Efficient Error-Bounded Lossy Compression for CPU Architectures

Griffin Dube*, Jiannan Tian†, Sheng Di‡, Dingwen Tao†, Jon C. Calhoun§, Franck Cappello‡
*Department of Computer Science, Northwestern University, Evanston, IL 60208
†Luddy School of Informatics, Computing, and Engineering, Indiana University, Bloomington, IN 47408
‡Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439
§Holcombe Department of Electrical and Computing Engineering, Clemson University, Clemson, SC 29634

***Abstract*—Modern HPC applications produce increasingly large amounts of data, which limits the performance of current extreme-scale systems. Lossy compression, helps to mitigate this issue by decreasing the size of data generated by these applications. SZ, a current state-of-the-art lossy compressor, is able to achieve high compression ratios, but its prediction/quantization methods contain RAW dependencies that prevent parallelizing this step of the compression. Recent work proposes a parallel dual prediction/quantization algorithm for GPUs which removes these dependencies. However, some HPC systems and applications do not use GPUs, and could still benefit from the fine-grained parallelism of this method. Using the dual-quantization technique, we implement and optimize a SIMD vectorized CPU version of SZ (vecSZ), and create a heuristic for selecting the optimal block size and vector length. We propose a novel block padding algorithm to decrease the number of unpredictable values along compression block borders and find it reduces the number of prediction outliers by up to 100%. We measure performance of our vecSZ against an CPU version of SZ using dual-quantization, pSZ, as well as SZ-1.4. Using real-world scientific datasets, we evaluate vecSZ on the Intel Skylake and AMD Rome architectures. vecSZ results in up to 32% improvement in rate-distortion and up to 15× speedup over SZ-1.4, achieving a prediction and quantization bandwidth in excess of 3.4 GB/s.**

***Index Terms*—lossy compression, compression, big data, vectorization, program optimization**

I. Introduction

As data produced by large scale scientific applications becomes larger, efficient management of application data in high-performance computing (HPC) systems is becoming increasingly important. Current petascale applications such as the Hardware/Hybrid Accelerated Cosmology Code (HACC) [1] can produce 21.2 petabytes of data when simulating 2 trillion particles for 500 time-steps. Large amounts of data are difficult or impossible to handle due to the limitations of I/O bandwidth and storage on modern HPC systems. One mechanism for coping with the massive amounts of data generated by these applications is through the use of data compression.

Data compression is broken down into two areas: lossless and lossy. Lossless compression reduces data size and exactly preserves the original data. However, it is only able to achieve limited compression ratios 1–4× on floating-point HPC datasets [2]. Lossy compression is able to achieve higher compression ratios than lossless compression by introducing error into data [3], [4]. Error-bounded lossy compression (EBLC) presents an attractive solution to the data reduction challenge because of its ability to achieve high compression ratios while guaranteeing the error introduced remains within a specified error bound. The ability to tune the level of loss in the data enables EBLC to be integrated into HPC applications and workflows, making them more efficient [5], [6].

SZ [7] is a popular EBLC algorithm seeing rapid development lately. SZ supports a variety of error bounding modes — e.g., absolute and relative error, peak signal-to-noise ratio (PSNR). Newer versions of SZ optimize the compression ratio and the compression/decompression bandwidth; however, to significantly improve the compression/decompression bandwidth, SZ must take advantage of accelerators. The CPU version of SZ is limited to coarse grain parallelism for its prediction and quantization process due to a read-after-write (RAW) dependency that prevents fine-grained parallelism necessary to perform single instruction multiple data (SIMD) operations. A GPU implementation of SZ, cuSZ [8], addresses this by introducing a dual-quantization (DQ) method that removes the RAW dependency, allowing for fine-grained parallelism in this compression step. However, DQ has not been explored for CPUs, resulting in unrealized performance.

