an accelerator with enough memory for the entire dataset by over 2×. We have also developed a near-storage high-dimensional nearest neighbor search accelerator (ZipNN) which uses a hardware-optimized Group Varint compression algorithm to remove the host-side communication bottleneck. Our ZipNN prototype outperforms an accelerator without compression by 6×, and even much costlier in-memory multithreaded software implementations by over 2×.

Chapter 1

Introduction

While heterogeneous computing systems equipped with application-specific hardware accelerators are becoming staples in datacenters due to their high performance and power efficiency, their performance is often limited not by computation capacity, but by host-side 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, equipped with fast on-board memory. 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 [118, 27, 19, 5, 41, 117]. 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 [39, 129, 23, 26, 143, 111].

Figure 1.1 shows this issue. On the host side, we have large volume and fast memory devices. On the device side, the computation engine can work very fast and the engine may be equipped with fast memory device as well. However, PCIe communication bandwidth

becomes the performance bottleneck between them due to its low bandwidth.

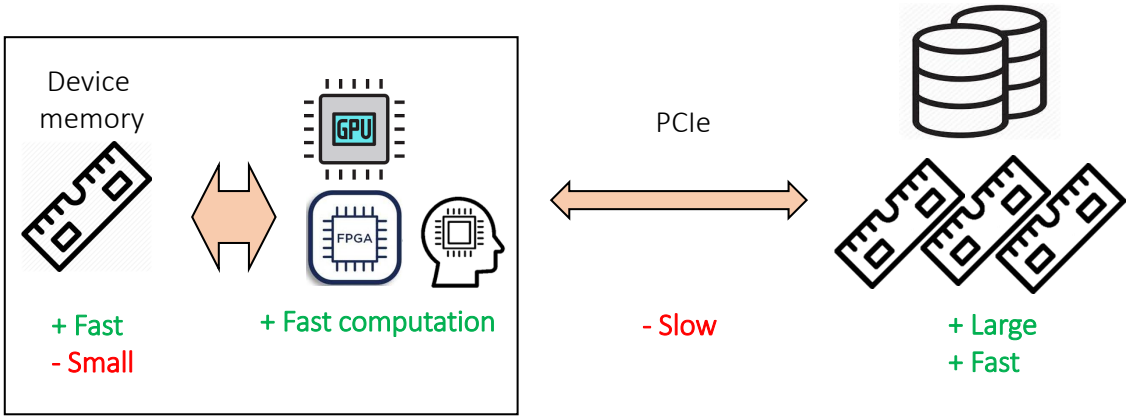


Figure 1.1: PCIe bandwidth becomes the bottleneck between Host and Accelerator

Data Compression is a commonly used technique that can effectively save storage space and the overhead of data movement. If the data movement over the communication channel can be in compressed form, and if the compression and decompression can be done fast enough to keep up with the computation and memory performances, the accelerator can enjoy a sufficiently high effective bandwidth over the communication channel. This brings us to an important challenge regarding compression design and implementation: How can we achieve high enough performance to remove the communication bottleneck, while maintaining efficient compression, and also while minimizing accelerator resources utilization? During the exploration, we face several key questions:

- Which algorithm can achieve the necessary compression efficiency?
- Can this algorithm achieve high performance on the accelerator?
- Can this algorithm be implemented efficiently on the accelerator, leaving enough resources for the actual computation?
- If the algorithm cannot achieve high performance with low resources in its original form, how can it be modified for better performance and resource utilization?

In this dissertation, we explore the use of high-efficiency compression schemes to solve the above-mentioned issue by integrating one or multiple compression components into the accelerator platform. We focus mainly on FPGAs as the target accelerator due to their extremely high power efficiency and performance [104, 51], which not only makes them popular as stand-alone accelerators, but also opens up many new venues for systems architectural exploration, such as near-storage [67, 38, 68] and near-network [45, 137] accelerators. As a result, our compression algorithm explorations also focus on efficient implementation as application-specific hardware accelerators.

We demonstrate our approach using two important application case studies: Grid-based scientific computing, and high-dimensional nearest-neighbor search. The grid-based scientific computing application predominantly involves floating point data and operations, and we explore the use of highly efficient, error-controlled lossy compression algorithms for faster data movement of intermediate state. Algorithmic exploration was especially important for this application as general-purpose dictionary-based copression algorithms such as the LZ series of algorithms are notoriously ineffective at compression floating point data [82]. The nearest neighbor search application involves quickly scanning a database of sparsely encoded data elements, and we explore the use of column-wise Varint compression for faster access into the database.

For both approaches, we notice that the original compression algorithms are designed for efficient software implementation, and are not well-suited for effective hardware accelerator implementation. We noticed that in many cases, only a minimal amount of changes to the algorithm are necessary for more efficient hardware implementation. This dissertation presents the approaches we took to improve the hardware implementation efficiency of these algorithms, and demonstrate the benefits they have on the end-to-end application performance.

Given the same resources, the introduction of efficient compression results in impressive

performance improvement which goes beyond merely removing the communication bandwidth limitations. Not only does our compression-enabled accelerators vastly outperform accelerators that suffer from communication bandwidth limitations, they often significantly outperform even much costlier systems configurations with sufficient memory capacity for the entire working set of the application.

1.1 Case Study: Lossy Floating-Point Compression

The use of compression for scientific computing acceleration has traditionally been limited because of the high performance overhead of floating-point compression algorithms. General-purpose lossless compression schemes such as DEFLATE [33] and LZW [133] are typically not very efficient with floating point data, which often make up a large part of scientific datasets [82, 34]. On the other hand, floating-point specific lossy compression algorithms such as ZFP [82, 36] and SZ [34] 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.

Previous research have shown that use of these error-bound lossy compression schemes do not cause meaningful quality degradation of computation [10, 65], *even for iterative algorithms compressing intermediate data* [46]. However, these complex algorithms also have high performance overhead compared to lightweight data-oblivious compression algorithms like LZ4 or LZO, making their demonstrated performance insufficient to keep up with the internal computation capabilities of scientific computing accelerators.

One natural solution to this problem is to re-implement them on hardware. We choose ZFP as our target algorithm because most of the algorithm is easily parallelizable. This is in contrast to SZ [34], the other prominent lossy floating point compression algorithm, which

is organized around individual data prediction.

However, our evaluation on FPGA implementations of ZFP showed that a straightforward FPGA implementation of ZFP cannot bring significant compression/decompression throughput improvement over its software version due to the inherently serial characteristic of a single stage (*"embedded coding"*) in the algorithm. To address this problem, we made minor, hardware-aware changes to the algorithm for easier parallelization, resulting in higher-efficiency hardware implementations. We have created a family of hardware-optimized ZFP variant, which we call the ZFP-V family, and it includes ZFP-V1 (1D) and ZFP-V2 (2D) targeting 1D ZFP and 2D ZFP, respectively. The two designs solve the serial aspect of the original algorithm in different ways.

As a result, both ZFP-V implementations are capable of providing wire-speed compression and decompression of floating point data while using only a small fraction of on-chip resources. The new coding schemes do trade a small amount of compression while obtaining an order of magnitude performance improvement. However, this proved not to be an issue for us, as the goal of our hardware compressors is removing the host-accelerator communication bottleneck, which the compression efficiency of both ZFP-V designs more than achieve.

A high-level overview of the algorithmic modifications are provided below, and more details about the ZFP-V algorithm design, as well as the accelerator implementation and performance evaluations are given in Chapter 2.

1.1.1 Optimizing ZFP For Hardware Implementation

ZFP is a block-based algorithm, which processes input in d-dimensional (d = 1/2/3) blocks of size 4^d. During compression, the input floating-point data go through multiple stages of preprocessing and transformation and are turned into a list of decorrelated fixed-point numbers

where earlier values in the list have statistically larger values. This list is compactly encoded taking advantage of the statistically large number of zero bits near the most significant bits. The algorithm also takes advantage of the relaxed, *lossy*, compression requirement, only encoding a subset of the upper bits of the data until the required error control is satisfied.

During data encoding, the original ZFP algorithm adopts a technique called "group testing" to encode the transformed block, and this is the stage which is the most difficult to parallelize as-is. This approach repeats a simple process of emitting individual nonzero bits and then checking if the remainder of the value being encoded is now zero. Figure 1.2 uses a simple example to show how group testing is used to compress a bit plane that has 16 bits. For a given n-bit bit plane value, the algorithm scans bits one by one from the LSB (least significant bit) position until the remaining bits become all zeros. If the remaining bits are not all zero, a *test* bit "1" is emitted, then the LSB bits until a nonzero bit are emitted. This process is repeated until all remaining bits are emitted, after which the *done* bit "0" is emitted.

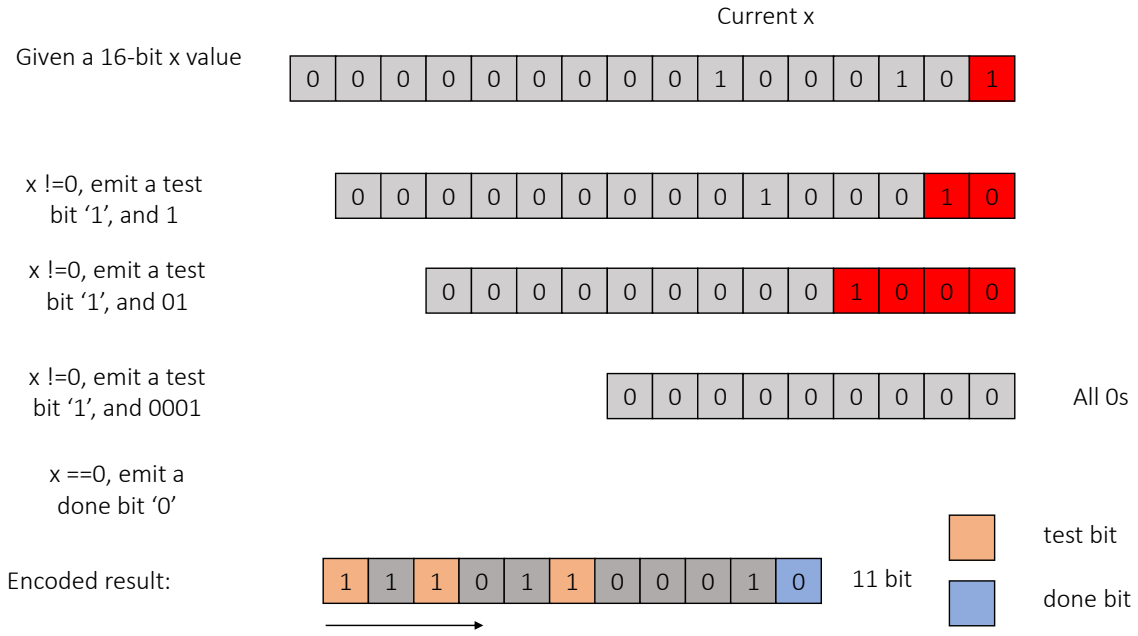


Figure 1.2: One example of Group Testing Encoding

Since the algorithm cannot know the remaining bits before the current sequence of nonzero bits are emitted, this portion of the algorithm is inherently sequential. As a result, in the

worst case only 1 bit can be processed per cycle when it is implemented on hardware, which gives a 31.25 MB/s speed based on a 250 MHz FPGA design. On the other hand, our scientific computing engine (e.g., stencil computing engine) can handle data ingestion at wire-speed, which is 8 GB/s on a 256-bit datapath clocked at 250 MHz. As a result, a single pipeline of the unmodified ZFP algorithm is insufficient to match the data ingestion speed of the computing engine.

We modify the embedded coding stage to use a header to specify the position of the most significant nonzero bit, instead of a sequence of test bits interleaved with data. This way, there is no internal dependency when processing a single data element, and a single data element can be either encoded or decoded within a single clock cycle. By doing so, we can achieve a processing speed of 16 bits/cycle (500 MB/s) on a 250 MHz design, which is a significant improvement over the original approach.

However, allocating a fixed-size header for each data element results in a significant degradation of compression efficiency. We address this problem using a variable-length header scheme, which is described in more detail in Chapter 2.

This design not only improves the performance of a single pipeline implementation, but also drastically reduces the chip resource utilization per unit bandwidth. As a result, we can afford to implement a multi-pipeline design to eventually achieve wire-speed processing, matching the data ingestion and emission bandwidth requirements of the computation engine.

1.2 Grid-Based Scientific Computing with Compression

With these high-efficiency hardware compressors, we are able to design and implement BurstZ+, a scientific computing accelerator platform that integrates one or multiple hardware compressors/decompressors and a high-speed scientific computation engine. It ad-

dresses the communication bandwidth issue of scientific computing accelerators by providing the computation engine with highly efficient compression engines.

BurstZ+ is able to sustain high performance regardless of the size of the working set relative to the on-board fast random access memory. If the intermediate data size exceeds the memory capacity, the overflow data is exported to either the memory or storage of the host server, in a compressed and randomly accessible format. The compressed data is fetched and decompressed piecemeal at the accelerator side only when required.

We evaluate the performance of BurstZ+ using three realistic grid-based scientific computing applications with widely differing characteristics. Grid-based scientific computing applications process data organized into a grid of cells, where the data type of each cell may be widely different for each application. A *kernel* or *stencil* is run on each cell, where the kernel can read values of neighboring cells within a fixed distance. Many important and difficult non-linear systems can be decomposed into kernels and stencils, which supports effective parallelization. Each kernel can have high or low *operational intensity* based on the amount of computation required by the algorithm per every byte of data I/O. We use the three following applications for evaluation:

- 3D Heat Dissipation: 3D stencil kernel with high memory bandwidth requirements but relatively low operational intensity.
- 2D Speckle Reducing Anisotropic Diffusion (SRAD): 2D stencil kernel with high operational intensity.
- 2D Fluid Dynamics via Lattice Boltzmann Method (LBM): 2D stencil kernel with high operational intensity, and also large cell sizes in bytes.

Using these applications, we demonstrate that the ZFP-V accelerators have both high performance enough to supply the accelerator cores with enough bandwidth, as well as achieves

high enough compression efficiency to close the bandwidth gap between the PCIe and on-board DRAM bandwidth. As a result, our compressor is efficient enough to remove not only the PCIe performance bottleneck, but also improve the effective performance of the on-board DRAM by storing compressed data even on the fast on-board DRAM, and decompressing on the fly. Evaluated against hand-optimized implementations of kernel accelerators without compression, our single-pipeline BurstZ+ prototype outperforms an accelerator without compression by almost 4×, and even an accelerator with enough memory for the entire dataset by over 2×.

1.3 Case Study: Integer Stream Compression

We also explore the difficult problem of high-dimensional nearest-neighbor search, and apply the approach of hardware-optimized compression algorithms to the application. High-dimensional nearest neighbor search suffers from the so-called *curse of dimensionality*, where the effectiveness of indexing algorithms quickly degrades for data types with high dimensions. As a result, search into such datasets cannot take advantage of efficient indexing, devolving into linear search. In such a situation, the performance of the search system depends on how fast the database to search can be streamed from storage to the processing engine.

For this application, we encode the high-dimensional data using typical sparse multi-column encoding, and apply integer stream compression to the individual columns. Data is stored in a compressed format, and decompressed on-the-fly and entered into the comparator engine. Each decompressor contains a configurable sequence of integer stream compression algorithm implementations, including a Delta Encoder, Run-Length Encoder (RLE) [103], and Group Varint Encoder [29]. Since each column shows different statistical distributions, the decompressor can be configured with a subset of provided compression algorithms to achieve maximum compression efficiency. For example, Delta encoding calculates and stores

the difference between two consecutive integers in a given integer list instead of the original integers. So it is beneficial for an *ascending* and *small difference* list.

Varint encodes an integer by just storing the bytes that contain non zero bytes instead of the fixed number of bytes, coupled with a small header describing the actual number of encoded bytes. So each 32-bit integer needs 2-bit header that denotes the number of bytes stored. Obviously, it is only useful if the data being compressed is composed of typically *small* integers. Group Varint is an optimized variant of Varint that groups headers of 4 32-bit integers together (called a prefix). As a result, the 4 headers in the prefix occupy one byte and can be read at once, which provides higher decoding speed than Varint by reducing the amount of dependencies between each encoded value. Figure 1.3 shows the encoding process.

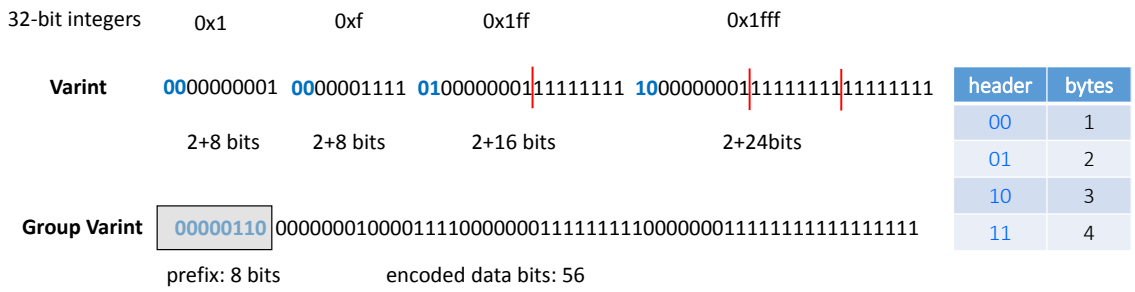


Figure 1.3: Varint and Group Varint Encoding

These three encoders can be combined as a chained structure to achieve excellent compression effect, such as a Delta-RLE-GroupVarint encoder or Delta-GroupVarint encoder depending on the needs and available hardware resources.

While the run-length compression and the delta compression algorithms can be effectively implemented for a necessary wide datapath in a straightforward manner, the Varint compression algorithm involves dependencies between data elements, limiting the performance and efficiency of the hardware implementation. For example, even with the Group Varint algorithm, at most 4 integers can be decoded at once, after which a multi-cycle shifting process is required to find the beginning of the next encoded set.

In order to achieve higher decoding speed we introduce a wider Group Varint algorithm where we group more encoded data into each header, allowing the algorithm to pre-calculate and shift the input stream while the data decompression process is still ongoing. With this minor adjustment, we demonstrate that the decompression engine can achieve wire-speed performance at a very low chip resource utilization.

We use this algorithm implementation to construct an accelerator for high-dimensional nearest neighbor search, ZipNN. Another characteristics of ZipNN is that it is implemented as a near-storage accelerator, enjoying the relatively higher *internal bandwidth* of the storage device, which is often 2× faster than the under-provisioned host-side PCIe bandwidth. Our system demonstrates using three column decompressors, that the near-storage accelerator is capable of supporting high enough decompressed bandwidth to saturate the comparator engine, achieving over 6× the performance compared to the same accelerator without compression, and over 2× the performance of multi-threaded software running on much costlier systems with enough fast random-access memory capacity to store the entire dataset.

1.4 Dissertation Contributions and Organization

This dissertation explores the use of hardware-optimized compression to solve the host-accelerator communication performance issue. The claimed contributions are listed as follows:

1. Design and implementation of high-efficiency hardware accelerated floating point compression algorithms (ZFP-V series).

2. Demonstrate a bandwidth-efficient scientific computation platform BurstZ+ that completely removes the host-accelerator communication bottleneck using ZFP-V.

3. Demonstrate ZipNN, which designs and implements a hardware-optimized library of heterogeneous integer compressors/decompressors to accelerate high-dimensional similarity search.

The rest of this dissertation is organized as follows:

Chapter 2 and 3 present how our ZFP-V algorithms solve the performance issue of hardware ZFP implementations using hardware-optimized design changes, and its application to grid-based scientific computation. Chapter 2 presents a detailed breakdown of inefficiencies of the original ZFP algorithm and how it can be improved with a minor algorithmic change, as well as the performance and resource utilization analysis of the hardware implementation. Detailed evaluation against software ZFP and unmodified hardware ZFP are conducted. Chapter 3 presents the BurstZ+ framework and evaluates its performance by three scientific computing accelerators.

Chapter 4 explores hardware-optimizing a high-throughput integer stream compression algorithm, and applies it to the high-dimensional similarity search application. The resulting system is called ZipNN, which uses a near-storage accelerator configuration of a heterogeneous library of hardware-optimized compression algorithms for integer streams, in order to accelerate high-dimensional similarity search.

Chapter 5 concludes this dissertation and discusses some potential future research work.

Chapter 2

ZFP-V: Hardware-Optimized Lossy Floating Point Compression

2.1 Introduction to Compression

Traditionally, compression is an effective method to reduce the storage cost and the overhead of data movement. As the amount of data processed by many important applications exceed the DRAM capacity of reasonable single-node machines, the limiting factor of processing capacity has become the data transfer speed between machines in a cluster, as well as the capacity and transfer speed of secondary storage devices.

Compression reduces cost and improves performance of computing by reducing the amount of data that must be stored and transported. Therefore, high-performance data compression algorithms are some of the critical components of high-performance computing.

2.1.1 Lossless Compression and Lossy Compression

Generally, compression algorithms can be categorized into two types: lossless compression that allows the original data to be reconstructed without any loss from the compressed data, and lossy compression that implements the compression by discarding partial original data. With lossy compression, there always exist an error between the decompressed data and the original one.

Common lossless algorithms include dictionary-based algorithms including the LZ series (e.g., LZ77/78, LZW, LZO, LZ4, etc.) and others, such as huffman coding, deflate, snappy, gzip, and so on. These algorithms find repetitions in the data stream and encode them using a shorter code. These algorithms are typically data-agnostic, and can provide high compression efficiency in a wide range of data types spanning from text to binary data. There are data type-specific algorithms as well, such as delta encoding, run-length encoding (RLE), Varint and Group Varint encoding, which can effectively compress integer numbers with specific distribution patterns. These algorithms typically do not provide as high compression ratios as the more complex dictionary methods, but are typically much simpler to implement, and provide very high performance.

On the other hand, lossy compression tolerates a certain amount of noise during the compression and decompression process, and are used for data types that can tolerate such noice, such as images, video, and sound. There are a lot of related industry standards, such as JPEG, MPEG, H.264/265, Xvid, and VP8. These algorithms can either be configured to achieve a certain compression ratio regardless of the introduced error, or to maintain noise below a certain level. If noise can be maintained below a user-defined level, the algorithm is considered *error-controlled*.

Another interesting area for lossy compression is floating point compression. Due to the high entropy involved in the floating point representation, they are notoriously difficult to

compress. For example, gzip and other lossless compression methods typically achieve no compression on floating point data [121]. Floating point numbers are often used in the scientific computing and hence their compression is the focus of this dissertation.

2.1.2 Floating Point Compression

Traditional data-oblivious compression algorithms (based on Huffman coding, LZ77/78, and other methods) are typically unable to achieve high compression ratios on scientific data, which are often composed mainly of floating point (FP) values. Floating point data introduces a large amount of variation in the byte-stream which makes it difficult for data-oblivious algorithms to compress efficiently.

Recently proposed lossy floating-point compression algorithms take advantage of how a small amount of error is tolerable with floating-point operations, and provide very high compression ratios while ensuring data retrieval with errors within a user-specified margin.

Many lossy floating-point compression schemes have been researched, including ZFP [83, 37], SZ [35, 124, 81], FPZIP [84], ISABELLA [75], SSEM [115], and others, each with strengths and weaknesses.

These algorithms specifically targeting multi-dimensional floating point data, and allow the user to make a trade-off between compression ratio and error margin. They typically achieve an order of magnitude better compression ratios compared to data-oblivious algorithms like gzip, while controlling error below an acceptable margin for many scientific computing applications. Such highly efficient compression algorithms have improved the performance of high-performance computing while lowering the cost, by reducing storage and network requirements [11, 101, 57].

Of the many proposed algorithms, ZFP, proposed by Lindstrom et al.[83], and SZ, pro-

posed by Di et al.[35] are two most commonly used algorithms in the scientific computing community [1, 88, 123, 101], with different strengths and weaknesses on different data types.

These two are based on different principles: the former is based on block transformation and the latter is based on value prediction. For diverse scientific datasets, they play a complementary role. Tao et al. [125] points out that for some dataset ZFP is better, for others SZ is better.

2.1.3 Why Accelerate Floating-Point Compression Algorithms?

While these floating-point compression algorithms have been designed with performance in mind, running them on general-purpose processors does impose a limitation in performance. Our profiling of ZFP shows that neither a single thread compressor or decompressor on a high-end Intel Xeon processor can achieve 1 GB/s of throughput even for favorably compressed data, often processing 200 MB or less per second. With SZ, according to benchmarks, a single-thread software version of SZ on a high-end Intel Xeon was able to process data at less than 100 MB/s [136]. Considering the multi-GB/s bandwidth of modern network and storage devices, software implementations of these algorithms will not be able to sustain the high throughput of modern system infrastructure.

Naturally, hardware-accelerated implementations would be desirable. There are not many prior work on floating-point compression algorithms for scientific data. GhostSZ [136] is a state-of-the-art FPGA implementation of SZ on the Arria-10 FPGA platform, and has demonstrated single-pipeline throughput of over 800 MB/s. To the best of our knowledge, there does not yet exist an FPGA implementation of ZFP. One system, TerseCades [101], uses ZFP in some of its pipelines, but it does not use a hardware version of the algorithm.

In this dissertation, We select ZFP due to its high efficiency on the datasets of our interest. In addition, most components of ZFP were readily parallelizable numerical operations.