In this paper, we investigate the performance of cuSZ's DQ technique when applied to CPUs for prediction and quantization of data, finding it obtains only a small fraction of peak CPU performance (<25%). Through rigorous use of performance models and analysis, we develop vecSZ. vecSZ applies fine-grain program optimization via auto and manual vectorization and thread-level parallelism via OpenMP. To dynamically select the best configuration for each dataset and CPU, we auto-tune for block size and vector length for the DQ technique, substantially improving the prediction and quantization bandwidth. To improve vecSZ's prediction accuracy on commonly misspredicted elements and subsequently improve the compressor's rate-distorion, we devise a novel padding scheme that dynamically sets the border boundary values used during prediction based on statistical properties of the dataset.

In this paper, we:

- Generate a Roofline performance model demonstrating the performance of the basic DQ algorithm only reaches up to 25% of peak on current CPU architectures;
- Leverage the Roofline performance analysis to develop and optimize vecSZ, a vectorized and threaded DQ prediction algorithm, increasing the prediction/quantization

bandwidth by 15.1×;
- Argument vecSZ with an auto-tuning framework to select the best vector length and compression block size across multiple time-steps; and
- Propose a novel compression block padding scheme for block borders, reducing the overall number of unpredictable values by as much as 100%, leading to a 32% improvement in rate-distortion.

II. Background and Related Work

Lossy Data Reduction. Data sets in HPC contain large amounts of floating-point data. Due to the random nature of mantissa bits, lossless compression methods do not give significant reduction. Instead of saving data at each time-step, decimation stores data from a subset of time steps, often in full resolution. Truncation lowers the precision level of the data — e.g., 64-bit to 32-bit floating-point. To control accuracy in the reduced data, error-bounded lossy compression (EBLC) algorithms such as SZ [3] and ZFP [4] provide order-of-magnitude larger reductions than lossless compression while meeting user specified levels of data fidelity. However, setting error bounds to ensure fidelity is an open question [5], [9].

SZ. SZ [3], [7] is an EBLC that compresses a data set by first decomposing the data into fixed sized blocks and then applying a multistep process: (1) Data Prediction – data points in each block are predicted based on previously predicted values using either a Lorenzo predictor or a local linear regression predictor; (2) Linear-scale Quantization – error in the predicted value for each point is converted from a floating-point to an integer by applying an equal-bin-size quantization between the range of $[-\epsilon, \epsilon]$, where ϵ is the compressor's error bound; and (3) Encoding – sequence of integer codes are further compressed using entropy encoding techniques such as Huffman coding and dictionary based methods such as GZip or Zstd. SZ bounds the error in multiple ways and supports multiple I/O libraries. In addition, SZ takes advantage of on-node parallelization such as OpenMP, GPUs [8]. We select SZ due to its superior performance [10] among competing HPC EBLC algorithms such as ZFP, and truncation.

III. Performance Optimization

The current CPU SZ is designed to give large compression ratios at reasonable compression bandwidths. To improve the compression bandwidth, SZ employs thread-level parallelization via OpenMP where each thread works independently on a number of blocks. The current CPU SZ has a read after write (RAW) dependence precluding optimization via SIMD parallelism, limiting compression bandwidth. We enable fine-grained SIMD parallelism by exploiting the dual-quant (DQ) algorithm of cuSZ [8] for data prediction and error quantization[1]. We contribute improvements to DQ through autotuning and dynamically selecting border padding values.

[1]We focus on this step as opposed to the Huffman encoding because there exist vectorized implementations of Huffman encoding [11].

1) Dual-Quantization: SZ chunks the original dataset D into fixed sized blocks. Each block is compressed independently. Algorithm 1 shows an overview of compression and decompression. We denote variables we generate and use during compression with an open circle superscript, and data in decompression with a closed circle superscript.

Compressing each data point $d \in D$ begins by predicting the data value via Lorenzo prediction ℓ. Lorenzo prediction predicts the value of d based on the values of previously predicted surrounding data, d_{SR}, in the block [7]. After prediction, we compute the error e° in the original data d and our prediction p°. Next, we quantize the error based on the user-selected error bound eb. Quantizing the error allows us to represent the error in the prediction as an integer, which compresses more efficiently than floating-point. Data whose prediction error is larger than eb is stored in the `else` block verbatim with no loss in accuracy. Decompression uses the Lorenzo prediction to reconstruct each d, reversing the quantization process.