2.1.4 Straightforward Hardware Implementation is not Enough!

While a hardware accelerator is a natural way to achieve high throughput while freeing up the CPU cores for useful work, a straightforward implementation of the ZFP algorithm is unable to achieve very high performance due to the inherently serial nature of some of its sub-components. Specifically, the "group testing" step is a completely serial process which, in the worst case, can emit a single bit per cycle.

To overcome the above-mentioned weaknesses, this paper presents ZFP-V, a hardware-optimized error-controlled lossy floating-point algorithm which modifies the ZFP algorithm to achieve higher performance on hardware accelerators. ZFP-V replaces the serial group testing step with a new algorithm using variable-length headers, which allows the accelerator to invariably handle a bit plane per cycle.

In the following sections, we will first introduce the basic principle of ZFP algorithm, then introduce the design and implementation of ZFP-V, and finally evaluate the performance of ZFP-V.

2.2 ZFP Compression Algorithm

ZFP is a lossy floating-point (both single- and double-precision) data compression scheme and was designed specifically for the high-precision numerical data used in scientific computing. It provides multiple compression modes including bit rate, error tolerance or precision control, making it very flexible. In our work, we use the most recent version 0.5.4 as the starting

point of our development.

ZFP applies compression/decompression on a block level, where it divides a 1D, 2D or 3D array into a series of small, fixed-size blocks which are processed independently. Each block consists of 4^d values, where d is the dimension of the block. For example, a 3D block contains 4x4x4 values and a 2D block contains 4x4 values. The choice of block dimensions is up to the user, trading performance for better compression with higher dimensions. Block processing allows efficient compression at an arbitrary point in data, and also can achieve a very high throughput because multiple blocks can be processed in parallel.

For each block, compression is performed in four steps: (1) **Fixed-point conversion** via block-floating point representation [92], (2) **Block transform** to de-correlate the integers, (3) **Sequency ordering** to roughly order the transformed values according to their magnitude, and (4) **Embedded coding** to compactly encode data in the order of significance.

The rest of this section describes each step in more detail, and describes how the embedded coding step design causes a performance bottleneck in a hardware implementation.

2.2.1 Fixed-Point Conversion

In this step, each floating-point value in a block is converted to a block-floating point (BFP)[132] representation, which is a floating-point representation where all values share the same exponent. By calculating the largest exponent in the block and expressing all values with respect to it, each value in the block is represented with a signed integer.

2.2.2 Block Transform

ZFP uses a 4^d orthogonal linear transformation that is similar to the discrete cosine transform (DCT) used in JPEG and other algorithms, to decorrelate the values in a block. Decorrelation results in many near-zero coefficients, which can be compressed efficiently. The unique structure of the linear transformation used by ZFP allows it to use the *"lifting scheme"*, which factorizes the transformation into a series of simple filters, greatly simplifying its computation.

2.2.3 Sequency Ordering

Next, the values in the block are reordered by its *"sequency"*, which is the sum of the indices of each value in each of its dimensions, i.e., by $x + y + z$ for a 3D matrix. The ZFP authors show that after the block transform, the sequency ordering roughly corresponds to the magnitude ordering of the data, without having to perform a costly sort. The nearly sorted characteristic of sequency ordered data makes the embedded coding scheme highly efficient in terms of compression ratio.

2.2.4 Embedded Coding

In this step, ZFP applies embedded coding to compactly encode the current block and write the compressed bits to the output bit stream. There are many embedded coding techniques, such as zero-tree coding [24], set partitioning [109], etc. ZFP adopts a modified version of a technique called *"group testing"* to encode multiple bit planes.

Embedded Coding Using Group Testing

Put simply, group testing efficiently encodes a word by repeatedly testing if the remainder

of the word is zero, and only encoding it if there are nonzero bits present. As a result, group compression generally demonstrates good compression performance when there are enough zeros in the data. The group testing algorithm encoding a word x can be described via the following pseudocode:

```
while x != 0 do
    Emit 1 ;                                   ▷ Emit test bit (x not zero)
    repeat
        b = x[0] ;
        Right shift x by 1 ;
        Emit b ;                                   ▷ Emit bits until a 1
    until b == 1;
end
Emit 0 ;                                                   ▷ Emit done bit
```

Algorithm 1: Group testing algorithm used by ZFP to encode a word

The algorithm repeatedly checks if x is zero, and terminates with a done bit **0** if so. If not, it emits a *test bit* **1**, and emits bits one-by-one until a **1** is emitted, and repeats the process.

Given a 16-bit x value of "0000000001000101" in binary, x can be encoded as $\underline{1}1\underline{1}01\underline{1}0001\underline{0}$, which requires only 11 bits. The underlined bits are test bits. It should be noted that this particular value of x can be efficiently compressed because it contains many leading zeros. As a counter example, if x is, say, "0000100001000101", which has a small number of leading zeros, the encoded result would be $\underline{1}1\underline{1}01\underline{1}0001\underline{1}0000010$. The encoding requires 18 bits, which is more than the uncompressed bit width of 16.

Bit Plane Encoding Using Group Testing

We use a simple example to show how group testing is used to compress a block of data. Fig. 2.1 shows the layout of a 2D block after the sequency ordering step. Here we use the double data type, so each value has a 64 bit width. The k^{th} bit of all values form a **bit plane** (0^{th} bit plane is marked by a red ellipse). With a 4x4 block, we have 16 values. So each bit planes contains 16 bits, and we have 64 bit planes. ZFP encodes bit planes one by one from the most significant bit plane to the least significant.

Recall that ZFP allows users to specify the error tolerance and precision control, not all bit planes are encoded here. According to the parameters provided by users, ZFP first calculates a bit plane lower bound called *"kmin"*, which means only the bit planes from 63 to kmin will be encoded. The greyed bits in Fig. 2.1 represents the bits to be encoded. The rest are simply ignored.

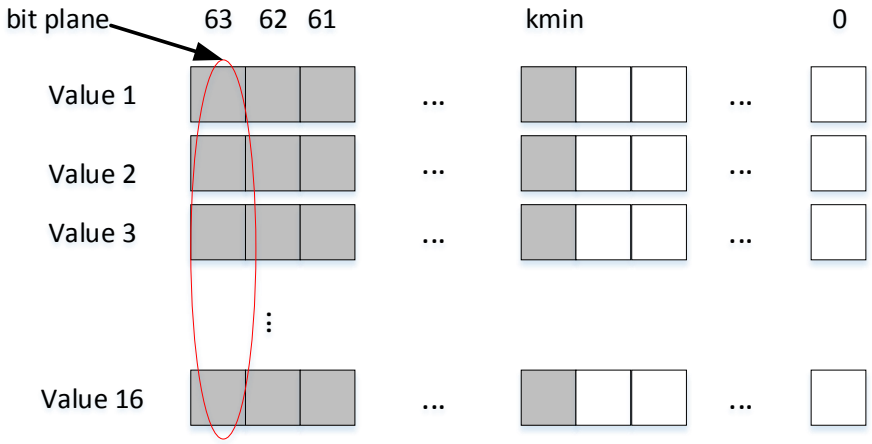


Figure 2.1: The bits distribution of a 2D block with double type

ZFP takes advantage of the data distribution of sequency ordering to further reduce data size. ZFP encodes parts of each bit plane verbatim before applying group testing, and the number of bits emitted verbatim is the MSB index of the previous bitplane. Because the sequency ordered data is roughly ordered by size, this scheme prevents group testing from being inefficiently applied to data with many nonzero bits. In other words, having roughly sorted numbers thanks to sequency ordering can result in an early exit per bit plane, after emitting all the nonzero least significant bits.

2.2.5 Inefficiency of Group Testing For Hardware

While group testing can result in efficient encoding, it has a high performance overhead in a hardware implementation, especially on a reconfigurable platform such as an FPGA with lower clock speed compared to ASICs.

Because each loop iteration in Algorithm 1 depends on the results of its previous iteration, and because each iteration emits only one bit of data, a straightforward hardware implementation can only emit one bit per cycle and in the worst case may require 4^d cycles to emit a single bit-plane.

While this problem can be somewhat mitigated during compression by parallelizing each bit-plane encoding, such an approach is less feasible for decompression. 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.

In this dissertation, we have designed and implemented multiple algorithms based on ZFP, which result in more efficient hardware implementations on reconfigurable fabric. They not only improve a single bit plane's compression and decompression performance, but also achieve parallelism across multiple bit planes by replacing the group testing-based encoding algorithm with a header-based encoding scheme. The collection of our proposed algorithms is called ZFP-V. ZFP-V consists of 1D and 2D variants of ZFP-V, which we will denote as ZFP-V1 and ZFP-V2. Different optimization methods were employed for the two variants due to the differences in data distribution.

We describe these optimizations and their effects in the next sections.

2.3 ZFP-V Design and Implementation

In this section we describe the design and optimizations of our ZFP-V compression step by step. The first three steps in ZFP can be easily parallelized to support deterministic wire-speed operation without major design changes when implementing it on FPGA. For the last step "group testing" which is hard to parallelize, we introduce a novel and high-efficient algorithm to accelerate it.

We first describe a simple algorithm that uses a fixed-length header to improve performance, but loses compression efficiency. Then we provide a sequence of further optimization approaches we employed to both reclaim compression efficiency and also improve performance. Our optimizations include the variable-length header as well as the multi-level header approach for the 2D algorithm, and the coarse-grained header for the 1D algorithm. We also describe and compare the actual compressed bitstreams generated by these algorithms, as well as describe coarser-grained parallelism approaches using padded chunks of compressed data.

2.3.1 Fixed-Length Header-Based Encoding

As can be seen from last section, group testing can only process one bit at a time in the worst case, and therefore is often a completely serial process. As a result, the group testing step was usually the biggest performance bottleneck of our optimized ZFP implementation efforts.

By analyzing group testing encoding, we found that the low efficiency is from the fact that the test bits are mixed with the data bits. And because of the dependency relation, each bit plane has to be scanned serially bit-by-bit until a "1" is seen.

To expose more parallelism, we decided to use a header to store the number of data bits to be written for each bit plane. Algorithm 2 describes the encoding algorithm using headers. Because encoding each bit plane no longer requires an irregular loop, each bit plane can now be processed at once.

One weakness of a fixed-length header-based algorithm is that it could increase the size of compressed bit stream due to **the fixed overhead of the header bits**, reducing the compression ratio. Given a favorable example word "0000000001000101", we can write a

```
for k = 64 to kmin do
    bitplane = extract_bitplane(block, k) ;
    msb = msb_index(bitplane) ;
    emit_bits(msb, sizeof(header)) ;
    emit_bits(bitplane, msb+1) ;
end
```

Algorithm 2: Encoding a block using a header-based algorithm

header "7" and "1000101" to the bit stream. Since a bit plane contains 16 bits for a 2D block, a naive header needs at most 4 bits. As a result, we need $4 + 7 = 11$ bits, which is equal to the number of the result generated by the group testing. However, if the bit plane is a string with many leading zeros, e.g., "0000000000001001", group testing just needs 6 bits to encode it, but the new algorithm needs $4 + 4 = 8$ bits. The header consumes 4 bits which are too much for this string.

As an improvement, we can partition this string into 8 substrings with each one containing 2 bits, like "00|00|00|00|00|00|00|01". In this scheme, we just need 3 bit to store the number of nonzero substrings. In this case, we store the data bits in substring units. As a result, we need $3 + 4 = 7$ bits and the compressed bits are "0101001". The underlined bits are the header and represent 2 2-bit substrings that followed. Similarly, we can partition the string into 4 4-bit substrings, then we just need 2 bits to store the header. However, our experiments show it is not better than the 3-bit substring. If a substring is "0001", for example, we will waste 3 more bits than the 4-bit header scheme, and 2 more bits than the 3-bit scheme.

We call these schemes with fixed header width *fixed length header (FLH) algorithms*. Though they can simplify the original group testing algorithm and increase the parallelism, they cause the compression ratio to drop significantly. In next section, we will introduce a *variable length header (VLH) algorithm* that can achieve as much bit-level parallelism as the FLH algorithms while causing negligible compression ratio loss.

2.3.2 Variable Length Header (VLH) for ZFP-V2

We showed that fixed-length headers can cause much extra bit overhead by using a header for each bit plane. The biggest problem in this is mostly from adding a fixed overhead to bitplanes that consist entirely of zeros, which are not only very common, but also can be very efficiently compressed in the original algorithm. Actually, we observed that for many benchmarks, both real-world and synthetic, the distribution of MSB indices were skewed towards either lower or higher indices, without many values in the middle[1]. In order to solve the issue caused by FLH, based on this observation, we partition the binary string of a bit plane into 5 sets of exponentially increasing width, and assign each of them a unique code word as its header. As a result, different sets can have different header lengths. This is called variable length header (VLH) scheme. Fig. 2.2 shows the 5 sets with different colors, and Table 2.1 shows the variable length header coding scheme. For example, if the word has no nonzero bits, we use "0" to represent the header and write only one header bit and one data bit to the stream; if msb is 1, we use "100" to represent the header and write three header bits and two data bits to the stream; if the msb is 2 or 3, we use "101" to encode the header and write three header bits and 4 data bits to the stream. The reason encoding is skewed in this way is because we wanted to improve the common case where a bit plane has zero or only one nonzero bit.

This encoding efficiently handles bit planes with one or less nonzero bits using two bits, and also requires only a three-bit header in all other cases. Though VLH needs a little bit more logic to encode or decode the header before processing the data bits than FLH, both of them can process one bit plane at a time, which provides much better performance than the original group testing algorithm.

[1]For the eight real-world datasets we evaluated in Section 2.5, 80% of all encoded bitplanes had MSB indices that were either less than 1, or more than 11. (46% less than 1, 34% more than 11)

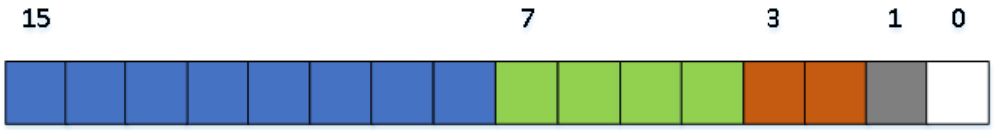


Figure 2.2: The MSB distribution of the binary string in a bit plane

Table 2.1: Header Coding of the bit plane

MSB	Code word	#header_bits	#data_bits
0	0	1	1
1	100	3	2
2 - 3	101	3	4
4 - 7	110	3	8
8 - 15	111	3	16

2.3.3 2-Layer Header for ZFP-V2

Though VLH effectively improves 2D ZFP's encoding/decoding performance, the introduction of a variable-length header introduces dependency between the offsets of each encoded bit-plane, limiting parallelization and potentially harming performance. For example, while each bit plane can be invariably decoded in a single clock cycle, we cannot start working on the next bit plane at the same time because we cannot start decoding the next header before we know the encoded length of the header and data bits in the previous bit plane. Since a bit plane is only 16 bits for 2D, this approach as-is cannot achieve our performance goals.

In order to overcome this limitation caused by inter-bit plane dependency while maintaining high compression efficiency, we implement a 2-layer header structure. This is a new algorithmic improvement over the originally published 2D ZFP-V algorithm [121].

We observe from Table 2.1, only the bit planes whose MSB is 0 need a 1-bit header and all others need a 3-bit header. So we extract the first bit of all original headers to group them to a "header-of-headers" called "header level 1", and then group the rest of the original headers to a "header level 2". Since at most 64 bit planes need to be encoded or decoded, header level 1 encodes at most 64 bits, and header level 2 encodes at most 192 bits. The number

of encoded bit planes are also encoded in the compressed format. This is described in more detail in Section 2.3.5. When decompressing, we can conduct completely parallel bit-level operations on header level 1 to get each bit plane's header size, and hence the size of all headers. We can then immediately read all header level 2 headers to reconstruct the headers of all bit planes, within one cycle. With the decoded headers, we can calculate the size of each compressed bit plane in the next cycle, and decide how many bit planes to decompress in parallel. While theoretically this approach can decode all 64 bit planes in parallel in a single cycle, we have decided on a narrower datapath to avoid the high resource utilization and routing difficulty of very wide datapaths. Our current implementation decompresses 8 bit planes (128 bits) per cycle. On our prototype running at 250 MHz clock speed, this guarantees a 4 GB/s of decompression bandwidth lower bound per pipeline.

2.3.4 Coarse-Grained VLH for ZFP-V1

VLH is used in ZFP-V2 and effectively improves its throughput. However, this approach would not be very efficient for the 1-dimensional algorithm ZFP-V1 because of the following observations: First, putting a header per bit plane is too expensive, because each block in the 1D ZFP only has four values, resulting in only four bits per bit plane. Second, after block transform and sequency ordering, the first 64-bit element of the four has a very high index of the first nonzero MSB bit.

In order to address the first issue of header overhead, ZFP-V1 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 described below would have nonzero bits

in upper bits because of the first element. To solve this second issue, ZFP-V1 treats the first element specially, and encodes it first before encoding other elements. Only the remaining three elements, which often have many leading zeros, are encoded using the coarse-grained header.

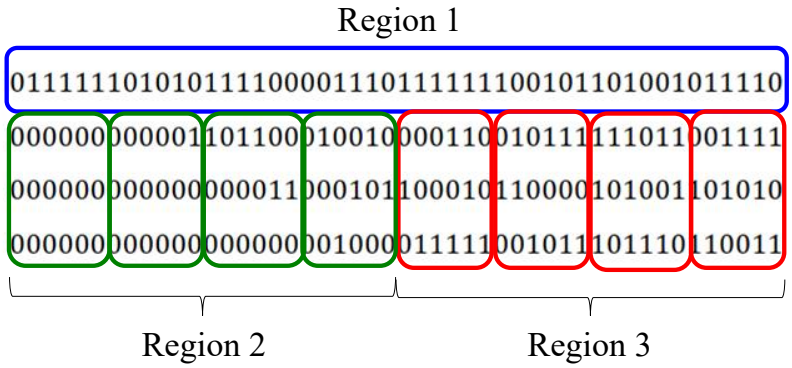


Figure 2.3: Three different encoding schemes are used for three different regions (Blue, green, red)

Figure 2.3 shows the resulting different regions of the four-element unit with different encoding methods, using an example block after sequency ordering. The first, top-most element 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 (region 2 and 3), which are in turn divided into four sub-groups. Each group is assigned a two-bit header, representing how many sub-groups need to be encoded. If the desired error margin is achieved within the first group (region 2), encoding can stop after encoding the first element (region 1) and valid sub-groups of the green group. If the error margin requires more bit planes to be encoded, region 3 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 2.3. 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 ZFP-V1 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 ZFP-V1 to achieve very high throughput with very small on-chip resources.

2.3.5 Bitstream Structure

Figure 2.4 shows what the encoded block looks like for the original ZFP, ZFP-V1 and ZFP-V2, respectively. For all methods, the first two encoded elements are a 1-bit zero-block flag bit, and an 8 or 11 bit *emax* field. The zero-block flag denotes whether all elements in this block are zeros or not. If this block is a zero block, ZFP just simply writes a '0' to the bitstream and processes the next block without any more action. If it is not a zero block, it stores an 8 bit (single precision floating point) or 11 bit (double precision floating point) *emax* value that represents the maximum exponents among all the floating point values in this block, encoded with an offset similarly to the floating point encoding. This *emax* value is used to recover the number of bit planes encoded, by comparing it against the error margin requested during compression.

The encoding scheme for the three approaches begin to differ after this point. For the original ZFP, it then proceeds to encode bit planes one by one, where each bit plane is encoded in a 1 to 31-bit variable-length format with mixed flag and data bits. On the other hand, ZFP-V1 first encodes the first element of the 4-element block (region 1), and then one or both of regions 2 and 3, each coupled with a 2-bit header. ZFP-V2 encodes the level 1 and 2 headers, and then the 1 to 64 encoded bit planes in sequence.

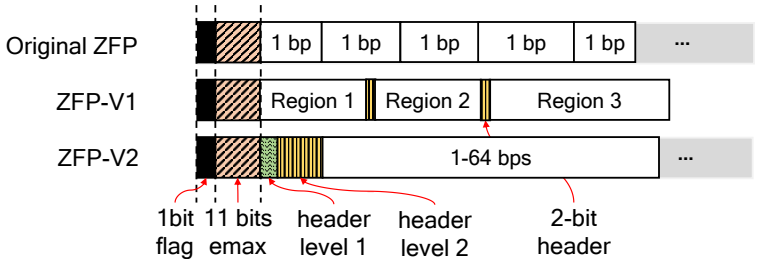


Figure 2.4: The encoded block layout of the original ZFP, ZFP-V1 and ZFP-V2

2.3.6 Independent Aligned Chunks

Despite the increased performance thanks to the header-based encoding scheme, a single-pipeline performance of ZFP-V is likely still not enough for very fast PCIe or memory. While each pipeline can process a steady state 4 to 8 GB/s of uncompressed data, due to the high compression efficiency of ZFP-V the compressed data rate is likely not high enough to make full use of the PCIe or memory bandwidth. ZFP-V solves this issue by organizing compressed data into independent, aligned chunks, enabling straightforward parallelism across multiple pipelines even for a single stream of data. In our prototype implementation of BurstZ+, ZFP-V 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 ZFP-V core to achieve high enough performance to easily saturate even on-board DRAM performance.

2.4 ZFP-V Accelerator Architecture

In order for compression to be useful, it must be able to keep up with the bandwidth of the memory and computation engine. Ideally, it should achieve wire-speed, meaning it adds no performance overhead regardless of how fast the other components become. ZFP-V1 and

ZFP-V2 employ different methods to achieve wire-speed, due to the differences in how they balance compression and performance.

2.4.1 ZFP-V1 Decompression Accelerator

For decompressing ZFP-V1, the major performance bottleneck is the decoding stage, as described in Section 2.3. 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.

Taking advantage of this insight, our implementation first replicates the decoder modules to achieve wire-speed per pipeline, before starting to replicate the entire pipeline. The internal architecture of a decompressor accelerator can be seen in Figure 2.5. The input stream is broken into 6 KB 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.

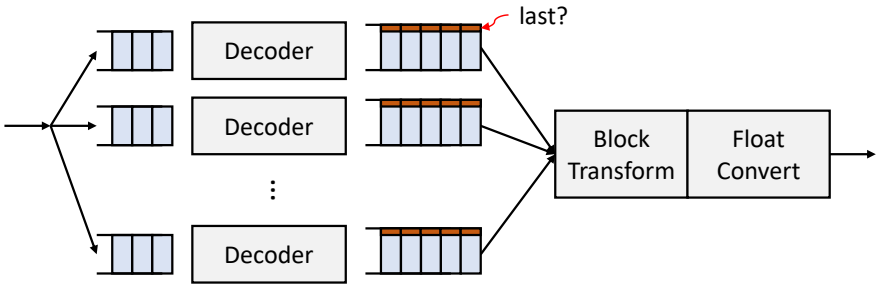


Figure 2.5: A multi-pipeline ZFP-V decompressor accelerator

Because we cannot predict how much uncompressed data will be generated from a *chunk*-sized compressed 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.

2.4.2 ZFP-V1 Compression Accelerator

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 the size of each uncompressed 4-element block is constant, each can simply be round-robin distributed to each encoder without having to wait until each is encoded. The encoder array does not need to work in terms of aligned chunks, but with individual elements.

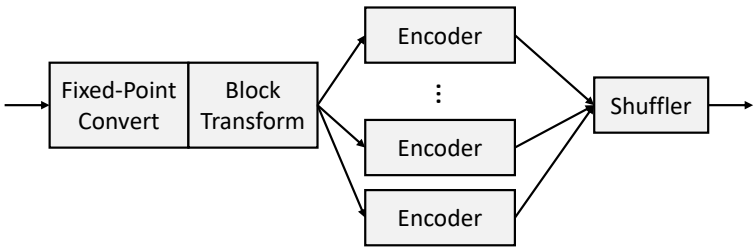


Figure 2.6: A multi-pipeline ZFP-V compressor accelerator

Figure 2.6 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 the same round-robin way by the shuffler, which bit-packs the compressed blocks, as well as handles chunk-alignment via padding.

2.4.3 ZFP-V2 Decompression Accelerator

By dividing the header into a level 1 and a level 2 header, the decoding module of ZFP-V2 can achieve wire-speed even with a single pipeline. The decompressor cascades header

parsing, as well as the multi-cycle variable-length shift operations necessary for encoded bit plane extraction. As a result, it can always keep the output datapath full.

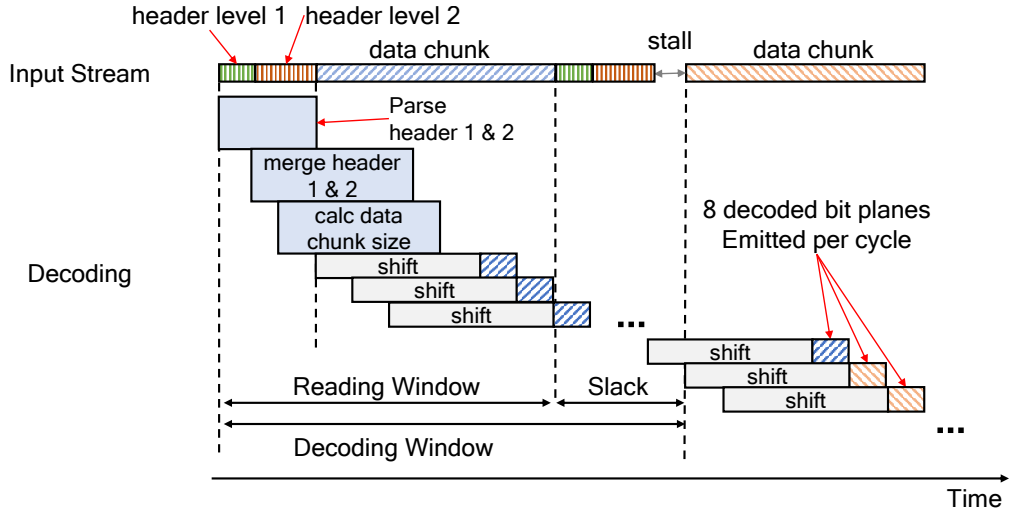


Figure 2.7: The decoding module of ZFP-V2 cascades multiple block decoding to achieve wire-speed output

Figure 2.7 shows the cascading decode process over two compressed blocks. Since the bit offsets of the compressed bit planes are all pre-calculated at the beginning of block decoding, each cascading shift operation can trivially extract multiple encoded bit planes at once, instead of having to wait until one bit plane is decoded in order to know the offset of the next one. In our current implementation, each shift operation handles a group of 8 encoded bit planes at once, which will transform into 128 bits of decoded data every cycle. If the cycle count required to emit the decoded data is larger than the cycle count required to ingest the encoded data (*slack* is larger than header parsing latency), the input pipeline may even be stalled.

2.4.4 ZFP-V2 Compression Accelerator

Compression of ZFP-V2 is trivially parallelized similarly to ZFP-V1. In our current implementation, the compressor module supports wire-speed performance on a 256-bit datapath.

2.5 Evaluation of Compression Efficiency

In this section, we first present the change of compression efficiency introduced by our hardware-optimization approaches to ZFP-V1 and ZFP-V2.

2.5.1 Compression Efficiency with with Fixed Error Bound

We first present the compression efficiency impact of ZFP-V2 compared to unmodified ZFP, with a fixed error bound.

Benchmark Datasets

We use benchmarks from the "Scientific Data Reduction Benchmarks"[2], which is a real world dataset collection and covers various scientific areas. We exclude two small size datasets that are less than 100 MB and one that is of 16-bit floating point precision. ZFP supports both 32-bit and 64-bit precision. Finally, eight benchmarks (CESM-ATM, HURRICANE, HACC, NYX, NYChem, SCALE-LETKF, QMCPACK, and Brown Samples) are used to evaluate ZFP and ZFP-V2. We use the fixed-accuracy mode because this is the mode the ZFP's authors recommend [86]. In this mode, we use 1E-3 as the error bound for all input dataset. This error bound is used in the examples provided by ZFP. Usually the higher the error bound, the lower the compression ratio. More details can be found in[83, 37].

Compression Efficiency Against Unmodified ZFP

The compression ratio is calculated as the ratio between the original data size and the compressed data size. We compressed all individual data files in each dataset, and calculated the arithmetic mean (average) of the compression ratios of all files in each datasaet. Fig. 2.8

shows the compression ratios of the original ZFP, ZFP-V2, and GZIP, for all datasets tested. For easier comprehension, we ordered the data points according to the compression ratio achieved by the original ZFP. As a result, the results for ZFP displays a straight line of y=x. As we can see, ZFP-V2 has a very close compression ratio compared to ZFP, with negligible loss. For comparison, GZIP, as an integer-based lossless compression algorithm, does not compress floating point data very well.

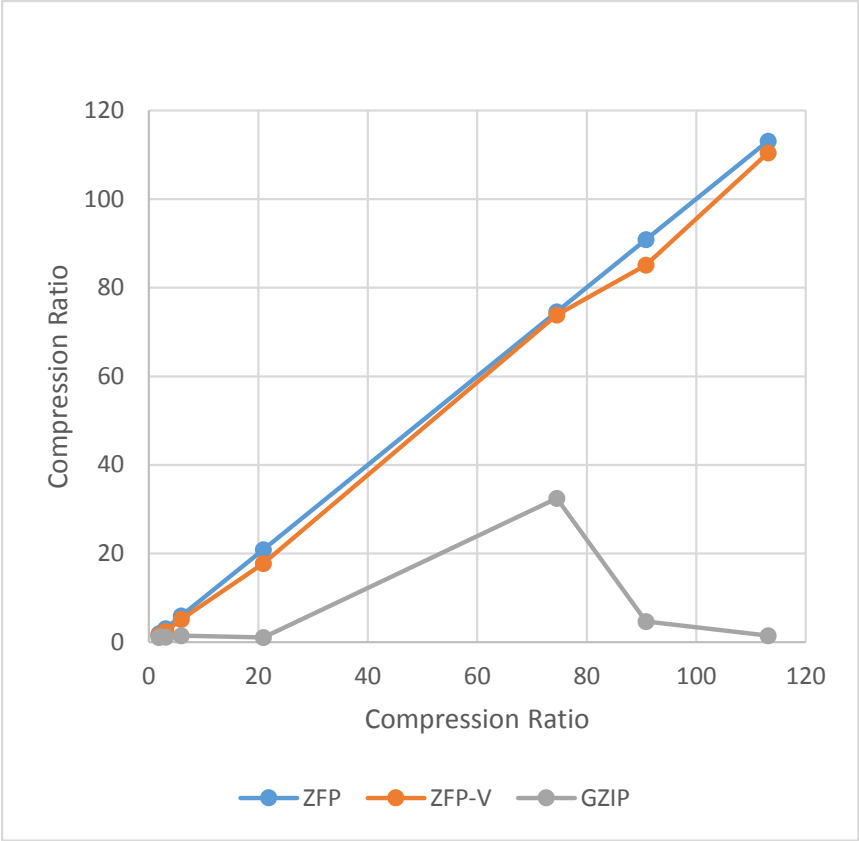


Figure 2.8: The Compression ratio distribution of ZFP, ZFP-V2, and GZIP

2.5.2 Compression Efficiency with Changing Error Bounds

As the error bound of ZFP may change according to application, we also demonstrate the impact of error bounds on the compression effectiveness. In this section, we only evaluate against a subset of the SDR benchmarks targeted for iterative grid-based scientific computing

applications, which is our ultimate target for the ZFP-V accelerators.

Figure 2.9 shows the efficiency of the original ZFP, ZFP-V1 and ZFP-V2 compression algorithms across benchmark datasets and error bounds. Each colored bar represents a compression algorithm configuration. For example, 1D3 is ZFP-V1 with 1-dimensional blocks, used with an error bound of 1E-3, and 2D6 is ZFP-V2 with 2-dimensional blocks, used with a much stricter error bound of 1E-6. Orig-2Dx represents the original ZFP with 2-dimensional blocks. We also provide comparison against gzip (the first bar in each benchmark).

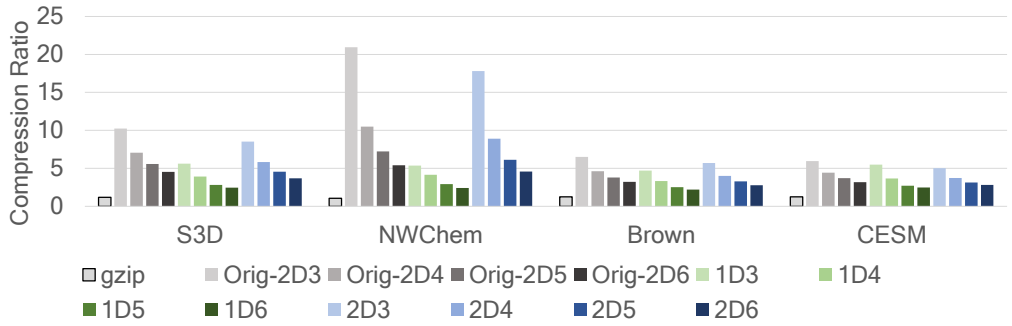


Figure 2.9: Compression efficiency of ZFP-V, across four datasets, with varying error bounds

First of all, gzip performs badly with floating point numbers, achieving less than 2x compression across all datasets. The original 2D ZFP achieves the best compression efficiency among the four algorithms tested, but 2D ZFP-V2 also consistently performs very closely. As we will show later, this small loss of compression efficiency is a worthwhile trade-off considering the order-of-magnitude superior throughput of the ZFP-V2 accelerators compared to the best effort ZFP accelerators. This is especially true considering the goal of ZFP-V is not to achieve the best possible compression for archiving purposes. Its goal is to achieve high enough compression at high enough throughput to close the performance gap between PCIe and memory. The compression efficiency needs only to be hide enough for that purpose. Similarly, while ZFP-V1 demonstrates worse compression than ZFP or ZFP-V2, it does still consistently provide $3\times - 4\times$ compression, which is often sufficient to remove the host-side communication bottleneck, at a much lower chip resource utilization. In Section 3.5.3, we will show that these compression efficiency numbers are sufficient to remove the communication

bottleneck.

2.5.3 Stability of Accuracy

	S3D	NWChem	Brown	CESM
2D3	*19	*5	27	*20
2D4	23	*8	31	24
2D5	26	*11	34	27
2D6	29	*14	37	30

Table 2.2: Number of bit planes encoded on average

Table 2.2 shows the average number of bit planes encoded during ZFP-V2 compression. Existing research has shown that when more than 24 bit planes are encoded for ZFP, the error from lossy compression accumulated over iterative algorithm execution is actually less than the error caused by the accuracy limitations of double-precision floating point [46].

We can see from Table 2.2 that many of the configurations we evaluated will actually result in reliable accuracy even while compressing intermediate data. We also include some cases where that is not the case, to evaluate the performance impact in such situations. Configurations with less than 24 bit planes encoded are prefixed with an asterisk (*).

2.6 Evaluation of Accelerator Performance

In this section, we evaluate the performance impact of our ZFP-V algorithm optimizations in two scenarios: Implementation using OpenCL High-Level Synthesis (HLS), and hand-optimized implementation using RTL.

2.6.1 OpenCL Implementation Evaluation

We first present a detailed performance evaluation of ZFP-V2 compared against our best-effort hardware implementation of the unmodified ZFP algorithm, a well as multithreaded software.

FPGA Platform

We conduct our evaluation on the Intel HARP[3] platform, where an Arria 10 GX FPGA (10AX115N2F40E2LG) is connected to a host system through PCIe. The host side has two sockets and each one has an Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz. The available host memory capacity is 376 GB.

Evaluated Implementations

We evaluate our FPGA implementation of ZFP-V2 against both software version of ZFP and our best-effort FPGA implementation of ZFP:

- Software ZFP: Multiple parallelized implementations of ZFP with different threads are evaluated. We use Para-N to denote an implementation with N threads. All software versions are compiled with "**-O3**" by gcc. These versions are denoted *Para-N* in the following sections, where N is the number of threads.
- FPGA-based ZFP: We have developed a best-effort version of the original ZFP algorithm using OpenCL. All optimizations we attempted for ZFP-V2 except VLH is applied here. The version of Intel FPGA SDK for OpenCL used is 17.1.1.

Table 2.3: FPGA Resource Utilization of ZFP and ZFP-V

Versions	ALUTs	FFs	RAMs	DSPs	Compile time
ZFP-float	31%	12%	12%	2%	5h48m
ZFPV-float	27%	14%	13%	2%	2h29m
ZFP-double	33%	12%	9%	6%	5h21m
ZFPV-double	31%	12%	9%	6%	2h41m

Hardware Resource Utilization

Table 2.3 shows the Arria 10 resource utilization of ZFP and ZFP-V2 for both float and double versions. This table includes both compression and decompression pipelines. Compression and decompression pipelines by themselves consume about half of the numbers reported here. We can see ZFP-V2 uses less resource than ZFP because our VLH algorithm is less complicated logically. In addition, their respective compile time are also listed. As we can see, a simpler algorithm can significantly reduce the compile time required.

Compression Speed

Compression throughput is measured using how much uncompressed data volume is processed per second. Note that each dataset actually contains multiple input data files that have a great distribution diversity. Therefore we compress them individually and use the arithmetic mean of all of them as the average speed of this dataset.

Fig. 2.10 shows the average compression speed across 8 benchmarks. Each benchmark contains 7 implementations: Para-1, Para-4, Para-8, Para-16, Para-32, FPGA-ZFP and FPGA-ZFPV. We can see that for all benchmarks, FPGA-ZFPV shows significantly faster performance compared to FPGA-ZFP, and shows performance close to 2x in benchmarks QMCPACK and NWCHEM. Furthermore, FPGA-ZFPV shows comparable or better throughput even compared to Para-32, which is the 32-thread software version, in the benchmarks CESM-ATM, HACC, NYX, and QMCPACK. ZFP-V2 performed over 1 GB/s on all benchmarks

tested, and over 2 GB/s on three of the 8 benchmarks, and over 4 GB/s on NWCHEM. While the tested datasets were different, it consistently performed significantly better compared to the state-of-the-art FPGA implementation of SZ on the same FPGA platform [136], which demonstrated less than 850 MB/s compression on all published datasets.

The benchmarks HURRICAN, NWCHEM, and SCALE show better throughtput on Para-32 than FPGA-ZFPV, but we discovered this is because of some extreme data characteristics in some of the input data files, which we will analyze in detail in Section 2.6.1. In all benchmarks, a single pipeline of FPGA-ZFPV showed better performance compared to a 4-thread software implementation.

Another point of interest is that the software performance scaling levels out, or even becomes slower after 16 threads. The reason appears that contention in hardware shared resource (e.g., last level cache (LLC) and memory bandwidth) becomes more aggressive and severe as the concurrent running threads increase.

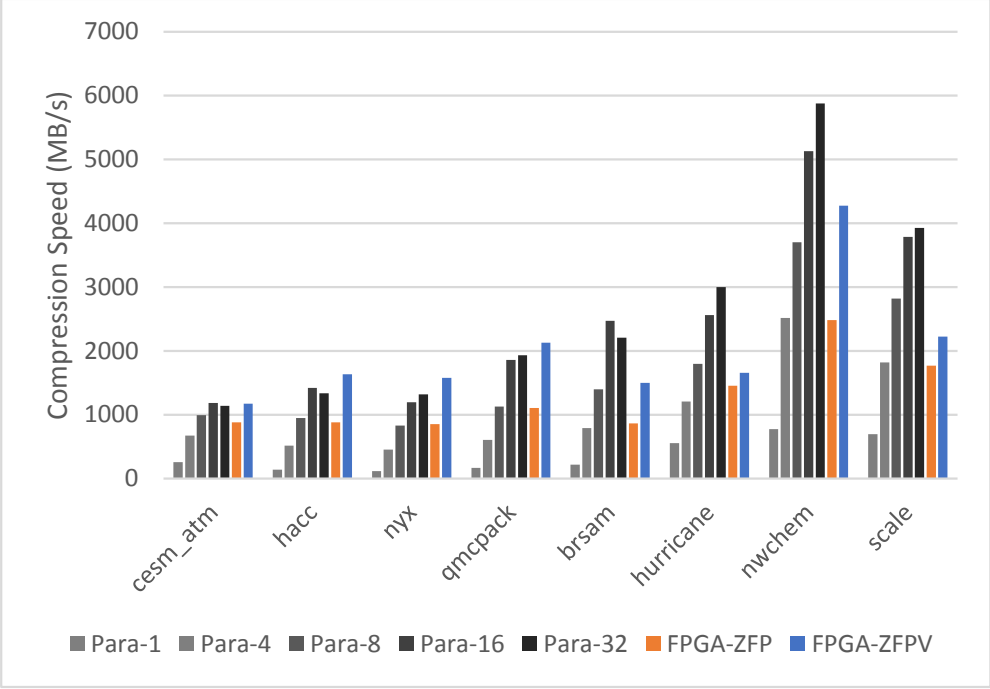


Figure 2.10: The Average Compression Speed distribution of Para-ZFP, Naive FPGA-ZFP, ZFPV

Decompression Speed

As the original ZFP uses block offset to identify the beginning of a compressed block bitstream, the para-N divides the whole bitsteram into N regions and stores N offsets to achieve high performance. FPGA-ZFPV uses the same method with a small ratio between N and the whole bitstream size.

Fig. 2.11 shows that FPGA-ZFPV has a higher average decompression throughput compared to FPGA-ZFP in all benchmarks, and significantly faster compared even to Para-32 in all benchmarks except BRSAM. For many benchmarks including HACC and NYX, FPGA-ZFPV demonstrates over 3x performance compared to Para-32, achieving over 10 GB/s on NWCHEM.

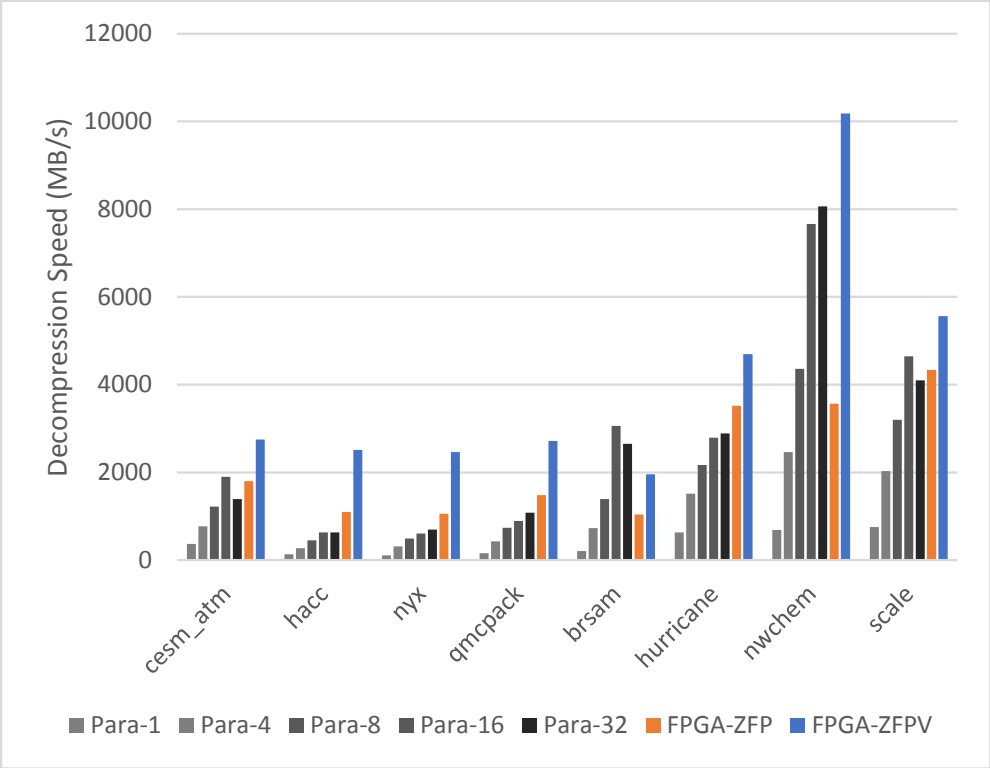


Figure 2.11: The Average Decompression Speed of Para-ZFP, Naive FPGA-ZFP, FPGA-ZFPV

Case Study on Fast Software Compression

In this section, we explain how Para-32 was able to perform compression faster than FPGA-ZFPV for some datasets, by analyzing an example case: HURRICANE.

This dataset contains 20 input data files and they show great diversity of the data distribution. We observe that most blocks are identified as a "Zero Block" in some files because all values are extremely small (say, 1E-90). For a Zero Block, ZFP just simply writes a "0" and returns immediately. No subsequent phases like transformation and embedded coding is performed. In this case, ZFP-V2 does the same operations as the original ZFP and does not show any advantage. For such a simple computation, the CPU with its fast memory access across multiple threads is able to achieve very high performance, compared to a single FPGA pipeline with a fixed memory access pipeline.

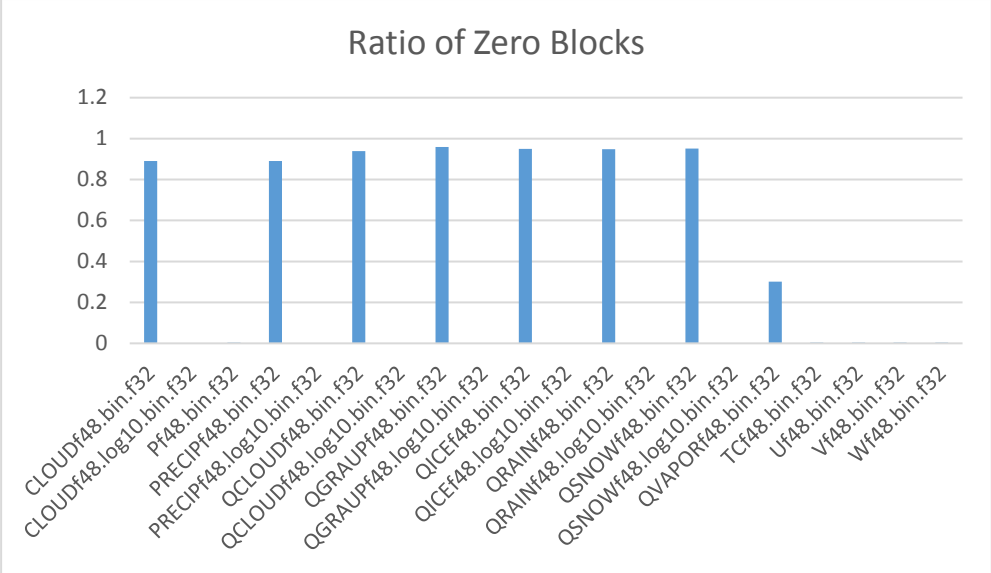


Figure 2.12: The Zero Block distribution of the benchmark Hurricane

Fig. 2.12 shows the Zero Block distribution of all data files in this benchmark. As can be seen, there are seven data files whose most blocks are Zero Blocks. When we compare it against Fig. 2.13 where the compression speed of each individual file is shown, we can see clear corresponding relation between the number of the Zero Blocks and the compression

speed: the more the former, the lower the latter. For files that are not predominantly zero, we can see that FPGA-ZFPV shows better performance compared to Para-32.

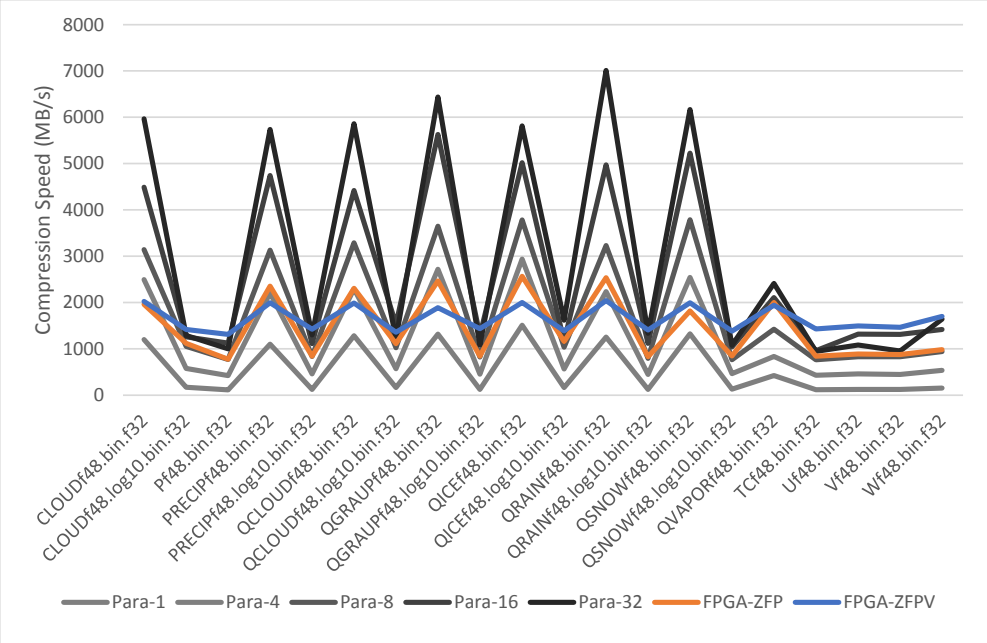


Figure 2.13: The Compression Speed of all data files in the benchmark Hurricane

Since files almost entirely consisting of zeros are somewhat extreme, we performed another performance evaluation of HURRICANE after removing these seven files, which can be seen in Fig. 2.14. We can see from these new results that the performance relations are now similar to the other benchmarks: FPGA-ZFPV shows better performance compared to Para-32, and software performance scaling levels out after 16 threads. Similarly, the benchmark NWCHEM and HACC contain similar, almost entirely zero data files that are very efficient with the multithreaded software implementations.

2.6.2 RTL Implementation Evaluation

Besides the HLS implementation, we also implemented unmodified ZFP, ZFP-V1, and ZFP-V2 using Bluespec, an RTL-level HDL, on a Xilinx VC707 FPGA development board. RTL implementations allows us to have finer control over the resulting hardware, and apply more

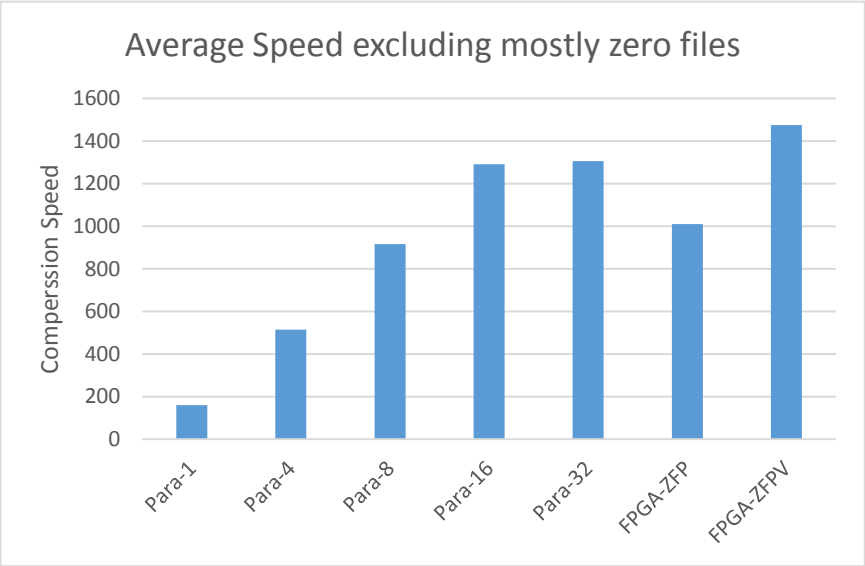


Figure 2.14: The Average Compression Speed of non Zero-Block data files in the benchmark Hurricane

fine-grained optimizations.