Algorithm 1: SZ-1.4 compression and decompression.

```
1  Function compress()
2      for d ∈ D do
3          p° ← ℓ(d_SR)
4          e° ← p° − d
5
6          if e°/eb < CAP (IN-CAP) then                          // quantization
7              e°_D ← INTEGERIZE(e°/(2 × eb))
8              RECONSTRUCTED ← p° + 2 · e°_D · eb
9              WATCHDOG(RECONSTRUCTED − d < eb, fallback: OUTLIER)
10         else
11             OUTLIER: e°_D ← 0 and record the verbatim x ← d
12         end
13
14         d ← RECONSTRUCTED or x accordingly                    // incurs raw
15     end
16 Function decompress()
17     for d• ∈ D• to reconstruct cascadingly do
18         p• ← ℓ(d•_sr)
19         d• ← p• + 2 · e°_D · eb if IN-CAP else verbatim x
20     end
```

Algorithm 1 has a loop carried RAW dependence on line 14. Because the Lorenzo predictor needs predicted data values from prior iterations, the SZ-1.4 algorithm is not able to be vectorized. To remove the dependence, the *DQ* algorithm (Algorithm 2) separates prediction and quantization, allowing for fine-grained parallelization [8]. Lines 2 and 3 represent the pre-quantization stage, where each datum $d \in D$ is quantized based on the error bound, forming a new dataset D° of quantized data d°. D° has an error that is less than the user's error bound $|d - 2 \cdot d^\circ \cdot eb| < eb$. After the pre-quantization step, we use Lorenzo prediction to predict the values of d°.

After pre-quantization, the DQ algorithm starts the post-quantization step, where we compute the difference, δ°, between the predicted value and the prequantized value, d°. We quantize d° similarly to linear-quantization in SZ-1.4 to obtain an integer quantization code. Because we have pre-computed all the prediction values, we are able to quantize all the predictions in parallel, removing the RAW dependence for DQ compression. For decompression, we are unable to vectorize due to the dependency that each data point cannot be decompressed until the values preceding it are reconstructed.

2) Performance Modeling: When optimizing, programmers seek the highest level of performance. However, the hardware's maximal computation rate is often an unrealistic goal. The true

Algorithm 2: vecSZ compression and decompression.

```
1  Function compress()
2      for ∀d ∈ D concurrently do
3          d° ← round(d/(2×eb))                                   // pre-quant
4          d ← d°                                                 // barrier
5          p° ← ℓ(d°_sr)
6          δ° ← p° − d°
7
8          if δ° < CAP/2 (IN-CAP) then                            // post-quant
9              δ°_D ← CAST<FLOAT2INT>(δ°)
10         else
11             OUTLIER: δ°_D ← 0 and record the verbatim x ← d°
12         end
13     end
14 Function decompress()
15     for d• ∈ D• to reconstruct cascadingly do
16         p• ← ℓ(d•_sr)
17         d• ← (p• + δ°_D) · (2 × eb) if IN-CAP else verbatim x
18     end
```

performance of an algorithm is highly dependent on whether the algorithm is computation or memory bound. To establish the maximal performance level, we construct a Roofline performance modelThe Roofline model is a way to visualize the peak floating-point and memory performance based on the operational intensity (operations per byte of DRAM traffic) of a given algorithm [12]. To generate the ceilings for the model, we find the sustainable memory bandwidth and the peak floating-point performance of each CPU using the Lawrence Berkeley National Lab's Empirical Roofline Tool (ERT) [13]. The ERT determines the machine characteristics by running micro-kernels on the target machine.

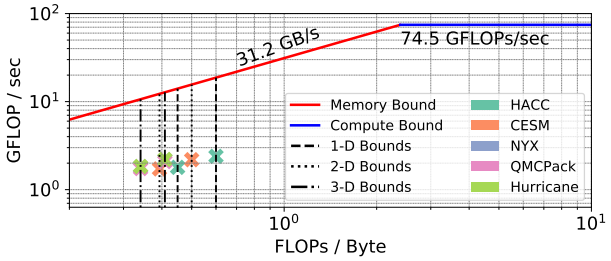


Fig. 1: Roofline model of the operational intensities for DQ.