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 [89]. It has also been shown that at the compression levels achieved by these configurations, there is no significant amount of accumulated error as a result of lossy compression [46]. In fact, the error introduced by compression is typically **lower than the error caused by the limited accuracy of double precision floating point**, for the more stringent of the presented error bounds. This has been demonstrated in more detail in Table 2.2. In the rest of this section, we will denote the ZFP-V configuration using the dDe notation, where d is the dimensions of the compression unit, and e is the error bound, in terms of 1E-e. For example, ZFP-V2 with an error bound of 0.0001 would be denoted as 2D4.

2.6.3 Compression Accelerator Performance

Figure 2.15 shows the compression performance of a single pipeline of three accelerators: our best-effort accelerator for the unchanged 2D ZFP algorithm, ZFP-V1, and ZFP-V2. Each system is prefixed with **Orig-2D**, **1D**, and **2D**, respectively. Each accelerator is configured to run with four different error bounds presented earlier. Our prototype accelerators use a 256-bit datapath running at 250 MHz, wire-speed can be achieved at 8 GB/s per pipeline. Both ZFP-V1 and ZFP-V2 either approach or exceed wire-speed in most cases.

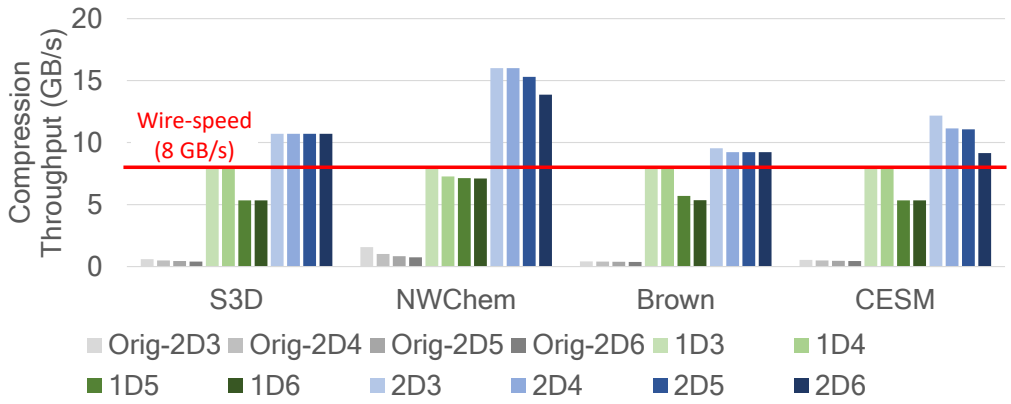


Figure 2.15: Compression performance of ZFP-V, across four datasets, with varying error bounds

The reason the ZFP-V2 accelerator exceeds the 8 GB/s wire-speed is because it actually has a wider input datapath (512 bits) compared to the rest of the system (256 bits). Normally it connects to the rest of the system through a deserializer, but we have benchmarked its raw performance using the natively wide dataset. For the compression accelerator, we can see that the performance across all benchmarks exceeds the wire-speed of 8 GB/s.

Especially the benchmark nwchecm, its speeds across 4 error bounds achieve 16 GB/s, 16 GB/s, 15.307 GB/s and 13.873 GB/s, respectively. This is because due to the data distribution, nwchem has very few bit planes that actually need to be compressed for each block. On average, the numbers of bit planes compressed of each block across the four error bounds are 4.517, 8.083, 11.038, and 14.034, respectively. Take the first error bound for example,

on average the variable length header algorithm just needs to process 4.517 bit planes and then immediately throw away the rest of the input block, which allows the next input block to be processed immediately and does not cause much back pressure to the front end of the pipeline (before variable-length header encoding).

We also can see that the throughput drops as the error bound becomes smaller for some datasets. This is because more bit planes need to be encoded for a smaller error bound for these datasets.

From the figure, we see that both ZFP-V1 and ZFP-V2 vastly outperform the unmodified ZFP accelerator. We are confident that our best-effort implementation of the unmodified ZFP is comparable to the state-of-the-art, since the performance demonstrated is similar to both our best-effort implementation using OpenCL on an Intel FPGA [121], as well as the published numbers for the unmodified SZ algorithm accelerator [136]. The source of the slow performance of ZFP is presented in Section 4.4, and is due to the inherent inefficiencies of the group testing-based encoding scheme when implemented in hardware. We also note that single-thread software performance of unmodified ZFP is even slower than the FPGA accelerator performance.

This performance difference is especially significant considering the chip resource utilization numbers presented in Table 3.3. Even considering the larger resource utilization of the ZFP-V2 accelerator, ZFP-V2 demonstrates almost an order of magnitude better performance per LUT compared to unmodified ZFP. ZFP-V1 requires even less resources than ZFP, but at the cost of lower compression efficiency as shown in Figure 2.9.

2.6.4 Decompression Accelerator Performance

Figure 2.16 shows the decompression performance of a single pipeline of the same three algorithms, with the same four error bound configurations. Similarly to the compression performance, both ZFP-V algorithms vastly outperform the unmodified ZFP algorithm, achieving much higher throughput per LUT. Figure 2.16 shows that a single pipeline of the decompressor often does not achieve wire-speed. This is due to the relative complexity of the decompression algorithm. Of course, thanks to the high throughput per LUT of the ZFP-V algorithms, wire-speed can be trivially reached with more pipelines. Similarly to the compression performance, the throughput drops as the error bound becomes smaller for some datasets.

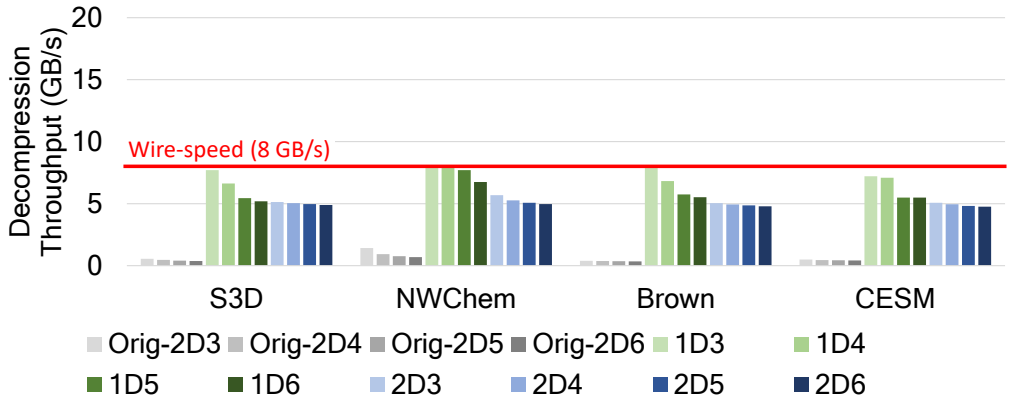


Figure 2.16: Decompression performance of ZFP-V, across four datasets, with varying error bounds

2.7 Summary

This chapter presents ZFP-V, a hardware-optimized floating point compression algorithm, which replace the ZFP algorithm's inherently serial embedded coding scheme with a higher-efficiency header based scheme.We implement and evaluate multiple variants of ZFP and show ZFP-V can achieve wire-speed compression/decompression on multiple real scientific

benchmarks with a small amount of compression ratio loss.

ZFP-V is capable of providing compression and decompression performance that can keep up with the high throughput of modern network and storage devices without having to consume valuable CPU performance. Furthermore, because the on-chip resource requirement of ZFP-V is low, it either frees up the FPGA for more application-specific acceleration, or multiple ZFP-V pipelines for faster throughput. Using ZFP-V, we plan to both improve the performance of high-performance scientific computing applications while also lowering the cost of storage and network requirement.

Chapter 3

BurstZ+: Scientific Computing Acceleration with ZFP-V

3.1 Introduction

The overall goal of this dissertation is to solve the communication bandwidth bottleneck between host and high-speed accelerators via compression. In the previous chapter, we demonstrated that minor algorithmic adjustments can result in an acceptably high performance accelerator of the traditionally performance-intensive lossy floating-point compression algorithm. In this chapter, we describe the impact of integrating hardware compressors and decompressors with application-specific accelerators, specifically in the area of grid-based scientific computing.

3.1.1 The BurstZ+ Platform

We present BurstZ+, which addresses the communication bandwidth issue of scientific computing accelerators by providing the computation engine with a highly efficient compression engine. Specifically, ZFP-V accelerators. 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. The compression engine uses a novel error-bound lossy compression algorithm with high enough compression ratio to improve the effective PCIe bandwidth to DRAM-levels.

This approach is especially valuable for the widely-used **iterative method** of scientific computing, where the output of a computation iteration is the input to the next iteration, across many thousands or millions of iterations [119, 106, 62, 116]. The initial and final state of the data for our prototype is currently compressed offline by a software implementation of the compression algorithm. Once the compressed initial data is entered into the system, the decompression and compression required for the subsequent iterations of computation is performed entirely by the accelerator. Data is always stored in a compressed format regardless of whether it is stored in memory, storage, or host, and it is also streamed to and from the accelerator in a compressed format without involving software compression. In fact, in many real-world scientific computing scenarios, lossy compression of collected scientific information is already applied before archiving, in an attempt to save storage resources [10, 32]. The compression accelerators on BurstZ+ can also be trivially retargeted to compress the initial and final states.

While some existing work has explored the use of lossless floating-point compression accelerators to improve the effective bandwidth of accelerator memory [128] for scientific computation, BurstZ+ is the first work to take advantage of the order-of-magnitude higher compression efficiency of error-bound lossy compression algorithms to close the much larger

bandwidth gap between the host and accelerator.

The novel compression algorithm ZFP-V, used in BurstZ+, is a variant of the lossy, error-bound compression algorithm ZFP. ZFP can regularly achieve compression ratios of 4× to 10× and beyond on multi-dimensional matrices of floating point data, but some of its design features prevents efficient hardware implementation. ZFP-V modifies ZFP and introduces a new embedded coding scheme which allows very efficient hardware implementation. As a result, it is capable of providing wire-speed compression and decompression of floating point data while using only a small fraction of on-chip resources. The new coding scheme trades a small amount of compression ratio for an order of magnitude performance improvement. As a result, our ZFP-V implementation is both high performance enough to supply the accelerator cores with enough bandwidth, as well as achieves high enough compression efficiency to close the bandwidth gap between the PCIe and on-board DRAM bandwidth. In fact, our compression accelerator is efficient enough to remove not only the PCIe performance bottleneck, but also improve the effective performance of the on-board DRAM by storing compressed data even on the fast on-board DRAM, and decompressing on the fly.

3.1.2 Applications: Stencil Computation on a Structured Grid

We evaluate the benefits of BurstZ+ using three typical iterative HPC applications with a wide range of *operational intensities*, which represents the amount of computation per memory access. All three applications belong to the Structured Grid class of applications, which is a popular method of scientific computing commonly used in areas including climate and seismic simulation, as well as approximate solutions of many other systems of partial differential equations. Complex partial differential equations can be systematically translated to multiple **iterations** of *stencil computation* application, where each cell in a multidimensional array is updated according to some fixed pattern, called a stencil. A stencil updates each

cell it is applied to, based on the values stored in a small number of surrounding cells. Since the same stencil is independently applied to every cell, the resulting computation pattern is very regular, as well as theoretically easily parallelizable.

The applications of focus in this work are: 3D heat dissipation simulation, computational fluid dynamics using the Lattice Boltzmann Method (LBM), and signal noise reduction using Speckle Reducing Anisotropic Diffusion (SRAD). These three applications have been selected to represent various characteristics of stencil-based applications including operational intensity, which can determine whether the application performance is limited by computation or memory [102, 71]. The operational densities of our implementations span between 0.8 and 12 FLOP/Byte, which covers a realistic range presented in existing research.

For example, the 3D heat dissipation kernel involves memory access complexity due to its three-dimensional nature, but has a low operational intensity of less than 1 FLOP/byte, which puts relatively more pressure on the memory system. On the other hand, 2D LBM is a more computation-bound application with larger, multidimensional tuple sizes per cell, with a high operational intensity of over 12 FLOP/byte. SRAD represents kernels with low data dimensions as well as a moderately low operational intensity of 7 FLOP/byte. We think these three kernels can do a good job of demonstrating performance and effectiveness of BurstZ+ in various real-world conditions.

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 [26, 129, 91, 30, 28, 96, 90, 130, 113, 112, 114, 9]. However, many previous works 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.

Caching algorithms such as temporal blocking help mitigate 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** [117, 40]. 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.

3.1.3 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 peak measured performance of over 11 GB/s. While the VC707 board is not as capable as newer devices such as those used by the Amazon F1 cloud instances, we argue that the insight from our prototype is directly applicable to the newer platforms, since the capabilities of device components have scaled at a similar rate. We describe this further in Section 4.5.

On the BurstZ+ prototype, we have implemented the three stencil application examples: 7-point 3D heat-transfer, 2D Lattice Boltzmann Method for computational fluid dynamics, as well as a 2D noise reduction using Speckle Reducing Anisotropic Diffusion. As described earlier, these three applications have varying computation requirements and data structures, and will help show a comprehensive evaluation of BurstZ+.

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 support 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. Even compared to such favorable conditions, BurstZ+ is able to achieve higher performance by improving even the effective bandwidth of the on-board DRAM module via wire-speed compression. As a result, BurstZ+ is able to move the performance bottleneck away from the PCIe link, turning it into a desirable situation where performance is bound only to the amount of actual useful computation the stencil accelerator can do.

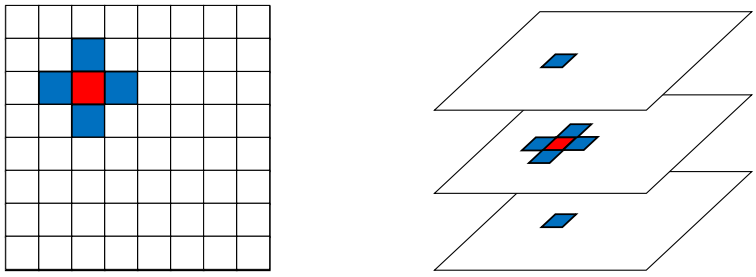
We also evaluate the projected end-to-end performance of a faster stencil accelerator implemented on a larger FPGA platform, with and without BurstZ+ support. We show that the BurstZ+ system can continue to scale with more internal computation capacity, while the same accelerator without compression will quickly saturate the PCIe bandwidth, until the BurstZ+ system achieves almost 7× the performance of a conventional accelerator.

We note that stencil core implementations often use various optimizations, such as temporal blocking, to improve caching effectiveness and 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.

3.2 Background and Related Work

3.2.1 Stencil Computing and its Acceleration

Stencil computing is an iterative computing method, which operates on a multidimensional grid representation of data. The contents of each cell varies according to the application requirements, spanning from single floating point values per cell for the simple heat dissipation application and SRAD, to 9 floating point values for 2D LBM, 19 floating point values for 3D LBM, and even beyond. 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 surrounding area. Figure 3.1 shows a graphical representation of a 2-dimensional 5-point stencil and a 3-dimensional 7-point stencil. 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.



(a) 2D 5-point stencil **(b) 3D 7-point stencil**

Figure 3.1: Example 2D and 3D stencils

Many useful partial differential equation systems can be systematically translated to a stencil form, which can achieve high accuracy with much less computational overhead. A wide variety of stencils have been designed depending on the application, including 9-point 2D stencils for 2D Laplacians, 25-point 3D stencils for 3D Euler equations, and many more. Each stencil kernel defines the computation to perform using the surrounding input points, and each grid point can store multi-dimensional data depending on the nature of the problem

being solved.

For example, the Lattice Boltzmann Method (LBM) is an important stencil computation method for computational fluid dynamics. Grid cells in the LBM method store a multidimensional tuple with values including floating-point representations of particle distribution in multiple grid directions, typically spanning 9 to 19 directions for particle movement. It also must handle many corner cases that make implementation more complex than simple stencils such as heat dissipation. One of the most prominent special cases is handling boundary conditions when fluid particles run into solid bodies: Depending on specific applications, several different boundary condition models with different complexities have been proposed, including Bounce Back, Boundary Conditions with Known Velocity, Periodic Boundary Conditions, and Imposed pressure Difference Boundary Conditions. Figure 3.2(a) shows a graphical representation of a D2Q9 LBM with 9 speed directions, and Figure 3.2(b) shows the boundary cells. For example, the top boundary cell must calculate f_4, f_7, f_8 using the given boundary condition in order to represent the interaction in high fidelity. In this work, we used the Heat Diffusion implementation introduced in [93].

Speckle Reducing Anisotropic Diffusion (SRAD) [140, 122, 22] is another important stencil which is used for ultrasonic and radar imaging applications to remove locally correlated noise, known as speckles, without destroying important image features. The value of each point in the grid depends on its four neighbors. Specifically, it needs to calculate each point's four direction derivatives with its four neighbors and then conduct a series of follow-up operations including gradient, laplacian, diffusion coefficient and divergence calculation. Compared with a 2D 5-point stencil, it needs more floating-point operations. It conducts a 2-stage update of the whole image: the first stage calculates the diffusion coefficient of each point and the second stage calculates the divergence of each point and eventually updates the image.

Due to their importance in many scientific applications, there have been a great amount

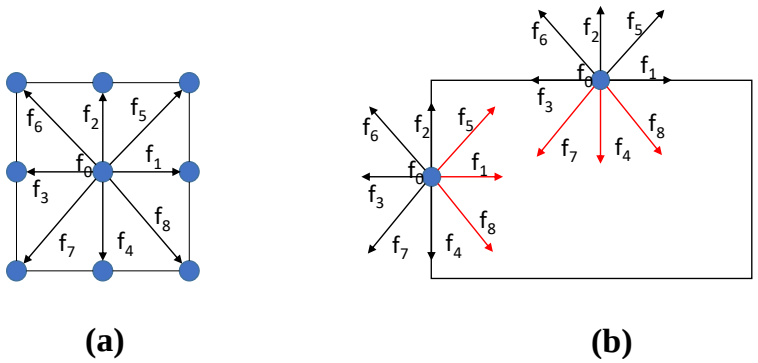


Figure 3.2: Example 2DQ9 LBM and its boundary cells

of previous studies 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 on FPGA-based acceleration as they often demonstrate very high power efficacy [144, 98, 70].

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 [56, 95, 30]. 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. Despite these efforts, one of the primary performance limitations of stencil accelerators is still the communications bottleneck when data spills over the accelerator memory capacity.

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 [107, 14, 21]. 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 3.3. 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.

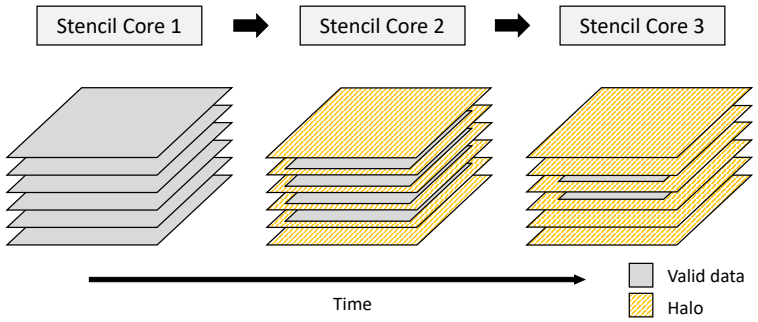


Figure 3.3: Deep temporal blocking increases the size of the *Halo*, reducing the amount of valid data

Most modern stencil accelerator designs take advantage of both tiling and temporal blocking, and more [41, 129, 39, 23, 143, 111]. A large body of work has focused on determining an optimal tiling and temporal blocking methods given the accelerator platform [111, 26, 30], as well as devising performance models and characterization methods about various memory optimizations [31, 39]. There has been research into efficient generation of stencil accelerator on FPGAs using high-level languages such as OpenCL [129, 143, 144].

Communication Bottleneck between Accelerators and Host

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 can 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 limit 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 [117, 40]. This is the situation we are interested in.

Some existing work have explored the use of floating-point compression to reduce the size of intermediate data [128]. However, we note that the goals of this project and ours are different. The custom, lossless floating point compression algorithm presented in [128] achieves high throughput, and successfully improves the effective bandwidth of on-board memory. However, our goal of closing the bandwidth gap between PCIe and on-board memory requires the higher compression efficiency of lossy compression algorithms such as ZFP.

In this work, we mainly focus on the issue of removing the host-side 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.

3.2.2 Error-Bounded Lossy Compression of Floating-Point Data

A traditionally effective method for reducing the overhead of data movement is compression. Lossless, data-oblivious compression methods including DEFLATE [33] and LZW [133] have been very effective in compressing enterprise data. High-throughput compression algorithms

such as LZO [97], LZ4 [25] and Stream VByte [79] sacrifice varying amounts of compression efficiency for speed, and has been useful in many high-performance processing environments, in applications including compressing network traffic [44, 134] and operating system swap space compression [74] for distributed processing. For example, stream VByte has 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 [82, 34]. 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 [82].

Another class of algorithms is floating-point aware, taking advantage of the knowledge of floating-point encoding schemes. Furthermore, an effective class of compression algorithms for floating point values is lossy compression algorithms such as ZFP [82, 36] and SZ [34]. 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, which 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 [89].

Research has shown that these methods do not cause meaningful quality degradation for realistic workloads [10, 65], even for iterative algorithms where the intermediate data is compressed and decompressed between every iteration [46]. As a result, such algorithms have been used in a wide array of applications including medical image reconstruction [52], extreme weather simulation [99], extreme-scale scientific frameworks [55], and many more [87].

In Chapter 2, we have already addressed how these algorithms can have high computational requirements, and are often not sufficiently fast even with hardware accelerator implementations. We have also presented how minor, hardware-aware modifications to the algorithm can result in sufficiently fast hardware accelerators.

3.3 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 in 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 high for the stencil implementation on the accelerator to handle, the ideally achievable maximum performance will be limited by 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 3.1. 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 affected by PCIe performance limitations. *compress4* assumes the existence of a wire-speed compression accelerator with an average compression ratio of 4×, which alleviates the performance bottleneck of both PCIe and memory.

Figure 3.4 shows the roofline analysis of these configurations, comparing the end-to-end per-

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 3.1: Different configurations for roofline analysis

formance of the accelerators as the computation performance improves while other systems characteristics remain the same. Once the peak internal performance of the accelerator grows beyond a certain point, the performance of each configuration is either limited by PCIe bandwidth, or by on-board memory bandwidth. For example, PCIe4 assumes data size exceeds the on-boad memory capacity, and must be streamed over PCIe. Its improves with computation performance of the accelerator hardware until the throughput it can sustain exceeds the PCIe bandwidth of 4 GB/s. On the other hand, largemem assumes data size does not exceed memory capacity, and performance continues to improve until data rate hits the much higher memory bandwidth limitation of 10 GB/s. 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 works on out-of-core stencil acceleration [117, 40].

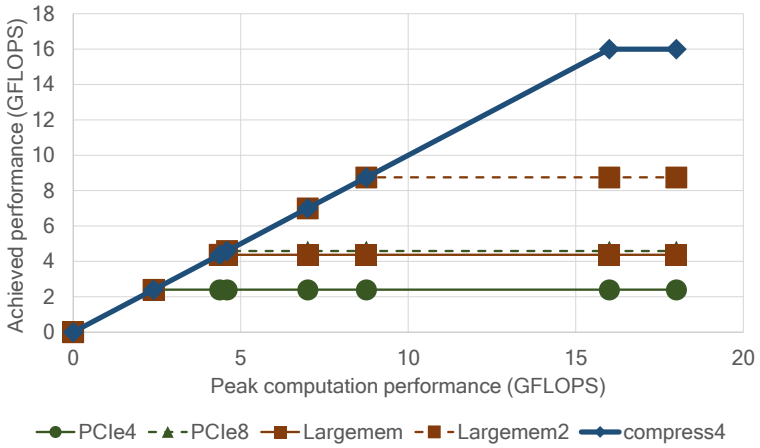


Figure 3.4: The stencil accelerator's performance is limited by both PCIe and DRAM's bandwidth

The analysis presented in Figure 3.4 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 end-to-end 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?