Analyzing Algorithm 2, we compute operational intensity (OI) as FLOPS/byte accessed by DRAM and FLOP/sec and establish conservative and lenient bounds. The conservative bound is calculated by including strictly arithmetic operations when calculating the FLOPS/byte, while the lenient bound includes operations such as floating-point type casts and comparisons. Choosing conservative and lenient bounds as opposed to a single OI ensures our actual performance lies between the two, easing algorithm analysis. Deriving conservative and lenient bounds is applicable to other codes. We find this technique useful for analyzing complex codes where counting operations is difficult or when unsure how compiler optimizations transform the code. Figure 1 shows the OI bounds for 1D, 2D, and 3D version of the DQ algorithm without any optimizations on our test applications (Table II). We find that for both the lenient and conservative estimates, the OI for DQ is memory-bound, corresponding to values under the slanted region of the model. Before we apply our contributions, DQ does not fully utilize the CPU's resources, reaching between 10–25% of the theoretical peak performance.

3) Vectorization: Using compiler based autovectorization presents a non-labor intensive way to apply vectorization without the need for further modification to the actual code. However, autovectorization relies on compile time analysis that is limited in applicability [14]. One example of this is the manner in which we ensure that computation is performed only within the dimensions of our data. Any out-of-bounds computation as a result of block partitioning data is discarded. In a sequential program, it is sufficient to continue to the next row or block if the remaining values in a block contain out-of-bounds elements. The control logic required for this operation prevents autovectorization. With vectorization, as long as the vector register contains at least one in-bounds element, there is no additional cost to compute the out-of-bounds elements. Thus, we modify the boundary check to perform fewer checks, at a vector register granularity as opposed to a data element granularity, by generalizing `if` statements that check boundaries and moving them outside `for` loops. This technique is portable to other codes where control flow prohibit autovectorization.

For the sections of code not autovectorized, we manually vectorize via GCC compiler intrinsics. Using intrinsics, we port this code to other CPUs that use AVX vectorization without modification to the code. Using the GCC intrinsic vector functions, we vectorize the pre/post-quantization loops manually to perform work on up to 16 floating-point values (AVX-512) and 8 floating-point values (AVX2).

Manually vectorizing DQ while leveraging different vector register lengths introduces several challenges that we must consider. The AVX-512 intrinsics contain a wider range of instructions. Determining how to map these instructions to the operations available on CPUs with lower vector capabilities is important for ensuring no performance degradation. In particular, we look at the latency and throughput of the instructions, selecting the most comparable one available for each set of intrinsics [15]. vecSZ has code paths for each vector length and block size and selects the best choice at runtime via autotuning (see § III-5).

4) Block Size: SZ logically decomposes the data set into small, fixed sized blocks to compress independently. The dimension of the blocks is not configurable in the original SZ. However, when mapping vector operations to the computation, certain block sizes lead to inefficiencies due to register underutilization. For example, a block size of 6×6 (2D data) and a vector that holds 8 values, 25% of the vector is not utilized for each operation. To reduce this inefficiency, we use block sizes that are multiples of the vector register in use. SZ's block size of 256 for 1D, 16×16 for 2D, and $6 \times 6 \times 6$ for 3D leaves room for additional computation in vector registers of 256-bits and 512-bits in length. The optimal block size for 1D, 2D, and 3D data varies based on the data and the vector register size in use. We use block sizes of 8, 16, 32, and 64 as 128 and 256 did not yield additional improvements.

5) Autotuning: The performance between different configurations of block size and vector length can vary prediction and quantization bandwidth by up to 300% (see § V-5). To

determine the optimal configuration of block size and vector length for a data set, we develop a heuristic for tuning parameters by performing computation on a sample of random blocks. Before running full DQ, we perform an exhaustive search of all configurations, sampling a fixed percentage of blocks from the dataset at random in order to estimate the optimal configuration of block size and vector length. We repeat this multiple times, choosing the best performing configuration for compression. We amortize the overhead of autotuning when running multiple time-steps of a simulation because the best configuration holds across the majority of time-steps.