3.4 BurstZ+ Architecture

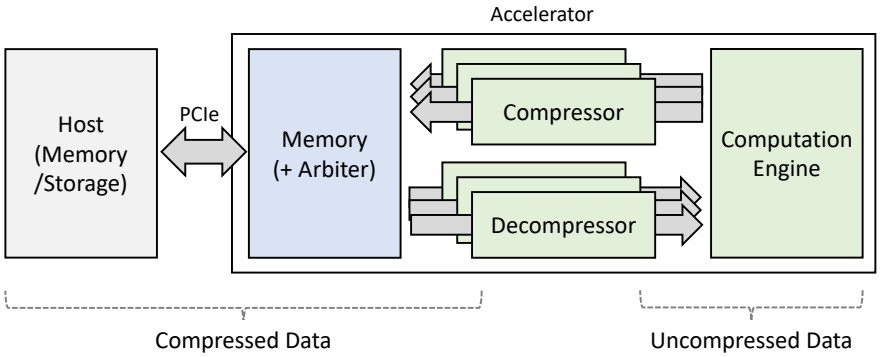


Figure 3.5: The overall architecture of BurstZ+. Data is stored compressed until it is used by the computation engine

Figure 3.5 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 one or more FPGA accelerators connected to the host server over PCIe. The BurstZ+ platform implementation programmed on 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. Each compressor and decompressor presented here can contain one or multiple internal *encoder(s)* or *decoder(s)*. The internal architecture of compressors and decompressors are described in

more detail in Section 2.4.

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 data access characteristics of the kernel. 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 stream, 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.

Furthermore, we note that our optimized compressor and decompressor accelerator performance is fast enough to not only make use of the maximum PCIe performance with a compressed stream, it is fast enough to make the maximum use of the on-board DRAM bandwidth. This is why the compressor and decompressor arrays are located between the on-board DRAM and the computation engine, and not between the PCIe and DRAM. This way, the computation engine has access to the fast on-board DRAM bandwidth multiplied by the compression accelerator, which is much faster than simply moving the data over PCIe in a compressed format, and using the DRAM as-is.

3.4.1 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 memory device characteristics such as row buffers and burst lengths, memory access is typically much faster for sequential accesses compared to random access. 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). 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. Intelligent arbitration is important because when multiple entities are accessing memory at the same time, the access requests arriving at the memory may be very random even if each entity's access pattern is sequential.

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 3.6.

The arbiter is parameterized so that the number of endpoints can be configured at compile time. Each endpoint interface also includes enough buffer space and corresponding flow control 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.

3.4.2 Example Stencil Application Accelerators

To evaluate BurstZ+, we accelerate three different stencil applications on our BurstZ+ prototype: a 3D 7-point stencil computation core and two 2D 5-point cores with different data types per element. To support these applications we implement flexible 2D and 3D sten-

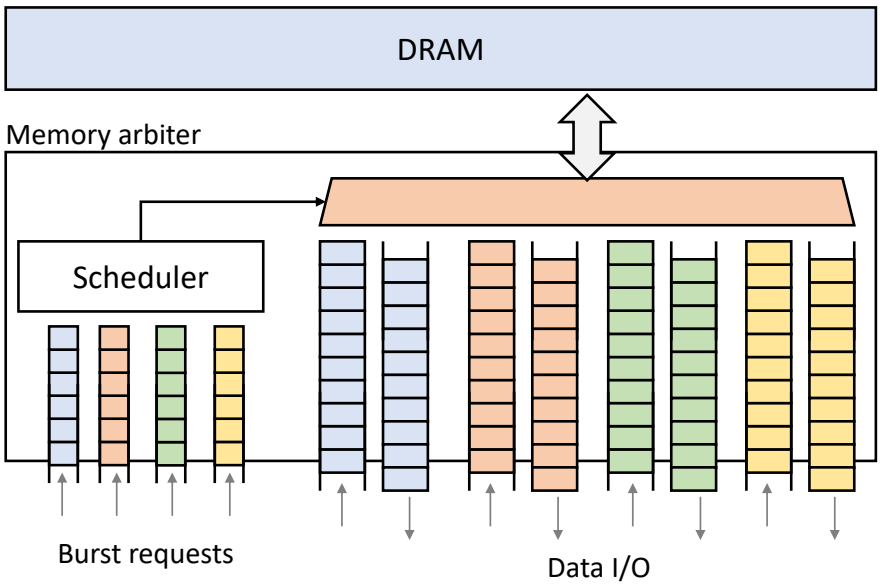


Figure 3.6: The memory arbiter provides high-performance multiplexing to multiple end-points

cil accelerators on BurstZ+, each of which can be configured at compile time with various parameters including computation between the cells, as well as the data type of each cell. The 2D accelerator also implements temporal blocking to further improve the operational intensity of the accelerator. The 2D and 3D cores are used to run the algorithms described in Section 4.1. The 3D stencil core executes a 3D heat dissipation simulation workload, and the 2D cores execute a D2Q9 LBM fluid dynamics simulation, as well as SRAD. Table 3.2 describe the characteristics of the three stencils. Previous research has placed most stencil applications of interest at operational intensities between 0 and 10, with the memory bandwidth bottleneck becoming a more serious issue with lower operational intensities [102, 71]. At a glance, we can expect applications such as the 3D heat dissipation stencil with, low operational intensities, to be more sensitive to communications bottleneck issues, whereas applications such as LBM with high operational intensities should be more resilient. With these three realistic accelerators, we expect to present a comprehensive evaluation on the effectiveness of BurstZ+.

	3D Heat	SRAD	LBM
Dimensions	3D	2D	2D
Points	7	5	5
Cell Size (Bytes)	8	8	72
Operational Intensity (FLOP/Byte)	0.88	7	12.12

Table 3.2: Various characteristics of the benchmark stencils

3D Heat Dissipation Stencil

Figure 3.7 shows the view of the working data set from the accelerator point of view. A 3D stencil operates on a 3-dimensional grid of values, as seen in Figure 3.7(a). 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 3.7(b), a 3D 7-point 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 simple stencil core without in-memory tiling, which must read each cell three times, once for each input plane. But we would like to note that, 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. Our implementation 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 support memory re-use and make better use of memory bandwidth, 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 3.7(c).

Figure 3.8 shows how we load the three planes' 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. The three decompressors and one compressor is connected to the memory via four endpoints in the memory arbiter. Including the host interface via the PCIe, the memory arbiter is configured with five endpoints for this application.

In order to facilitate high parallelism and bandwidth, each element in the row buffer actually consists of multiple floating point values. For example, the datapath in our prototype implementation is 32-bytes wide, meaning four double-precision floating point values are

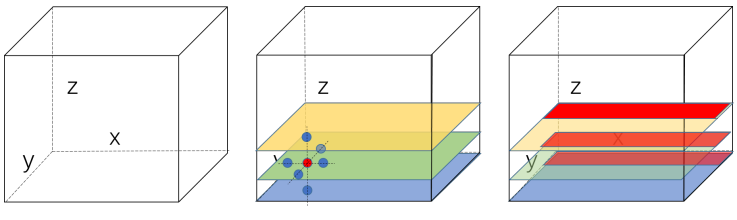


Figure 3.7: The basic principle of 3D stencil computation

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.

2D Stencil Core

Since 2D stencils do not need to access other z-planes, they require much less data to be streamed between computational iterations, making core design significantly simpler. Furthermore, 2D stencils have relatively low memory requirements for temporal blocking, by pipelining two accelerators to process two time steps in parallel. For 2D stencils, only two *rows* need to be streamed between accelerators, unlike 3D stencils, where enough memory for two whole *z-planes* need to be provisioned to stream intermediate data between cores. Two rows are typically small enough to cache either in on-chip or off-chip memory.

Our 2D accelerator is designed to pipeline a variable number of stencil cores to achieve parallelism as well as temporal blocking. Within the pipeline, the intermediate datapath can be configured according to the application characteristics. It is set to be wide enough that at least one cell can be forwarded every cycle, or 256 bits per cycle, whichever is larger. For the LBM implementation, the intermediate pipeline width is 576 bits, enough to store the nine double-precision floating point numbers per cell required by the LBM algorithm. For the SRAD implementation, it forwards 4 cells per cycle. Figure 3.9 shows a 3-stage

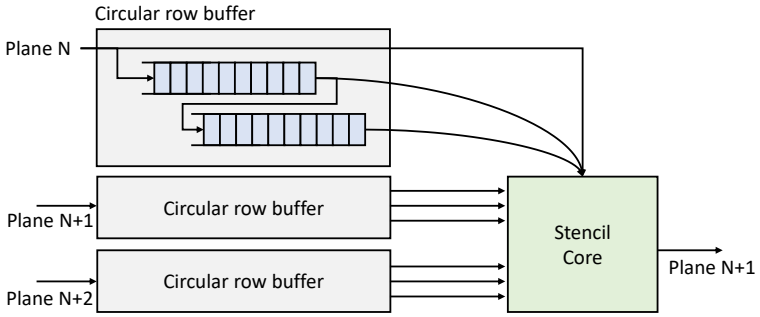


Figure 3.8: Three sets of two on-chip BRAM row buffers are used by the stencil core

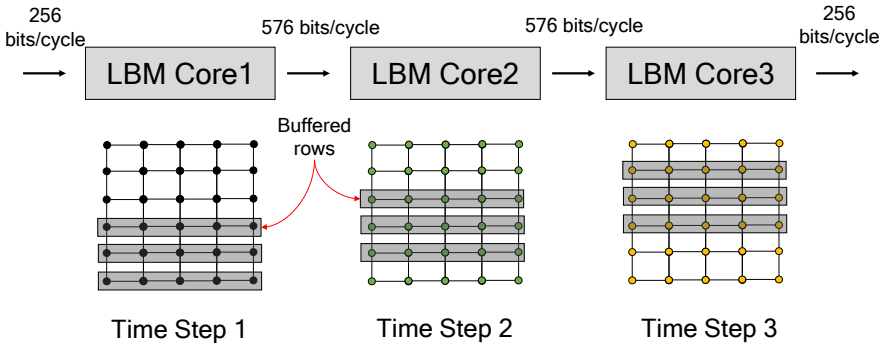


Figure 3.9: The Architecture of a 3-stage LBM pipeline

pipeline architecture with the LBM application, consisting of 3 cores numbered from 1 to 3. The decompressor can feed 256 bits to core 1 per cycle (8 GB/s) and the last core of the pipeline, core 3, can also output 256 bits per cycle.

On our prototype implementation, we instantiate two cores per pipeline for both applications, achieving state-of-the-art performance. For example, a previously published, optimized implementation of SRAD using OpenCL-based high-level synthesis on the Intel Arria 10 FPGA [142] has a run time of 4.17s when running for 100 iterations on an input of a 4000×4000 single-precision floating-point grid. This corresponds to a data ingestion throughput of 1.535 GB/s ($4000 \times 4000 \times 4 \times 100 Bytes \div 4.17s$). On the other hand, 2-stage pipeline achieves a steady 8 GB/s on a similarly placed Xilinx Virtex-7 FPGA.

3.4.3 Implementation Details

We have implemented a BurstZ+ prototype on a Xilinx VC707 FPGA development board. The VC707 board is equipped with a Xilinx Virtex-7 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 a theoretical peak of 4 GB/s duplex bandwidth. Our PCIe hardware module and driver delivers 3.1 GB/s of effective bandwidth over DMA. The accelerators and the intermediate datapaths are implemented to run at 250 MHz.

The VC707 is not a high-end FPGA by modern server standards, but we believe this is still a good platform for evaluating BurstZ+, because the relationship between PCIe bandwidth, FPGA capacity, and DRAM bandwidth has maintained relatively constant with future generations of FPGA development boards. For example, the FPGA accelerator installed on the Amazon F1 instances are based on the Xilinx VU9P chip, with over 2.5 million logic cells, about 5× the capacity of the VC707. Meanwhile, the F1 FPGA delivers over 15 GB/s of PCIe DMA bandwidth (~5× vs. VC707), as well as 68 GB/s of DRAM bandwidth (~6× vs. VC707) [131]. While some of these numbers cannot be accurately compared one-to-one (e.g., each logic cell of the Ultrascale+ and Virtex 7 chips are different), we do believe the approaces introduced by BurstZ+ will have similar scale of benefits on a more modern FPGA environment.

Table 3.3 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 total resource utilization is based on using one compressor and one decompressor. When using the simpler ZFP-V1 cores, The BurstZ+ platform consumes about 24% of the on-chip resources of our prototype platform, and less than **3%** of the on-chip resources of a modern, high-end FPGA such as the Virtex Ultrascale+. We also present the resource utilization of our best effort unmodified ZFP accelerator implementation, the performance of which we will present in Section 4.5 in relation to ZFP-V. We note that the resource utilization of the single unmodified ZFP accelerator pipeline is comparable to the published resource utilization numbers of an unmodified SZ accelerator pipeline [136], as well as the best-effort OpenCL implementation of ZFP on an Arria 10 FPGA [121]. 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. While using more complex ZFP-V2 cores results in slightly more resource usage, it is still a small amount of overhead in a more modern FPGA device such as the Virtex Ultrascale+.

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.

Module	LUTs	VC707%	VCU118%
Platform (PCIe+DRAM+Arbiter)	22K	7%	<1%
1x ZFP-V1 Decompressor	26K	9%	1%
1x ZFP-V1 Compressor	25K	8%	<1%
1x ZFP-V2 Decompressor	40K	13%	<2%
1x ZFP-V2 Compressor	41K	14%	<2%
1x Unmodified 2D ZFP Decompressor	29K	10%	<2%
1x Unmodified 2D ZFP Compressor	32K	11%	<2%
Total (using ZFP-V1)	73K	24%	<3%
Total (using ZFP-V2)	103K	33%	<4%

Table 3.3: FPGA LUTs usage breakdown of the BurstZ+ platform for stencil computation

3.5 Performance Evaluation

We demonstrate the effectiveness of our BurstZ+ platform, emphasizing the application performance benefits of BurstZ+ on our target stencil applications. 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. We also present the negative performance impact we would suffer, if we were to use an unmodified ZFP accelerator instead of our optimized ZFP-V.

3.5.1 Benchmark Datasets

In order to evaluate our system under realistic scenarios, we use real-world datasets from the *Scientific Data Reduction Benchmarks (SDRBench)* [2], which includes various real-world datasets from fields including climate simulation, molecular dynamics, and cosmology simulations. We selected three datasets from SDRBench which represent multi-dimensional data using 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.

3.5.2 Making Efficient Use of the Host-Side Link Bandwidth

Figure 3.10 shows the average bandwidth pressure put on the host-side PCIe link, when compressed data is being streamed over the PCIe link and decompressed by a single decompressor pipeline at the FPGA. The decompressed data rate and compression ratio used is the geomean of the configurations presented in Section 2.6.4. If the bandwidth pressure exceeds what is available from the PCIe link, compressed data cannot be supplied to the decompressor fast enough, and the decompressor will no longer function at maximum performance. This situation is depicted with the red hatch pattern. With only one pipeline, all configurations put less pressure on the PCIe link than is available. In such a situation, we can say the performance bottleneck has been moved away from the PCIe.

However, the balance between performance and communication may change if the gap between the accelerator performance and communication bandwidth continues to grow, represented by the increasing number of pipelines while the PCIe bandwidth is constant. The communication bottleneck may return if the required data rate of the compressed file to support normal operation is higher than the PCIe bandwidth, as the decompressor will be

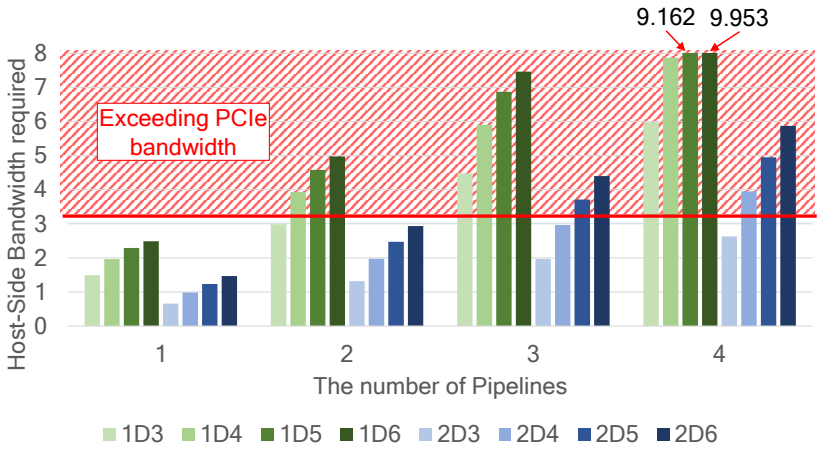


Figure 3.10: The communication bandwidth required for the full performance operation of multiple decompressor pipelines, for various fault tolerance settings of ZFP-V1 and ZFP-V2

unable to function at its best performance.

Given two algorithms emitting a decompressed stream at the same rate, an algorithm with higher compression ratios will put less bandwidth pressure on the communications link compared to one with lower compression ratio. We can also see this in Figure 3.10, where ZFP-V1 puts more bandwidth pressure on the link on average compared to ZFP-V2. As a result, the performance of ZFP-V1 will hit the PCIe bandwidth bottleneck quicker than ZFP-V2. For example, the PCIe bandwidth can only support 51.9% of the required compressed bandwidth to four pipelines of 1D3, meaning the decompressed data rate will also decrease by that much. However, four pipelines of ZFP-V2 in the 2D3 configuration can still run at full speed, delivering 21.04 GB/s of decompressed, effective data throughput from a mere 3.1 GB/s available from the PCIe. Considering the typical ratio of PCIe bandwidth and the amount of on-chip resources on modern FPGAs, we can confidently say that the compression efficiency and performance achieved by ZFP-V can still remove the bandwidth bottleneck of PCIe.

3.5.3 End-to-End Application Performance

Evaluation Configurations

Table 3.4 lists the system configurations for BurstZ+ and others. For evaluation, we use the ZFP-V1 accelerators since its relatively low compression ratio is still good enough to close the bandwidth gap between the PCIe and the accelerator bandwidth. With more pipelines of stencil accelerators, the host-side bandwidth requirement may exceed the available PCIe bandwidth as described in Figure 3.10. We also present the projected performance scaling of BurstZ+ with various number of pipelines as well as different compression algorithms, in Section 3.5.4.

We compared the performance of BurstZ+ against various other accelerator architectures that could be implemented on a hardware platform with the same components. Compared configurations 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. **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 scanned 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.

Figures 3.11, 3.12, and 3.13 show the end-to-end performance of the various system configurations described in Table 3.4, on 3D heat dissipation, LBM, and SRAD, respectively. All performance is normalized to **Largemem**, an unlikely situation when the working set

Name	Description
Configurations with ZFP-V compression	
B'z3	BurstZ+ with ZFP-V1, error bound of 1E-3
B'z4	BurstZ+ with ZFP-V1, error bound of 1E-4
B'z5	BurstZ+ with ZFP-V1, error bound of 1E-5
B'z6	BurstZ+ with ZFP-V1, 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 3.4: Evaluated accelerator configurations

is small enough to fit entirely in on-board memory. For each benchmark column, the left four bars represent the BurstZ+ system using each of the error bounds for compression. The remainder of the bars are different stencil accelerator architectures implemented on the same hardware platform, using the same application accelerator architecture. For application evaluations, BurstZ+ systems use a pair of one ZFP-V1 compressor and one decompressor for the 2D stencils, and three decompressors and one compressor for the 3D stencil.

As seen in Figure 3.10, a single compressor and decompressor pipeline running at full bandwidth do not fully saturate the back-end PCIe bandwidth, meaning there is more opportunity for higher communication bandwidth in such situations, unlike conventional configurations like **Nocomp** and **Largemem**, whose performance has already reached its limit due to PCIe and DRAM bandwidth limitations. We explore scenarios with more pipelines taking full advantage of the back-end communication bandwidth in Section 3.5.4.

3D Heat Dissipation Evaluation

Figure 3.11 shows the performance evaluations of the 3D heat dissipation kernel. 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 [60], we can be confident our stencil accelerator has a reasonable design. Higher GFLOPS can be achieved using the same temporal blocking methods used in the Intel design. However, these optimizations will affect all compared system configurations similarly, and are orthogonal to the data movement issue we are addressing. So in the interest of clarity, we instead present normalized performance results.

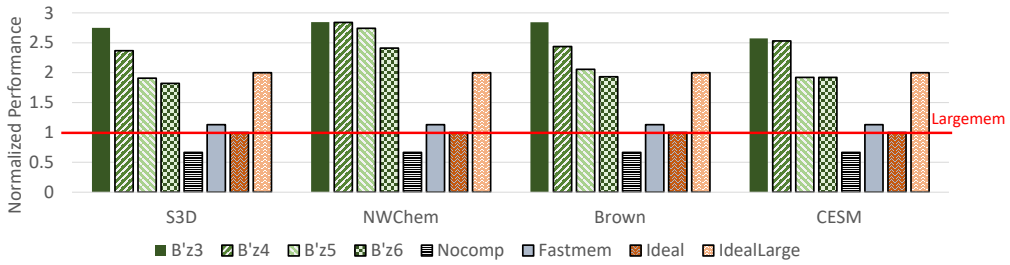


Figure 3.11: 3D stencil evaluation: BurstZ+ outperforms even in-memory systems with ideal caching

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**. IdealLarge 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×. 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. These results show that BurstZ+ is able to achieve benefits beyond what conventional caching approaches can achieve.

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**. A significant point to note is that BurstZ+ even outperforms Largemem, which is an entirely in-memory configuration, without the performance limitations of PCIe. These results show that BurstZ+ is able to move the performance bottleneck away from the PCIe into the accelerator itself.

As a result, for all measured BurstZ+ systems, the biggest performance limiting factor is not the PCIe, but the on-board DRAM performance, unlike systems like **Fastmem**, which is limited only by PCIe bandwidth. This means that for BurstZ+, the problem has now moved away from communication bandwidth, and has 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.

2D LBM Evaluation

Figure 3.12 shows the performance of the 2D Lattice-Boltzmann Method. Unlike a 3D stencil, which can require complex caching to avoid having to stream a z-*plane* multiple times, a 2D stencil only needs to maintain a small number of one-dimensional rows on-chip to turn a kernel sweep into a single sweep over the dataset. As the size of a row is typically small enough for on-chip buffers, we do not consider special caching approaches in this application. As a result, projected ideal caching performances represented by **Ideal** and **IdealLarge** are not presented. Instead, we implement **Nocomp** to stream data directly from PCIe DMA input

to PCIe DMA output, achieving maximum bandwidth with the given hardware. **Largemem** has the usual implementation of streaming data directly from memory.

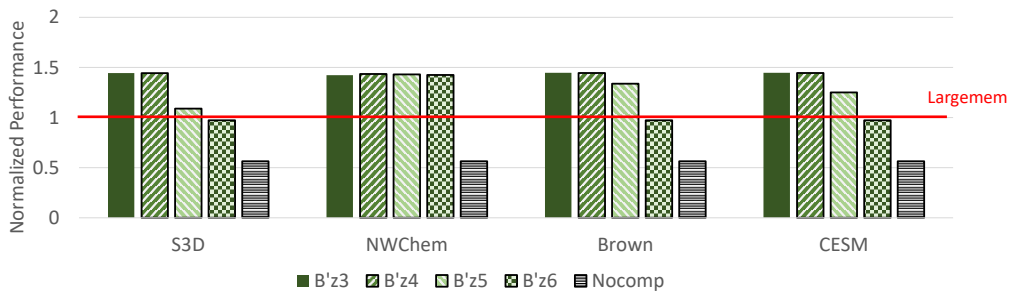


Figure 3.12: LBM evaluation: BurstZ+ outperforms an accelerator with no compression by over 2×, and often outperforms even in-memory accelerators

Our results show that the same favorable performance relations continue even on a 2D kernel, which puts even more pressure on the PCIe due to the small number of cell value re-use, as well as the much higher operational intensity. Compared to **Nocomp**, BurstZ+ consistently demonstrates over 2× the performance. BurstZ+ even consistently outperforms **Largemem**, which does a single scan over the fast on-board memory. We can see that the ZFP-V compression is effective enough, while also being fast enough, to remove the communication bottleneck of not only PCIe, but also from on-board memory.

We emphasize the performance presented is only for a **single-pipeline**, and BurstZ+ performance can scale further with more pipelines, unlike all other systems compared against. Unlike with the 3D stencil application, the DRAM is not the performance bottleneck in this scenario, as compressed data is streamed directly from PCIe to the decompressor and from the compressor to the PCIe. As a result, a single BurstZ+ accelerator pipeline does not fully saturate the back-end PCIe bandwidth, due to the effective bandwidth improvement via compression. This can be seen in more detail in Figure 3.10. This leaves open opportunities for BurstZ+ to **further scale performance with more pipelines**, unlike **Nocomp** and **Largemem**, whose performance has hit its limit due to PCIe and DRAM bandwidth limitations, respectively. In the BurstZ+ systems, we see that the performance bottleneck has been successfully moved away from communication to computation.

2D SRAD Evaluation

Figure 3.13 shows the performance of 2D SRAD accelerator systems. As with the LBM example, all BurstZ+ configurations outperform **Nocomp** by over 2×, and often even outperforms **Largemem**. While performance relations are somewhat different compared to the LBM application, the high-level observations are the same. BurstZ+ has successfully moved the performance bottleneck away from PCIe communication to computation, as can be seen from its superior performance while not saturating the back-end PCIe bandwidth.

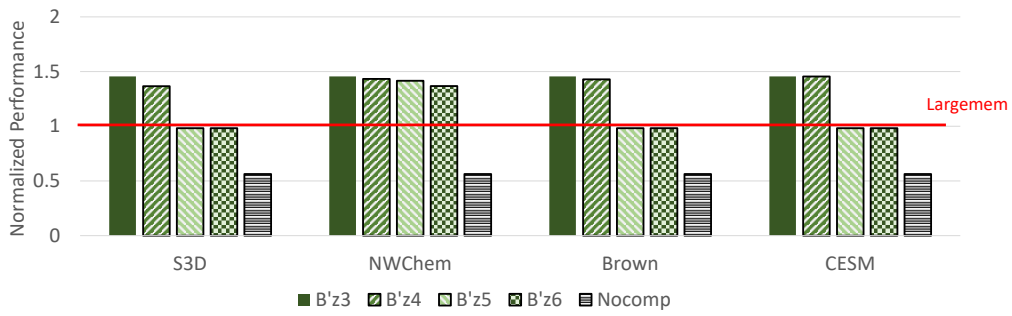


Figure 3.13: SRAD evaluation: BurstZ+ outperforms an accelerator with no compression by over 2×, and often outperforms even in-memory accelerators

Performance Impact of a Slower ZFP Compression Accelerator

Figure 3.14 shows the performance of the computing engine when a single pipeline of the original, unmodified ZFP accelerator is applied in place of ZFP-V. Due to the slow performance of the ZFP accelerators, the accelerator's performance is completely limited by the compression accelerator performance. Replicating the compression pipeline is not a viable solution, since a pair of ZFP compression and decompression accelerators consume more than 20% of the chip resources, putting a limit on how many compression/decompression pipelines can be instantiated. Even with 5 pipelines, the performance of BurstZ+ with the unmodified ZFP algorithm will be slower even compared to Nocomp.

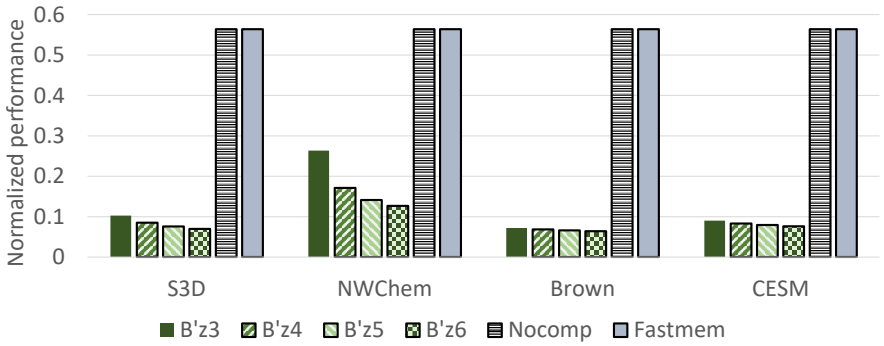


Figure 3.14: The performance of the computing engine when the original ZFP is used in BurstZ+

3.5.4 Scalability Analysis

As seen in Figure 3.10, a single ZFP-V pipeline running at full bandwidth does not saturate the back-end PCIe communication bandwidth, leaving opportunities for continued performance scaling beyond the single pipeline. This is in contrast to non-compressed configurations that have already hit their performance limitations due to bandwidth issues. While 2D stencils have the option of improving computation throughput using a deeper pipeline of stencil cores, such opportunities are less available to 3D stencils due to on-chip memory constraints.

Figure 3.15 shows the projected performance of various BurstZ+ configurations with one or more pipelines, normalized against **Nocomp**, which is limited by PCIe bandwidth. Each bar group represents the geomean of performance across the benchmark datasets, using either ZFP-V1 or ZFP-V2 with varying error bounds. For clarity, we assume a 2D stencil scenario where data is streamed directly from PCIe to PCIe. All configurations are over 2× faster than **Nocomp**, and reaching almost 7× with 2D3, with ZFP-V2 at an error bound of 1E-3.

The figure shows that with efficient compression configurations like 2D3, BurstZ+ can continue scaling performance well beyond a single pipeline, unlike configurations without compression. Interestingly, although the single-pipeline performance of ZFP-V2 cores are slower than ZFP-V1, having more efficient compression like ZFP-V2 with a lenient 1E-3 (2D3)

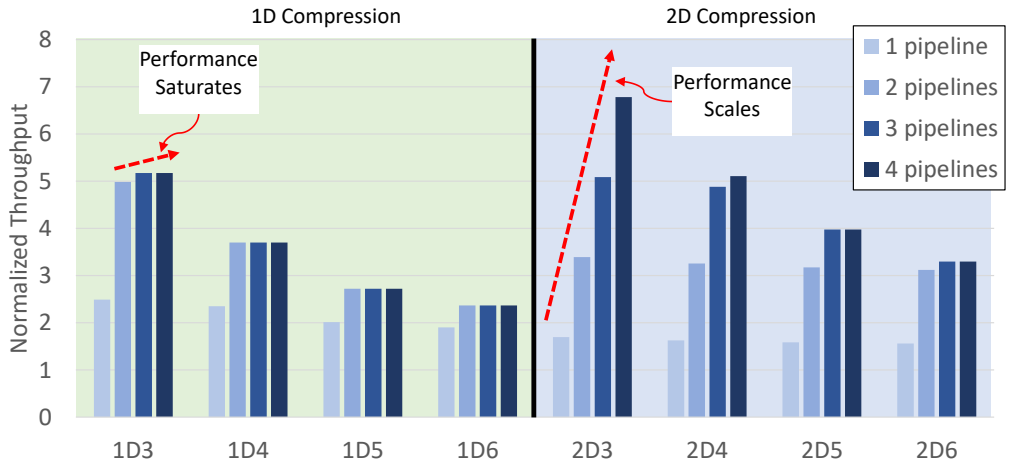


Figure 3.15: More efficient compression using ZFP-V2 allows good performance scaling within a strict bandwidth budget

allows better performance scaling under stringent PCIe bandwidth limitations, ultimately reaching higher performance. From our experience, it was difficult to fit more than four processing pipelines on typically available FPGA platforms, meaning even while saturating the computation performance of FPGA platforms, we are still not fully utilizing the PCIe bandwidth. This again shows that BurstZ+ is capable of successfully **moving the performance bottleneck away from PCIe** bandwidth to computation, achieving the goal of the system.

3.6 Summary

We present BurstZ+, a bandwidth-efficient scientific computing accelerator platform for large data. BurstZ+ uses a class of novel, hardware-optimized compression algorithms called ZFP-V, and successfully **removes the communication bottleneck between the host and the accelerator**, which is conventionally the primary performance limiting factor of large-scale scientific computing acceleration. In fact, BurstZ+'s ZFP-V 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 and datacenter operation, 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.

Chapter 4

ZipNN: High-Dimensional Similarity Search with Compression

4.1 Introduction

Last Chapter introduced the BurstZ+ platform, which uses high-efficiency floating point compression and decompression to overcome the PCIe communication bandwidth bottleneck between host and high-speed FPGA accelerator. In this chapter, we continue this line of exploration to another class of applications, which processes streams of integer values. We explore how minor modifications to integer stream compression algorithms can also allow accelerators to achieve sufficiently high bandwidth for accelerators. This chapter introduces the resulting accelerator, ZipNN, which uses combined integer compressors/decompressors to improve the performance of high-dimensional similarity search. In ZipNN, we implement multiple hardware-optimized compressors: Delta Compression, Run-Length Encoding (RLE), and a pipelined Group Varint encoding. This is based on an important observation: while a complex general-purpose compression algorithm such as DEFLATE has a high

overhead of implementation and operation, competent compression can be achieved with a combination of much simpler algorithms configured based on the statistical distribution of input data.

Many important applications including content-based search [17], plagiarism detection [20, 12], and bioinformatics [138, 141] depend on fast search into a large, high-dimensional dataset. A common formulation of this problem is the K-nearest neighbors (KNN) algorithm. This algorithm's basic idea is to search a high-dimensional dataset, represented as a set of vectors, in order to find K vectors that are closest to a given query vector based on a distance metric. Common distance metrics include Euclidean, Manhattan, Hamming, cosine similarity, and Dynamic Time Warping (DTW), and their selection depends on the data type and application requirements.

Due to the so-called *curse of dimensionality* [59], conventional indexing methods such as kd-trees increasingly become inefficient as the target data has more dimensions. As a result, instead of pin-pointing a query target via an index as a database would, high-dimensional similarity search must scan through a large amount of data and calculate the similarity between each data element in the dataset against the query, in order to keep track of the most similar elements. While approximate methods such as Locality-Sensitive Hashing (LSH) [59] can help reduce the search space, the majority of the time taken during a high-dimensional similarity search is still spent calculating similarities and keeping track of the top results, as opposed to traversing index structures [80, 126].

Due to the sheer size of the datasets of interest in this area, it is often required to store and access them from secondary storage [73, 18]. Since most secondary storage devices are much slower compared to DRAM, this results in the performance of the whole system being limited by the capability of the secondary storage device to supply the computation unit with data quickly enough. Furthermore, storage access granularity is typically in the order of kilobytes, much coarser than system memory, which further impacts indexing efficiency.

As a result, a great amount of effort has been put into creating optimized indexing and data layout schemes for storage access, as well as approximate methods [15, 59, 69].

This issue is aggravated with the introduction of computation acceleration engines such as Graphic Processing Units (GPUs) [47, 48] and Field-Programmable Gate Arrays (FPGAs) [4], which further moves the bottleneck away from computation to data transfer. As a result, accelerator-based architectures can deliver extremely high performance when the dataset fits in the on-board memory resources, but quickly becomes bottlenecked by the storage bandwidth when the dataset becomes larger [66].

Compression can help overcome storage bandwidth limitations by increasing the effective bandwidth of existing storage, **as long as the compression algorithm is fast enough to keep up with the storage and accelerator bandwidth**. Unfortunately, most popular compression algorithms including DEFLATE and LZ4 are designed with software implementations in mind, and their straightforward FPGA implementations are often not fast enough to support storage or computation bandwidth.

In order to address these issues, we present ZipNN, which explores mitigating the storage performance bottleneck of hardware-accelerated high-dimensional similarity search via two approaches:

1. **Compression accelerators optimized for the application, as well as for hardware implementation.**
2. **Deploying the accelerator near-storage.**

Many high-dimensional datasets are sparsely encoded, meaning a large part of these datasets are columns of integer streams, representing index values such as user index, vertex ID, or document index. To efficiently store and process these data types, we have designed a very high performance accelerator for integer stream compression, which consists of a configurable

pipelined chain of delta compression, run-length compression, and a hardware-optimized Group Varint compression accelerator. A compression accelerator is assigned per column, and each accelerator can be configured to use a subset of the available compression algorithms according to the observed data distribution of the column.

Even for such simple algorithms, some hardware-optimized re-design was necessary to get sufficient performance on FPGA accelerators. Delta compression and run-length compression can be easily implemented and achieve wire-speed on FPGA. However, a straightforward implementation of a conventional Group Varint compression algorithm cannot achieve sufficient decompression speed, due to the small size of the group. As described in Section 1.3, a group in a conventional Group Varint encoding contains 4 integers, whose 2-bit headers are grouped together into an 8-bit byte. For software implementation, this byte-aligned format is shown to achieve significant performance improvement over the original non-grouped format.

However, a single-byte header is still too small for efficient FPGA implementation, because a variable-width shifter in an FPGA requires multiple cycles of latency in order to shift the input stream and discover the next header block. Our original implementation's variable-width shifter took 6 cycles to discover the next header from a 256-bit datapath, limiting the performance of a 4-block Group Varint algorithm to 4 32-bit integers per 6 cycles, which corresponds to a meager decoding speed of 0.67 GB/s under a 250 MHz design.

To overcome this limitation, we further modify the Group Varint encoding to group a much larger number of headers, in order to overlap the shifter latency with payload processing. The resulting large header is organized into a two-level hierarchy of headers so the shifter can start the shifting operation even before the entire header is consumed.

ZipNN instantiates an instance of this accelerator for each column of the target dataset, and each accelerator pipeline can be configured to use a subset of the compression cores depending

on the data distribution. Tested on real-world datasets, each decompressor pipeline can achieve efficiency comparable to gzip with over 4× compression, while achieving wire-speed performance. On our FPGA prototype, this is 8 GB/s *per pipeline*. This is an order of magnitude faster performance compared to FPGA implementations of gzip [76].

Furthermore, ZipNN is designed as a near-storage accelerator. Due to the very fast performance of modern flash storage devices, modern SSDs typically show a large performance difference between its peak bandwidth and the host-side PCIe link provisioned for it. A near-storage accelerator can take advantage of this high internal bandwidth. Thanks to the high performance of both the compression accelerators and the similarity calculation accelerator, ZipNN demonstrates efficient performance scaling with faster internal bandwidth.

We have evaluated our implementation on the MIT BlueDBM platform [67], and demonstrated the effectiveness of the ZipNN approach compared to standalone accelerator cards. When limited by the NAND flash bandwidth of BlueDBM, ZipNN demonstrated over 3× performance compared to the same accelerator without compression. We also projected the performance of faster storage devices by emulating it with its on-board DRAM. When configured with a reasonable 2× difference between internal and link bandwidth [63], ZipNN demonstrated a further 40% performance improvement, resulting in a total of 6× performance improvement over a standalone distance calculation accelerator. This is also an order of magnitude faster than a purely software implementation.

4.2 Background and Related Work

4.2.1 High-Dimensional Similarity Search

High-dimensional similarity search suffers from the so-called "curse of dimensionality". This describes a situation where indexing structures such as kd-tree and variants [15, 69] become rapidly less useful as the dimensionality of data becomes larger, because the relative importance of each dimension becomes rapidly lower. Approximate nearest-neighbor search algorithms address this issue by giving a statistical guarantee on the correctness of the result [85, 59, 80, 7]. Despite these optimizations, the majority of effort for high-dimensional nearest neighbor search is spent on computing the distance between the query and a sizable set of candidate neighbors [47].

There are various distance metrics used to calculate the distance between two data points, and their selection depends on the application and data type. For high dimensional data such as bag-of-words, images, and time series, popular metrics include Dynamic Time Warping (DTW) [16, 105], and cosine similarity [139, 95]. Since these algorithms are computationally bound, their performance depends on the performance of the computation engine.

To move the bottleneck away from computation, there have been many successful attempts accelerating high-dimensional similarity search using accelerators such as GPUs or FPGAs [120, 58, 47, 48]. As distance calculation became faster, the bottleneck moved from computation to memory. For large datasets that were stored in secondary storage, the storage performance quickly became the most primary bottleneck [66].

4.2.2 Compression Algorithms on FPGAs

Compression is an effective method for reducing the overhead of data movement, by reducing the amount of data to be stored and transferred. If data is stored in a compressed format and decompressed on the fly, the effective bandwidth of the decompressed data may be much larger than the pressure put on the storage or memory device. However, this will only be beneficial if the decompression algorithm throughput is at least as fast as the memory and the nearest-neighbor computation unit. There is a wide range of selection for performance-optimized compression algorithms deployed in the datacenter, including Simple9 [6], LZ4 [25], Snappy [49], Run Length Encoding (RLE) [103], and Varint and Group Varint [29]. Many compression algorithms typically represent a trade-off between compression efficiency and performance, where compression ratio needs to be sacrificed to reduce the amount of work done during compression and decompression.

In the FPGA context, this means the decompression module needs to be fast enough, or at least small enough to fit enough parallel units in the target FPGA chip. However, this is difficult to achieve with complex compression algorithms like DEFLATE, which is used for GZip. A best-effort GZip implementation on a Virtex Ultrascale+ FPGA can sustain less than 500 MB/s while consuming 10K LUTs. Similarly, an optimized implementation by Xilinx consumes more than 40K LUTs on a Virtex Ultrascale+ FPGA while delivering less than 2 GB/s [135]. While this is much faster than a software implementation of the algorithm, it is nowhere fast enough to saturate the nearest-neighbor computation unit. Even simpler algorithms like LZ4 do not do much better, with single-pipeline performance rarely rising over 2 GB/s [43].

Compression can be especially natural for storage, since the coarse sector- or page-granularity of access fits well with the grouped operations of most compression algorithms, which can be aligned to page boundaries. For example, LSH on secondary storage must already deal

with page granularities, and generality is not hurt by introducing page-aligned compression.

4.2.3 Near-Storage Acceleration

Due to the very fast performance of modern PCIe-attached storage devices, as well as engineering cost, modern storage devices often have a discrepancy between the peak bandwidth attainable from the chips, and the host-side PCIe link provisioned for it. For SSDs available on the market, the difference is often 2× or larger [63]. As a result, if an application has very good access patterns into storage, the performance of the storage device may be limited by its host-side link, making suboptimal use of the storage fabric performance.

Near-storage processing, where some computation is offloaded to a computation unit on the storage device itself, is a practical solution to this problem. Modern SSDs handle very complex flash management functionalities, such as bad block management and garbage collection, on their flash controllers. Some classes of near-storage processing platforms opted to use the computation already available on the flash controller, since these resources are not always busy [72, 68, 38, 64]. While this approach can sometimes reduce the data rate in real time and remove the PCIe bottleneck, the computational capacity of embedded controllers are nowhere as fast as desktop or server CPUs, limiting their efficacy. Near-storage processing using reconfigurable hardware accelerators such as FPGAs is an attractive solution that is being actively explored, as FPGAs can deliver very high performance while maintaining a low power budget [50, 67, 61, 77, 110, 127, 108].

4.3 ZipNN Accelerator Architecture

Figure 4.1 shows the overall architecture of the ZipNN design. The FPGA-accelerated storage device is connected to a host server via PCIe. Each column of the dataset of interest is

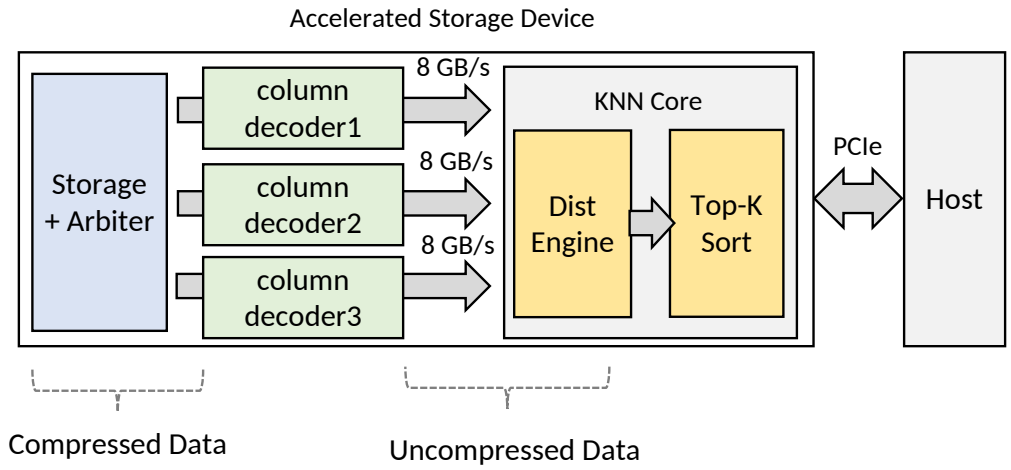


Figure 4.1: The Overall Architecture of ZipNN

stored separately in a compressed form, and are streamed independently into its own column decoder. All decoded streams are merged together, and are fed into the K-Nearest Neighbor (KNN) accelerator for processing.

4.3.1 Column Decoder Architecture

Each column decoder is equipped with three pipelined decoder accelerators for the following compression algorithms: Delta compression, Run length encoding, and pipelined Group Varint. ZipNN exposes an interface to the user to either enable or disable delta compression or run length encoding, according to the observed distribution of the column data. For example, if many consecutive values are known to be the same, run length encoding can be enabled. If the difference between consecutive values are known to be small, delta compression can be enabled to make Group Varint more effective.

In order to keep up with the full storage bandwidth, each pipeline stage must function at wire-speed on a wide pipeline. While designing wide, wire-speed delta and run length decoders is trivial, we discovered that is not the case for Group Varint. We describe the issues involved in Section 4.4, and present our design that invariably supports wide, wire-speed decoding.

4.3.2 K-NN Accelerator Architecture

Generally, the K-NN consists of two parts: (1) distance calculations and (2) top-k sorting. Distance calculation computes the distance between the query and each of the candidate neighbors in the dataset according to some distance metric, and top-k sorting incorporates this newly calculated results into a sorted list of top-k values. In this work, we present the design of a hardware K-NN accelerator on sparsely encoded data, which can support the very high bandwidth provided by the multiple column decoders.

We would like to emphasize that we are not arguing that our K-NN accelerator is the best possible design. The major contribution of this work is **not an optimized design of the K-NN accelerator**, but rather a demonstration of the benefits of hardware-optimized compression algorithm for this application.

Distance Calculation Engine

The ZipNN system expects the input dataset to be encoded in a sparse format, as it is a natural way to represent sparse data elements in a very high dimensional space. For example, the bag-of-words dataset in the UCI machine learning repository [8] represents each word occurrence as a 3-tuple consisting of a document ID, word ID, and occurrence count. Each document is represented as a variable-length stream of these 3-tuples, which can be thought of as three sparsely encoded columns.

We implement the cosine similarity distance metric, as it is a popular and high-functioning distance metric for high-dimensional datasets [94], although it has a much higher complexity than simple hamming distance. Since cosine similarity returns a larger value for similar pairs of data, top-k now must keep track of the k elements with the largest cosine similarity.

Figure 4.2 shows the internal architecture of the distance calculator engine. In order to

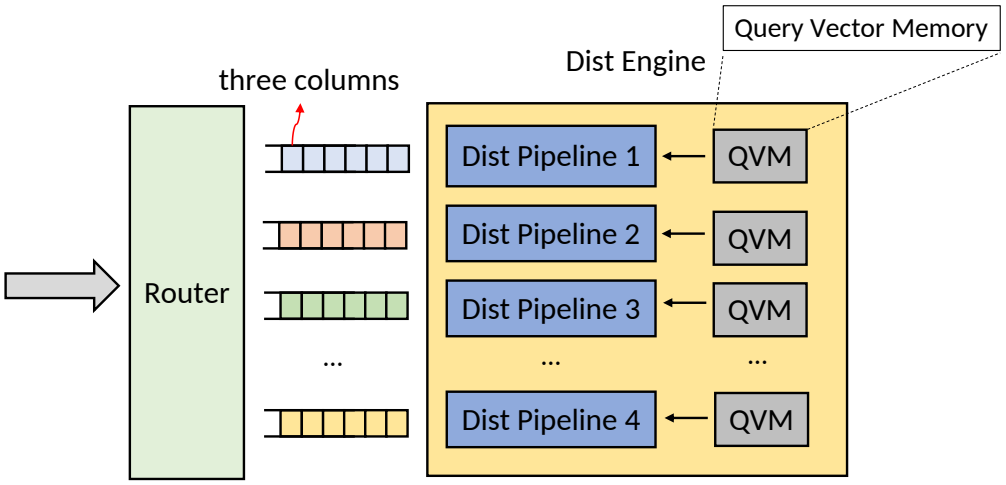


Figure 4.2: The architecture of cosine similarity calculation engine

achieve high performance, it contains multiple calculation pipelines working in parallel. Each pipe is connected to a QVM (Query Vector Memory) module, which stores a separate copy of the query vector. In order to support parallelism across different queries, each QVM can be configured to have a different value. Because the dataset consists of sparsely encoded data elements of variable length, the accelerator also includes a router before the distance calculators which identifies the delineation between each data element and sends it to an idle engine.

Each element of the input stream consists of a tuple, which would be a 3-tuple in the bag-of-words example. Meanwhile, the entry of a QVM contains multiple 2-tuple: <word id, frequency>. At each cycle, a distance calculator pipeline can accept 8 3-tuple input, or a 2-tuple entry from the QVM, or both. Since all data is sorted within a sparsely encoded data element as well as within a query, the distance calculator pipeline can use a comparator behaving like a merge sorter to quickly compare each 3-tuple and 2-tuple. Because the data ingestion rate of single distance calculator is limited by the 3-tuple datapath of the input data, ZipNN uses multiple calculator pipelines to achieve wire-speed performance, as seen in Figure 4.2.

Top-K Sorting Engine

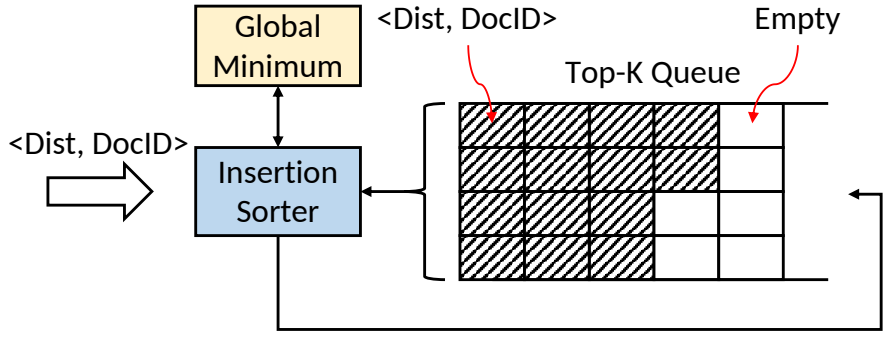


Figure 4.3: Internal architecture of the top-k sorting engine

ZipNN uses simple insertion sorting to maintain the top-k values. Figure 4.3 shows the architecture of the insertion sort accelerator. A wide tuple of the current top K <distance, document> pairs are stored in a buffer queue implemented using on-chip Block RAM. Every time a new calculated distance needs to be inserted into the buffer, the insertion sorter does a full sweep of the data currently in the queue, in order to insert the new data in a sorted location. Compared tuples are re-inserted into the queue to maintain ordering.

The width of the buffer queue must be wide, in order to reduce the number of cycles required by the insertion sorter to scan through the entire buffer. If, say, the width of the queue is four elements, we only need up to $K/4$ cycles to scan the buffer as opposed to K. The fact that each tuple is sorted also simplifies scanning logic. Assuming the queue is sorted in descending order, if the newly calculated distance is smaller than the smallest value of the tuple, the whole tuple can simply be skipped without further processing.

In order to avoid the multi-cycle latency of scanning the buffer queue for every distance calculation, we maintain a separate register *Global Minimum* as a further optimization. The global minimum register is updated with new input data, and if the newly calculated distance is smaller than the global minimum while the buffer queue is already full, the new data can be discarded without going through the costly process of scanning the buffer. As the data size relative to K became larger, the ratio of new data hitting the top-k buffer decreases,

effectively hiding the performance impact of insertion sort. To efficiently smooth out the performance impact of the rarely occurring insertion sort, we place a large on-chip Block RAM buffer between the distance calculator engine and the insertion sorter, so that distance calculation can continue uninterrupted even when a small number of insertion sort must happen.

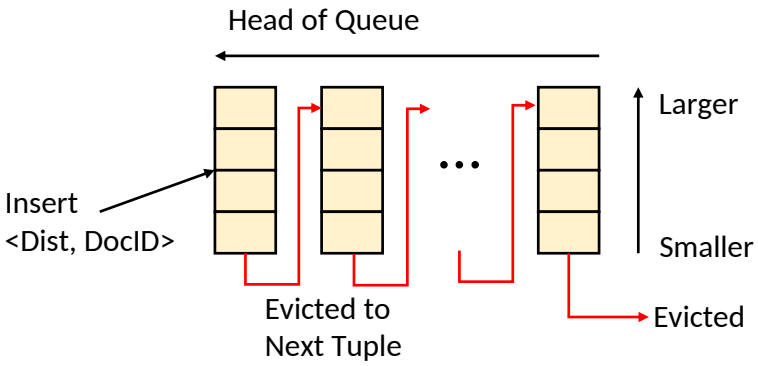


Figure 4.4: FIFO walking to insert a new data element into the top-k buffer

During insertion sort, once we discover the location where the new distance should be inserted, we may need to insert the new value into an already full tuple. Figure 4.4 shows this process when the FIFO is full. If the new value is inserted into the current tuple, all elements in the tuple smaller than the new value is shifted down by one spot. The smallest value in the current tuple is evicted to the next tuple, which we will process at the next cycle. This can continue until the very last element in the queue is evicted and discarded.

4.4 Hardware-Efficient Integer Stream Compression

4.4.1 Choice of Compression Algorithms

The choice of compression algorithm typically involves a trade-off between compression efficiency and performance. For example, an algorithm optimized for compression efficiency, such as DEFLATE, will achieve much lower performance compared to a performance-

optimized algorithm such as LZ4, even with FPGA acceleration [76, 42]. However, even FPGA-accelerated implementations of performance-optimized compression algorithms typically report sub-GB/s performance, and are not fast enough to support high-throughput accelerators and NVMe storage devices.

In order to achieve compression and decompression performance fast enough to move the performance bottleneck away from the storage device, we have chosen to use a simple but effective compression algorithm based on Group Varint [29]. Since a large portion of high-dimensional data is streams of integer values representing indices such as dimension index, we have chosen an algorithm optimized for it. We also augment group varint with delta compression and run-length encoding based on the observed data distribution for further compression efficiency. Our evaluation shows it works well with multiple real-world datasets, including the bag-of-words UCI machine learning repository [8], MovieLens 25M rating [53], as well as the Stack Overflow temporal network [100].

All of the above datasets consists of three columns, and we instantiate three column decoders in ZipNN to handle it. The three decoders are configured to be optimized for each column's data distribution.

4.4.2 Optimizing Heterogeneous Decoders to Data Patterns

All three compression algorithms used in ZipNN are very sensitive to the data distribution for effective compression. Delta encoding stores the difference between each value and the previous value in the stream, making it a good fit if the stream is in incrementally ascending order with small deltas. Delta encoding does not reduce the encoded size by itself, but is useful to transform the data stream to better fit the other two compression algorithms. Run length compression encodes a repeated sequence (a *run*) of values as a 2-tuple <value, count> and stores the 2-tuples instead of the original run. The count represents the length of

the repeated sequence, where a single symbol has a count of 1. Run length compression can achieve effective compression as long as the run length is longer than two. Varint encoding removes zero bytes from the MSB, and only encodes the nonzero LSB bytes, coupled with a two-bit header for a 4-byte integer. Obviously only *small positive* integers can benefit from this compression. When processing 32-bit integers, numbers larger than 2^{24}, as well as all negative numbers cannot benefit from it because they use the full 32 bit. In this case, the extra header overhead can actually increase the encoded data size.

ZipNN allows the user to observe the data distribution of each column and configure each column decoder to best match the patterns of its column. For example, the first column, which is the document/user/vertex ID column, consists of long stretches of identical IDs. This column is very efficiently compressed via a sequence of delta compression, runlength compression, and then Group Varint compression. Delta compression can make each value smaller, resulting in both elements of the run length compression output's 2-tuple (<value,length>) typically small enough to be effectively encoded by varint.

The second column, the word/movie/vertex ID column, consists of stretches of values increasing by small amounts, which is a good fit with delta compression pipelined with Group Varint. The semantics of the last column differ across datasets, and for some datasets like the movielens rating dataset, there is not much integer compression can do. As a result, the third column, is simply compressed via Group Varint, for better or for worse.

In our current design of ZipNN, this decision of compression algorithms must be made by the user based on human observation and manual statistical analysis. One potential future research topic is to automate this optimization process.

4.4.3 Limitations of a Hardware Group Varint Decoder

The baseline varint algorithm adds a 2-bit header per 32-bit integer specifying how many least-significant bytes in the integer has nonzero values. Group Varint is an improvement which reduces the overhead of header processing, by grouping 4 header-value pairs into one, making each header group a nicely aligned one byte in size. Single-thread software implementation of Group Varint have reported multi-GB/s of performance, especially when using SIMD extensions of modern CPUs [79].

Unfortunately, it is tricky to achieve high Group Varint performance on an FPGA, due to its relatively low clock frequency, as well as the high cost of variable-width shifters. Shifting a value by a variable width defined by a runtime variable is a costly operation in the FPGA fabric [13], which may significantly bring down clock speed. To remedy this, a *pipeline shifter* is often used, which can have multi-cycle latency but wire-speed throughput. Because the input bitstream must be shifted by a variable amount in order to decode the headers of successive groups, each header-payload pair may take multiple cycles to decode. FPGA accelerators are typically clocked in the orders of hundreds of MHz, compared to multi-GHz speeds of CPUs and GPUs. As a result, multi-cycle decoding of Group Varint results in an unacceptably slow performance.

4.4.4 Pipelined Group Varint

In order to solve this issue, we propose a new group varint design called "Pipelined Group Varint (PGV)", which introduces an even larger group size for hardware efficiency. Headers are grouped into large enough units that their payload can fill the width of the target datapath. Because our accelerator design had a 256-bit datapath, headers are grouped into 16-bit units (such a 16-bit unit is called a header group). These header groups are in turn grouped into units of N, which we call a "header chunk". Each 16-bit header corresponds

to 8 32-bit integer payloads, with a total size of 256 bits. This group of N payloads groups is called a "data chunk". We call a pair of "header chunk + data chunk" a *section*, and a compressed bitstream is composed of a stream of sections. The value of N could be 8, 16, 32 or more depending on the performance requirement, which we describe in Section 4.4.5.

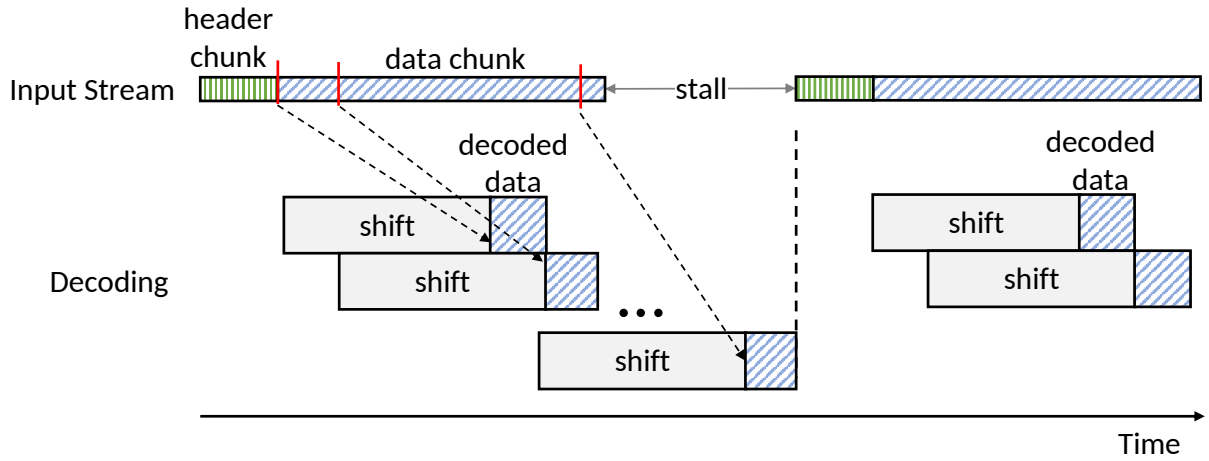


Figure 4.5: Pipelined Group Varint decoding across sections

This data layout allows us to read a large number of headers per cycle. For example, decoder implementation can accept a 512 bit input data per cycle, so at one cycle at most 32 8x2-bit headers (or 32 header groups) can be read and their corresponding payload sizes calculated. Once we process this one header chunk, we can decode 32x8 4-byte integers without having to wait for the results from the pipeline shifter, because we can instantly calculate the offset of the next encoded payload from the header values. With the header group size of 16, we can decode 8 integers, or 256-bits of decoded data coming out every cycle. As a result, a group of N payloads (i.e., a data chunk) needs N cycles to be decoded. Figure 4.5 shows the decoding process of Pipeline Group Varint. However, the Figure also shows that although this new design improves the decoding efficiency, it is still not ideal because it still has to wait for the shifter whenever it starts a new section.

4.4.5 Fully Pipelining Hardware Group Varint

Note that each header group corresponds to enough decoded payload data to fill the datapath. As a result, a data chunk of a section can be decoded within N cycles if completely pipelined. For example, if the output datapath is 256 bits as described before, a data chunk of size N payloads will emit a 256-bit decoded value every cycle, for N cycles. On the other hand, since each encoded integer can occupy 1 to 4 bytes, the compressed 8 integer in a group will occupy 8 to 32 bytes. For each header chunk, its payload size (i.e., data chunk size) will range from $8 \times N$ to $32 \times N$ bytes. If we design our decoder to have a wider input datapath, say, 512 bits, it will take the decoder $N/8$ to $N/2$ cycles to read the data chunk.

We can take advantage of the gap between the time it takes to read the encoded data (*reading window*), and the time it takes to emit the decoded data (*decoding window*). Figure 4.6 shows the relationship between these two time scales. We call the time gap between reading and decoding the *slack period*. As long as we can locate the start position of the next header within the slack period, we can avoid stalling and always keep the output datapath full.

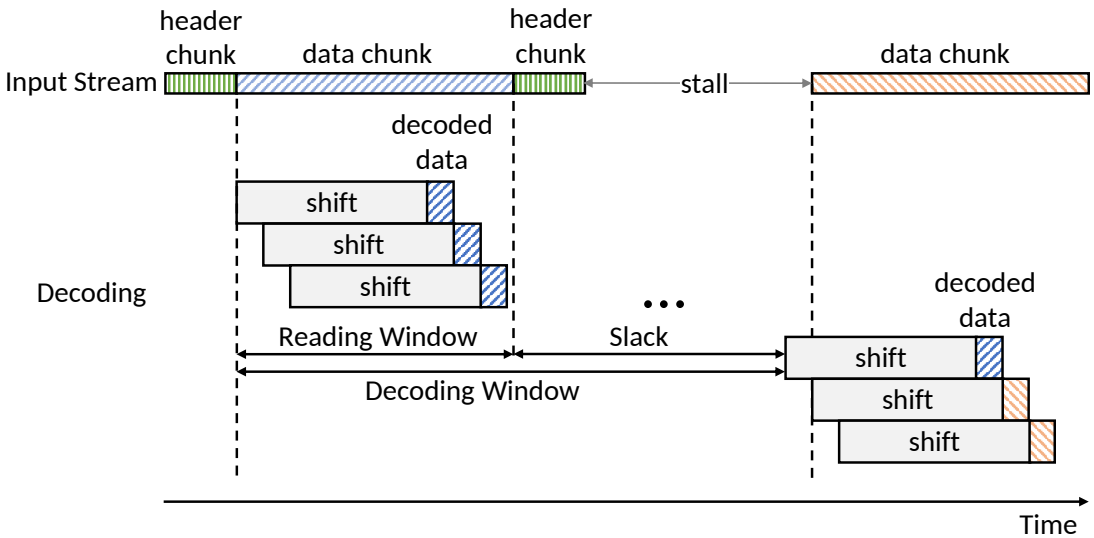


Figure 4.6: Optimally pipelined Group Varint using a lookahead buffer

In order to achieve this, we create a small look-ahead queue in the decoder so that while decoding is happening, we can keep reading successive section data looking for the next

header chunk. As soon as a header chunk is read, the offset of the next header chunk can be calculated by adding all $N \times 8$ 2-bit headers together. In our implementation, we hierarchically sum $N \times 8$ 2-bit headers using a three-level pipelined adder tree that has a latency of 3 cycles. We also need *reading window* cycles to reach the 512-bit word which has the next header chunk, after which we need to pipeline-shift it to get the next header. Since pipeline-shifting 512 bits at byte granularity can take $log_2 64$, or 6 cycles, the maximum latency for obtaining the header is $N/2 + 9$ cycles. As long as this is less than the decoding window of N cycles, our decoder will always have wire-speed throughput. In this situation, the N should be larger than 18. Also, the larger the N, the longer the slack period. In this paper we use a N of 32.

4.5 Performance Evaluation

4.5.1 Implementation Details

We use the MIT BlueDBM [67] platform to evaluate our system. The BlueDBM platform is equipped with 1 TB of NAND-flash storage, capable of delivering 2.4 GB/s per node. The flash storage is augmented with a Xilinx VC707 FPGA development board for application acceleration, which is also equipped with 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 has a maximum bandwidth of 4 GB/s duplex.

This device was a perfect fit to emulate our target storage device. It not only provided an actual, large-capacity SSD over a PCIe connection, but also provided fast enough on-board DRAM to emulate faster internal bandwidth compared to the PCIe link. The on-board DRAM bandwidth is over 2.5× faster than the PCIe connection, which is similar to the performance differential seen in the Samsung accelerated storage device [63].

Our prototype FPGA implementation of ZipNN runs on a 250 MHz clock. Each decompressor pipeline consumed 40,490 LUTs, about 13% of the VC707's FPGA. Removing the runlength decoder at compile time for columns that did not use it reduces the LUT use to 29,513, or about 10%. Such low space overhead leaves the majority of the chip for useful acceleration.

4.5.2 Benchmark And Configuration Details

Table 4.1 presents the datasets we used to evaluate ZipNN. We have tested seven high-dimensional datasets from three sources, the bag-of-words dataset of the UCI machine learning repository [8], MovieLens 25M rating [53], as well as the Stack Overflow temporal network [100]. The three sources have distinct characteristics, each being bag of words, recommendations, and a graph. The datasets span a wide range of sizes, compression ratio, as well as the size of each multi-dimensional data element, supporting a comprehensive evaluation of our system.

Dataset	Documents	Encoded Size (Uncompressed)	Tuples per Document
enron	39,861	42.46 MB	160.55
kos	3,430	4.04 MB	553.93
nips	1,500	8.54 MB	311.80
nytimes	300,000	0.77 GB	333.33
pubmed	8,200,000	1.5 GB	89.02
movielens	162,000	300 MB	154.32
stackoverflow	2,601,977	435 MB	13.92

Table 4.1: Datasets used to evaluate ZipNN

For the top-k sorter, we have chosen a K value of 128, according to the guideline in [54]. We have also tested much smaller and larger numbers of K, spanning from 64 to 512, with no significant difference in performance. We have tested the system using a set of random queries selected from the dataset.

4.5.3 Compression Effectiveness

Compression Ratio

Figure 4.7 shows the compression efficiency of our column compression methods, and a comparison against gzip. The figure presents the compression ratios of the individual columns, followed by the total compression ratio, and the compression ratio of gzip. Thanks to the compression accelerators configured for affinity to data distribution, our column compression scheme manages to achieve considerable compression, within 30% of the compression-optimized gzip algorithm. We think this is a completely worthwhile trade-off, considering the orders-of-magnitude performance improvement compared to the best-effort gzip decompressor accelerator on a comparable FPGA [76]. Interestingly, for the Stack Overflow dataset our encoder even did slightly better than gzip.

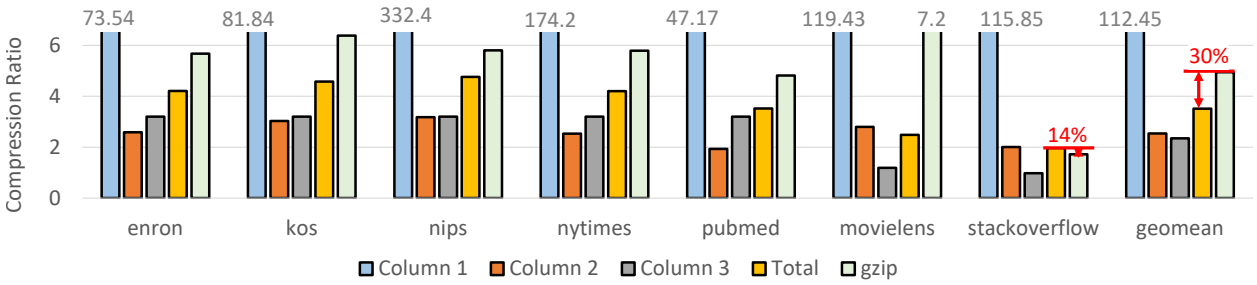


Figure 4.7: Per-column and total compression ratios

Decompressor Accelerator Performance

Because every part of the column decoder accelerator was designed to have non-blocking wire-speed performance, each of the column decompressors was invariably able to deliver full throughput, which is 8 GB/s on our configuration of 256-bit datapath running at 250 MHz. As a result, the three column decoders are able to provide a total of **24 GB/s** of decompressed throughput across the total dataset, at which point the question becomes whether the backing storage can keep up with it.

For context, this is much faster than the best-effort CPU implementation of Group Varint [78] using SIMD extensions. While software can achieve parallelism with more cores, the small footprint of the hardware decoder means that the hardware decoder can also scale with higher requirements.

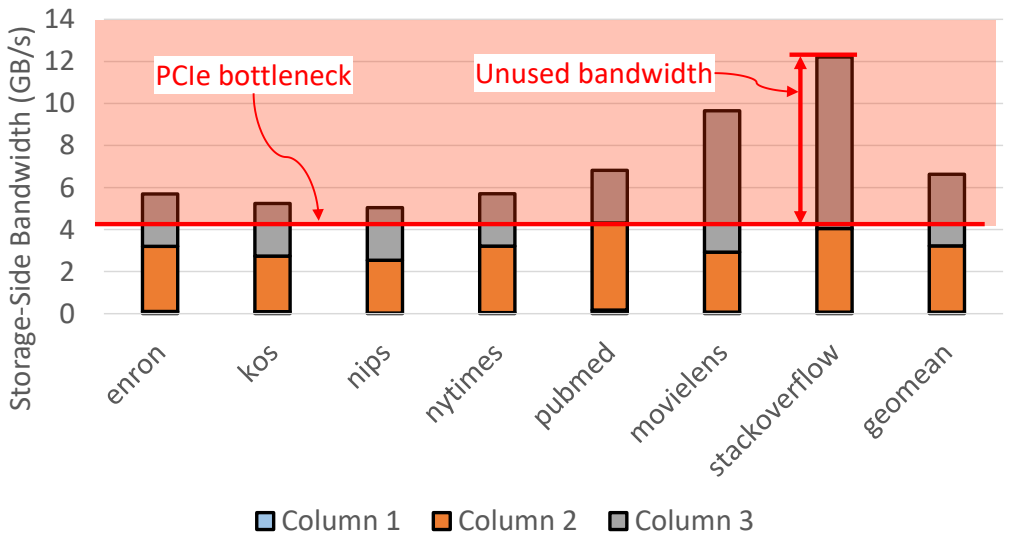


Figure 4.8: Storage bandwidth required to support wire-speed decompression exceeds the PCIe bandwidth.

An interesting factor to consider is how much backend storage bandwidth is required to support this wire-speed decoding, and whether a typical SSD is capable of supporting it. Figure 4.8 shows the throughput required of the backend storage in terms of compressed data, in order to support wire-speed decompression bandwidth of 8 GB/s. The figure presents the backend throughput required, broken down into each of the columns. Compared even to the ideal performance attainable by a modern NVMe SSD storage connected over a PCIe Gen3 x4, it can be seen that the backend storage does not have enough throughput to support wire-speed decompression.

This means the 3× compression ratio **translates directly to 3× effective storage bandwidth**, and that we can also achieve further storage performance by using a near-storage architecture and take advantage of the higher internal bandwidth.

4.5.4 K-NN Accelerator Performance

Of the two components in the K-NN accelerator, the distance calculation module implementing cosine similarity is wire-speed via parallelized distance calculation engines. As a result, the performance of the K-NN accelerator performance depends on the number of times insertion sort must be performed. If the data organization is extremely friendly, for example, if the distances calculated are continuously decreasing, vast majority of data elements will not require insertion sort because it will be filtered out by the global minimum register. In this situation, the whole accelerator will achieve wire-speed. Since insertion sort into a top-k buffer where K is 128 requires 32 cycles, a ratio that is less than $1/32 = 0.031$ would on average not cause stalls, supporting wire-speed performance. Figure 4.9 shows the ratio of data elements which incur insertion sort. It can be seen that the ratio of data elements requiring insertion sort is very small, especially for larger datasets, with a geomean of 0.03. This is small enough to support wire-speed on average.

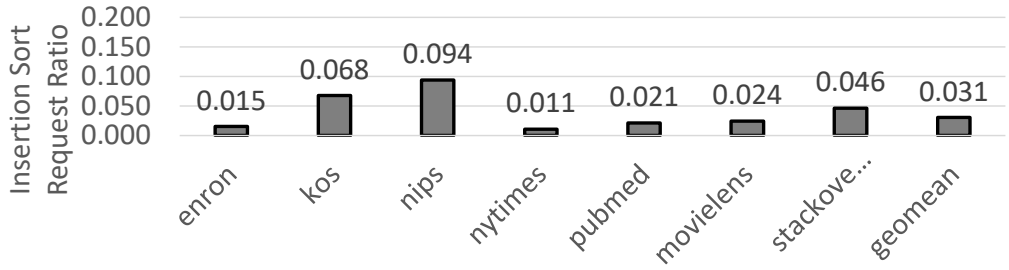


Figure 4.9: Ratio of data elements requiring insertion sort. Note the range of Y is 0 to 0.2

4.5.5 End-To-End Application Performance

With BlueDBM Flash

We first evaluated the performance difference between ZipNN, and the same KNN accelerator without compression support(**Acc**). The results can be seen, normalized against our best-effort single-thread, *in-memory* software implementation in Figure 4.10. The software

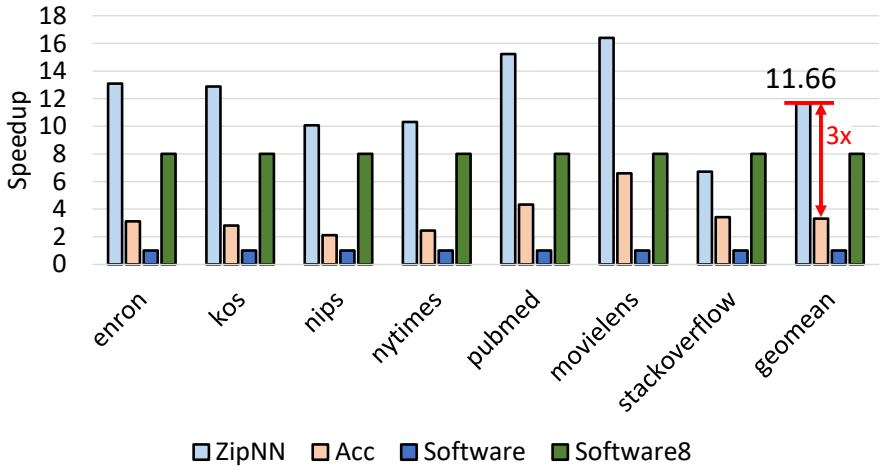


Figure 4.10: Normalized performance of ZipNN, using BlueDBM storage

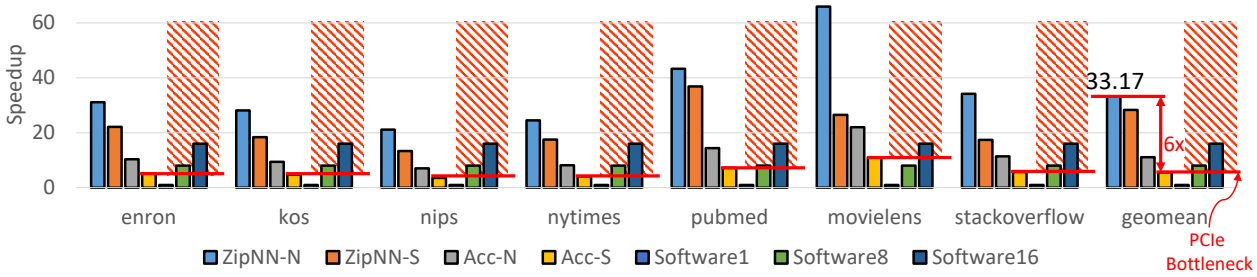


Figure 4.11: Normalized performance of ZipNN using emulated, high-performance storage. Ideally scaled software performance is also limited by PCIe.

implementation was hand-written, compiled using gcc with `-O3` optimizations. Software8 is an upper limit projection assuming ideal performance scaling.

In this scenario, both our KNN accelerator, as well as the decompressor array were capable of much higher speed than the 2.4 GB/s the BlueDBM flash could support. As a result, the increase in effective bandwidth as a result of compression were directly reflected to performance, and ZipNN was able to achieve over 3× performance compared to Acc, over 11× faster than single-thread software. However, because both hardware modules were capable of much more than the flash bandwidth, we are operating under the capacity of the ZipNN hardware accelerators.

With Emulated Flash Using DRAM

A more interesting scenario is when we emulate a much faster storage device using the on-board DRAM. Table 4.2 shows the system configurations evaluated. The backend storage performance of standalone accelerators was emulated by limiting the memory bandwidth to 4 GB/s, which is the upper limit bandwidth of a PCIe Gen3 x4 link, popular with contemporary PCIe SSDs. The near-storage systems were configured to support up to 8 GB/s. This performance differential is similar to the real-world numbers from Samsung [63].

Name	Description
ZipNN-N	ZipNN with compression, near-storage
ZipNN-S	ZipNN with compression, standalone
Acc-N	Non-compressed accelerator, near-storage
Acc-S	Non-compressed accelerator, standalone
Software$N$	N-thread software

Table 4.2: System configurations evaluated

Figure 4.11 shows the performance comparisons between these systems, normalized to a single-thread software implementation. Compared to Acc-S, which is completely bottlenecked by the projected PCIe bandwidth, the near-storage ZipNN-N is able to achieve 6× performance improvement. Even the standalone ZipNN-S device is able to achieve 4× the performance compared to Acc-S. However, the performance of ZipNN-S is limited by the capability of the backend storage as described in Figure 4.8. These results show the definitive benefits of both our compression scheme, as well as the near-storage configuration.

ZipNN-N achieves almost 30× performance compared to a single-thread software implementation. While Figure 4.11 shows multithreaded software performance exceeding storage bandwidth, in reality software performance will also be limited by the storage performance limitations.

4.6 Summary

We have presented ZipNN, a near-storage accelerator which uses low-overhead, hardware-optimized integer stream compression algorithms to remove the storage access bottleneck from a high-dimensional similarity search application. We have shown that by modifying the Group Varint compression algorithm to be better suited for the FPGA architecture, coupled with other low-overhead algorithms such as delta and run length encoding, ZipNN is able to achieve wire-speed data throughput while significantly reducing the performance pressure put on the backend storage. Thanks to these improvements, ZipNN was able to sustain wire-speed throughput in an FPGA accelerator with the storage throughput available to a near-storage accelator, demonstrating 6× the performance compared to a standalone FPGA accelerator without compression, and almost 30× the performance compared to a single-thread software implementation. In summary, ZipNN demonstrates the benefits of high-performance compression coupled with a near-storage acceleration architecture for high-throughput applications.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Modern high-performance hardware accelerators have been widely used to offload computing-intensive workloads from CPU. However, the whole system performance is often limited by the host-accelerator bandwidth, especially when large scale data sets have to travel back and forth between host and accelerator frequently. The data movement overhead gradually becomes the bottle of the system.

This dissertation proposes compression-based solution to overcome the above issue. By transferring compressed data between host and accelerator, the data movement overhead is reduced significantly. Implementing such an idea is not easy and must solve multiple challenges: how to implement high-efficiency compressors/decompressors while using small chip area and hardware resources, how to design high performance computation engine and let it interact with compressors/decompressors, and more.

We propose that a class of compression algorithms, with minor modifications made with

awareness of how the algorithms will be realized in hardware, can achieve sufficiently high performance and efficiency to remove the communication bandwidth issue. We first explored the hardware design and implementation of high-efficiency floating point compression algorithms, then integrate them with high performance scientific computation engines. We also explored hardware-optimized compression algorithms for integer streams, and applied them to a near-storage nearest neighbor search accelerator. We conducted intensive evaluation of both the compressor/decompressor and the whole compressor-computation system. The evaluation proves our idea is effective in removing the overhead of data movement on PCIe between host and FPGA accelerator.

The proposed framework consumes very few hardware resources on high-end FGPA boards and hence is very promising in large-scale computing platform (e.g., in scientific computing centers). In addition, it is flexible and can support combinations of various compressors/decompressors and computation engines depending on the real needs and the available hardware resources.

5.2 Future Work

Though this system proposed in this dissertation solves an important problem in the scientific computing accelerator area, there is still room for further improvement. Some potential exploration directions are as follows:

- Explore similar hardware-optimized modifications of more compression algorithms:
  As the compression algorithms we explored are datatype-specific, applications that require different data types will inevitably require different compression algorithms. Our goal in this regard is twofold: Explore and optimize more datatype-specific algorithms to cover more imporant application scenarios, and explore similar optimizations for

more general-purpose lossless compression algorithms, such as LZMA and DEFLATE.

- Supporting multiple concurrent computation engines:

  Currently, the system only supports one computation engine due to limited resources. With a larger board, it is possible to deploy multiple computing cores that can share the compressors. With this, a compressor may maximize its throughput by receiving interleaved data blocks from different application streams.

- Programmable, software-defined accelerators:

  We believe our compression-based approaches can dramatically reduce the cost of hardware acceleration on scalable problems. However, the accessibility of application-specific accelerators using FPGAs is limited not only by the accelerator scalability, but also the difficulty of programming the accelerators themselves. To address this issue, we are actively researching the software-defined accelerators. This line of research is guided by a wealth of insight and design patterns we have obtained while exploring the various accelerators presented in this dissertation. Our goal is to provide a high-level programming interface which can be used to easily generate and evaluate application-specific accelerators.

Bibliography

[1] `https://computing.llnl.gov/projects/floating-point-compression/related-projects`.

[2] `https://sdrbench.github.io/`.

[3] `https://wiki.intel-research.net/FPGA.html`.

[4] S. R. Agrawal, C. M. Dee, and A. R. Lebeck. Exploiting accelerators for efficient high dimensional similarity search. *ACM SIGPLAN Notices*, 51(8):1–12, 2016.

[5] M. S. B. Altaf and D. A. Wood. Logca: A high-level performance model for hardware accelerators. *ACM SIGARCH Computer Architecture News*, 45(2):375–388, 2017.

[6] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

[7] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[8] A. Asuncion and D. Newman. Uci machine learning repository, 2007.

[9] P. Bailey, J. Myre, S. D. Walsh, D. J. Lilja, and M. O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *2009 international conference on parallel processing*, pages 550–557. IEEE, 2009.

[10] A. H. Baker, D. M. Hammerling, S. A. Mickelson, H. Xu, M. B. Stolpe, P. Naveau, B. Sanderson, I. Ebert-Uphoff, S. Samarasinghe, F. D. Simone, et al. Evaluating lossy data compression on climate simulation data within a large ensemble. *Geoscientific Model Development*, 9(12):4381–4403, 2016.

[11] M. A. Bamakhrama, A. Arrizabalaga, F. Overman, J.-P. Smeets, K. van der Sommen, R. van der Vossen, and J. Wagensveld. Gpu acceleration of real-time control loops. *arXiv preprint arXiv:1902.08018*, 2019.

[12] U. Bandara and G. Wijayarathna. A machine learning based tool for source code plagiarism detection. *International Journal of Machine Learning and Computing*, 1(4):337, 2011.

[13] M. J. Beauchamp, S. Hauck, K. D. Underwood, and K. S. Hemmert. Architectural modifications to enhance the floating-point performance of fpgas. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):177–187, 2008.

[14] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix multiplication on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1033–1051, 2001.

[15] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[16] D. J. Berndt and J. Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. Seattle, WA, USA:, 1994.

[17] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR)*, 33(3):322–373, 2001.

[18] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys (CSUR)*, 33(3):322–373, 2001.

[19] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[20] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 398–409, 1995.

[21] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 13(11):1105–1123, 2002.

[22] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[23] Y. Chi, J. Cong, P. Wei, and P. Zhou. Soda: stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[24] A. E. Coding. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on signal processing*, 41(12), 1993.

[25] Y. Collet et al. Lz4: Extremely fast compression algorithm. *code.google.com*, 2013.

[26] J. Cong, P. Li, B. Xiao, and P. Zhang. 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):407–418, 2015.

[27] M. Daga, A. M. Aji, and W.-c. Feng. On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing. In *2011 Symposium on Application Accelerators in High-Performance Computing*, pages 141–149. IEEE, 2011.

[28] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick. Auto-tuning stencil computations on multicore and accelerators. *Scientific Computing on Multicore and Accelerators*, pages 219–253, 2010.

[29] J. Dean. Challenges in building large-scale information retrieval systems. In *Keynote of the 2nd ACM International Conference on Web Search and Data Mining (WSDM)*, volume 10, 2009.

[30] G. Deest, N. Estibals, T. Yuki, S. Derrien, and S. Rajopadhye. Towards scalable and efficient fpga stencil accelerators. 2016.

[31] G. Deest, T. Yuki, S. Rajopadhye, and S. Derrien. 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)*, pages 1–8. IEEE, 2017.

[32] J. Dennis. *Data Compression of Climate Simulation Data*, 2013.

[33] P. Deutsch. Deflate compressed data format specification version 1.3. 1996.

[34] S. Di and F. Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

[35] S. Di and F. Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

[36] J. Diffenderfer, A. L. Fox, J. A. Hittinger, G. Sanders, and P. G. Lindstrom. Error analysis of zfp compression for floating-point data. *SIAM Journal on Scientific Computing*, 41(3):A1867–A1898, 2019.

[37] J. Diffenderfer, A. L. Fox, J. A. Hittinger, G. Sanders, and P. G. Lindstrom. Error analysis of zfp compression for floating-point data. *SIAM Journal on Scientific Computing*, 41(3):A1867–A1898, 2019.

[38] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[39] K. Dohi, K. Fukumoto, Y. Shibata, and K. Oguri. 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)*, pages 1–6. IEEE, 2013.

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

[41] T. Endo and G. Jin. Software technologies coping with memory hierarchy of gpgpu clusters for stencil computations. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 132–139. IEEE, 2014.

[42] J. Fang, J. Chen, Z. Al-Ars, P. Hofstee, and J. Hidders. A high-bandwidth snappy decompressor in reconfigurable logic: work-in-progress. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, page 16. IEEE Press, 2018.

[43] J. Fang, Y. T. Mulder, J. Hidders, J. Lee, and H. P. Hofstee. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 29(1):33–59, 2020.

[44] R. Filgueira, D. E. Singh, A. Calderón, and J. Carretero. Compi: enhancing mpi based applications performance and scalability using run-time compression. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 207–218. Springer, 2009.

[45] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An open-source 100-gbps nic. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 38–46. IEEE, 2020.

[46] A. Fox, J. Diffenderfer, J. Hittinger, G. Sanders, and P. Lindstrom. Stability analysis of inline zfp compression for floating-point data in iterative methods. *SIAM Journal on Scientific Computing*, 42(5):A2701–A2730, 2020.

[47] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6. IEEE, 2008.

[48] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: Fast gpu-based implementations and application to high-dimensional feature matching. In *2010 IEEE International Conference on Image Processing*, pages 3757–3760. IEEE, 2010.

[49] Google. *snappy: A fast compressor/decompressor*, Accessed June, 2020.

[50] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, et al. Biscuit: A framework for near-data processing of big data workloads. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 153–165. IEEE Press, 2016.

[51] T. Hamada, K. Benkrid, K. Nitadori, and M. Taiji. A comparative study on asic, fpgas, gpus and general purpose processors in the o (nˆ 2) gravitational n-body simulation. In *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 447–452. IEEE, 2009.

[52] M. S. Hansen and T. S. Sørensen. Gadgetron: an open source framework for medical image reconstruction. *Magnetic resonance in medicine*, 69(6):1768–1776, 2013.

[53] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

[54] A. B. Hassanat, M. A. Abbadi, G. A. Altarawneh, and A. A. Alhasanat. Solving the problem of the k parameter in the knn classifier using an ensemble learning approach. *arXiv preprint arXiv:1409.0919*, 2014.

[55] M. A. Heroux, J. Carter, R. Thakur, J. Vetter, L. C. McInnes, J. Ahrens, and J. R. Neely. Ecp software technology capability assessment report. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2018.

[56] M. Q. Ho, C. Obrecht, B. Tourancheau, B. D. de Dinechin, and J. Hascoet. Improving 3d lattice boltzmann method stencil with asynchronous transfers on many-core processors. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–9. IEEE, 2017.

[57] N. Hübbe, A. Wegener, J. M. Kunkel, Y. Ling, and T. Ludwig. Evaluating lossy compression on climate data. In *International Supercomputing Conference*, pages 343–356. Springer, 2013.

[58] H. M. Hussain, K. Benkrid, and H. Seker. An adaptive implementation of a dynamically reconfigurable k-nearest neighbour classifier on fpga. In *2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 205–212. IEEE, 2012.

[59] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613, 1998.

[60] Intel. *AN 870: Stencil Computation Reference Design*, 2018.

[61] Z. István, D. Sidler, and G. Alonso. Caribou: intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11):1202–1213, 2017.

[62] P. Jakovits and S. N. Srirama. Evaluating mapreduce frameworks for iterative scientific computing applications. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 226–233. IEEE, 2014.

[63] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[64] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[65] V. Joseph, N. Chalapathi, A. Bhaskara, G. Gopalakrishnan, P. Panchekha, and M. Zhang. Correctness-preserving compression of datasets and neural network models. In *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 1–9. IEEE, 2020.

[66] S.-W. Jun, C. Chung, et al. Large-scale high-dimensional nearest neighbor search using flash memory with in-store processing. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2015.

[67] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, et al. Bluedbm: An appliance for big data analytics. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13. IEEE, 2015.

[68] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park. Enabling cost-effective data processing with smart ssd. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–12. IEEE, 2013.

[69] N. Katayama and S. Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. *ACM Sigmod Record*, 26(2):369–380, 1997.

[70] R. Kobayashi, S. Takamaeda-Yamazaki, and K. Kise. Towards a low-power accelerator of many fpgas for stencil computations. In *2012 Third International Conference on Networking and Computing*, pages 343–349. IEEE, 2012.

[71] E. Konstantinidis and Y. Cotronis. A practical performance model for compute and memory bound gpu kernels. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 651–658. IEEE, 2015.

[72] G. Koo, K. K. Matam, H. Narra, J. Li, H.-W. Tseng, S. Swanson, M. Annavaram, et al. Summarizer: trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–231. ACM, 2017.

[73] S. Kulkarni and R. Orlandic. High-dimensional similarity search using data-sensitive space partitioning. In *International Conference on Database and Expert Systems Applications*, pages 738–750. Springer, 2006.

[74] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, et al. 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*, pages 317–330, 2019.

[75] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In *European Conference on Parallel Processing*, pages 366–379. Springer, 2011.

[76] M. Ledwon, B. F. Cockburn, and J. Han. Design and evaluation of an fpga-based hardware accelerator for deflate data decompression. In *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE)*, pages 1–6. IEEE, 2019.

[77] J. Lee, H. Kim, S. Yoo, K. Choi, H. P. Hofstee, G.-J. Nam, M. R. Nutter, and D. Jamsek. Extrav: Boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 10(12):1706–1717, 2017.

[78] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.

[79] D. Lemire, N. Kurz, and C. Rupp. Stream vbyte: Faster byte-oriented integer compression. *Information Processing Letters*, 130:1–6, 2018.

[80] C. Li, E. Chang, H. Garcia-Molina, and G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):792–808, 2002.

[81] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447. IEEE, 2018.

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

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

[84] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

[85] T. Liu, A. W. Moore, K. Yang, and A. G. Gray. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, pages 825–832, 2005.

[86] LLNL. Zfp dimension. `https://zfp.readthedocs.io/en/release0.5.5/issues.html`, 2019.

[87] LLNL. Zfp related projects. `https://computing.llnl.gov/projects/floating-point-compression/related-projects`, Accessed April 2020.

[88] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, T. Liu, et al. Understanding and modeling lossy compression schemes on hpc scientific data. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 348–357. IEEE, 2018.

[89] T. Lu, Q. Liu, X. He, H. Luo, E. Suchyta, J. Choi, N. Podhorszki, S. Klasky, M. Wolf, T. Liu, et al. Understanding and modeling lossy compression schemes on hpc scientific data. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 348–357. IEEE, 2018.

[90] N. Maruyama and T. Aoki. Optimizing stencil computations for nvidia kepler gpus. In *Proceedings of the 1st international workshop on high-performance stencil computations, Vienna*, pages 89–95, 2014.

[91] R. Membarth, F. Hannig, J. Teich, and H. Köstler. Towards domain-specific computing for stencil codes in hpc. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1133–1138. IEEE, 2012.

[92] A. Mitra. On finite wordlength properties of block-floating-point arithmetic. *International journal of signal processing*, 2(2):120–125, 2008.

[93] A. Mohamad. *Lattice Boltzmann Method*, volume 70. Springer, 2011.

[94] L. Muflikhah and B. Baharudin. Document clustering using concept space and cosine similarity measurement. In *2009 International Conference on Computer Technology and Development*, volume 1, pages 58–62. IEEE, 2009.

[95] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 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*, pages 1–13. IEEE, 2010.

[96] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell. Exploiting run-time reconfiguration in stencil computation. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 173–180. IEEE, 2012.

[97] M. F. Oberhumer. *LZO real-time data compression library*, 2017.

[98] K. Okina, R. Soejima, K. Fukumoto, Y. Shibata, and K. Oguri. Power performance profiling of 3-d stencil computation on an fpga accelerator for efficient pipeline optimization. *ACM SIGARCH Computer Architecture News*, 43(4):9–14, 2016.

[99] L. Orf. A violently tornadic supercell thunderstorm simulation spanning a quarter-trillion grid volumes: Computational challenges, i/o framework, and visualizations of tornadogenesis. *Atmosphere*, 10(10):578, 2019.

[100] A. Paranjape, A. R. Benson, and J. Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 601–610, 2017.

[101] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou. Tersecades: Efficient data compression in stream processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 307–320, Boston, MA, July 2018. USENIX Association.

[102] I. S. Pershin, V. D. Levchenko, and A. Y. Perepelkina. Performance limits study of stencil codes on modern gpgpus. *Supercomputing Frontiers and Innovations*, 6(2):86–101, 2019.

[103] D. Pountain. Run-length encoding. *Byte*, 12(6):317–319, 1987.

[104] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35, 2016.

[105] T. M. Rath and R. Manmatha. Word image matching using dynamic time warping. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.*, volume 2, pages II–II. IEEE, 2003.

[106] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *ACM SIGMICRO Newsletter*, 12(4):183–198, 1981.

[107] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach. Performance portable gpu code generation for matrix multiplication. In *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit*, pages 22–31, 2016.

[108] Z. Ruan, T. He, and J. Cong. {INSIDER}: Designing in-storage computing system for emerging high-performance drive. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 379–394, 2019.

[109] A. Said, W. A. Pearlman, et al. A new, fast, and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on circuits and systems for video technology*, 1996.

[110] B. Salami, G. A. Malazgirt, O. Arcas-Abella, A. Yurdakul, and N. Sonmez. Axledb: A novel programmable query processing platform on fpga. *Microprocessors and Microsystems*, 51:142–164, 2017.

[111] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):695–705, 2013.

[112] K. Sano, Y. Kono, H. Suzuki, R. Chiba, R. Ito, T. Ueno, K. Koizumi, and S. Yamamoto. Efficient custom computing of fully-streamed lattice boltzmann method on tightly-coupled fpga cluster. *ACM SIGARCH Computer Architecture News*, 41(5):47–52, 2014.

[113] K. Sano, O. Pell, W. Luk, and S. Yamamoto. Fpga-based streaming computation for lattice boltzmann method. In *2007 International Conference on Field-Programmable Technology*, pages 233–236. IEEE, 2007.

[114] K. Sano and S. Yamamoto. Fpga-based scalable and power-efficient fluid simulation using floating-point dsp blocks. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2823–2837, 2017.

[115] N. Sasaki, K. Sato, T. Endo, and S. Matsuoka. Exploration of lossy compression for application-level checkpoint/restart. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 914–922. IEEE, 2015.

[116] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Characterizing the impact of soft errors on iterative methods in scientific computing. In *Proceedings of the international conference on Supercomputing*, pages 152–161, 2011.

[117] T. Shimokawabe, T. Endo, N. Onodera, and T. Aoki. A stencil framework to realize large-scale computations beyond device memory capacity on gpu supercomputers. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 525–529. IEEE, 2017.

[118] S. K. Shukla, Y. Yang, L. N. Bhuyan, and P. Brisk. Shared memory heterogeneous computation on pcie-supported platforms. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4. IEEE, 2013.

[119] S. N. Srirama, P. Jakovits, and E. Vainikko. Adapting scientific computing problems to clouds using mapreduce. *Future Generation Computer Systems*, 28(1):184–192, 2012.

[120] I. Stamoulias and E. S. Manolakos. Parallel architectures for the knn classifier–design of soft ip cores and fpga implementations. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–21, 2013.

[121] G. Sun and S.-W. Jun. Zfp-v: Hardware-optimized lossy floating point compression. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 117–125. IEEE, 2019.

[122] L. G. Szafaryn, K. Skadron, and J. J. Saucerman. Experiences accelerating matlab systems biology applications. In *Proceedings of the Workshop on Biomedicine in Computing: Systems, Architectures, and Circuits*, pages 1–4, 2009.

[123] D. Tao, S. Di, Z. Chen, and F. Cappello. In-depth exploration of single-snapshot lossy compression techniques for n-body simulations. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 486–493. IEEE, 2017.

[124] D. Tao, S. Di, Z. Chen, and F. Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1129–1139. IEEE, 2017.

[125] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello. Optimizing lossy compression rate-distortion from automatic online selection between sz and zfp. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[126] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 563–576, 2009.

[127] M. Torabzadehkashi, S. Rezaei, V. Alves, and N. Bagherzadeh. Compstor: An in-storage computation platform for scalable distributed processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1260–1267. IEEE, 2018.

[128] T. Ueno, K. Sano, and S. Yamamoto. Bandwidth compression of floating-point numerical data streams for fpga-based high-performance computing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 10(3):1–22, 2017.

[129] H. M. Waidyasooriya, Y. Takei, S. Tatsumi, and M. Hariyama. Opencl-based fpga-platform for stencil computation and its optimization methodology. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1390–1402, 2016.

[130] S. Wang and Y. Liang. A comprehensive framework for synthesizing stencil algorithms on fpgas using opencl model. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[131] X. Wang, Y. Niu, F. Liu, and Z. Xu. When fpga meets cloud: A first look at performance. *IEEE Transactions on Cloud Computing*, 2020.

[132] A. W. Wegener. Block floating point compression of signal data, Oct. 30 2012. US Patent 8,301,803.

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

[134] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. Ross. Improving i/o forwarding throughput with data compression. In *2011 IEEE International Conference on Cluster Computing*, pages 438–445. IEEE, 2011.

[135] Xilinx. Gzip - github, 2020.

[136] Q. Xiong, R. Patel, C. Yang, T. Geng, A. Skjellum, and M. C. Herbordt. Ghostsz: A transparent fpga-accelerated lossy compression framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 258–266. IEEE, 2019.

[137] Y. Yan, A. F. Beldachi, R. Nejabati, and D. Simeonidou. P4-enabled smart nic: Enabling sliceable and service-driven optical data centres. *Journal of Lightwave Technology*, 38(9):2688–2694, 2020.

[138] N. Yang, Y. Lu, X. Yang, J. Huang, Y. Zhou, F. Ali, W. Wen, J. Liu, J. Li, and J. Yan. Genome wide association studies using a new nonparametric model reveal the genetic architecture of 17 agronomic traits in an enlarged maize association panel. *PLoS Genetics*, 10(9), 2014.

[139] J. Ye. Cosine similarity measures for intuitionistic fuzzy sets and their applications. *Mathematical and computer modelling*, 53(1-2):91–97, 2011.

[140] Y. Yu and S. T. Acton. Speckle reducing anisotropic diffusion. *IEEE Transactions on image processing*, 11(11):1260–1270, 2002.

[141] X. Zhou, K.-Y. Liu, P. Bradley, N. Perrimon, and S. T. Wong. Towards automated cellular image segmentation for rnai genome-wide screening. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 885–892. Springer, 2005.

[142] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420. IEEE, 2016.

[143] H. R. Zohouri, A. Podobas, and S. Matsuoka. 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*, pages 153–162, 2018.

[144] H. R. Zohouri, A. Podobas, and S. Matsuoka. High-performance high-order stencil computation on fpgas using opencl. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 123–130. IEEE, 2018.