6) OpenMP: We introduce thread-level parallelism at a block granularity via OpenMP to further accelerate the fine-grained parallelism we apply through vectorization. Each block is calculated independently of all other blocks. We optimize scheduling of threads on cores using OpenMP thread affinity controls. Using `OMP_PLACES=cores` and `OMP_PROC_BIND=close` schedules threads to cores on a single socket before scheduling threads on the next socket [16]. This configuration ensures we keep threads as close to the data on which they are operating for as long as possible. While SZ currently only supports OpenMP for 3D data, we implement OpenMP capabilities in vecSZ for 1D, 2D, and 3D data.

IV. Prediction Optimization Via Dynamic Padding

During the prediction and quantization step, values without preceding elements (i.e. those found along borders) rely on block padding for prediction. The original DQ method uses zeros to pad the blocks, regardless of the data set. Zero padding yields inconsistent results across different data sets. On near zero data sets, it results in better border prediction than a dataset with relatively few near zero values. In extreme cases, 100% of the unpredictable data points are border elements. Increasing the number of unpredictable outliers, decreases the compression ratio because unpredictable data needs more storage per element than predictable data [3].

vecSZ dynamically selects a padding value based on statistical properties of the data to more closely represent the data along block borders, reducing the number of outliers by up to 91%. Figure 2 shows for CESM's CLDHGH data set, 62% of border elements are unpredictable with zero padding. Our dynamic padding yields only 6%. Our method of dynamic padding is applicable to other compressors that leverage predictor that bootstrap border elements.

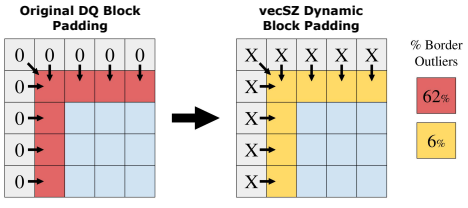


Fig. 2: Prediction of border values using zero-padding versus vecSZ's dynamic padding for CESM CLDHGH.

When computing padding values, we investigate the effect of choosing a minimum, maximum, or average value of our data at different granularities (global, block, and edge).

Global selects a single constant padding value for all the blocks based on the full data set and has the lowest storage overhead. The sacrifice we make to achieve this low overhead is in the optimality of the padding scalar chosen. Computing a single, global scalar falls victim to the limitations of zero padding. Even if computed via a global average, the granularity is often too coarse for most data sets.

Block computes a per padding value per block, which increases the storage requirements proportional to the number of blocks in the data set. This results in a higher storage overhead, but provides finer control over the padding value chosen per block. This granularity is best suited for data sets that vary significantly across their domain.

Edge computes a padding value for each block dimension, resulting in $nBlocks * nDim$ additional values to store. This fine-grained selection typically yields the best prediction. However, our experiments show the storage overhead often outweighs its benefit by decreasing the final compression ratio even when compressing the padding values.

Dynamic selection can include additional, more robust, methods for selecting padding values; however, we find that our chosen methods yield good results and highlight the necessary trade-offs in prediction accuracy and storage overhead.

V. Experimental Results

1) Hardware: We conduct our experiments using CloudLab, a facility for building clouds that provides access to computing, storage, and networking resources [17]. We concentrate our attention on two popular CPU models commonly found in HPC systems: the AMD EPYC Rome and the Intel Xeon Gold, described in detail in Table I. The Intel nodes we use contain two Xeon CPUs, and the AMD nodes contain one AMD EPYC CPU.

	CPU	
	AMD EPYC Rome 7452	*Intel Xeon Gold 6142*
RAM	127 GB	384 GB
Cache	128 MB	22 MB
Cores	32	16
Vectorization	AVX2	AVX-512

TABLE I: Detailed CPU specifications used for experiments.

A notable difference between the two CPU architectures is the level of vectorization supported. The Intel CPU has AVX-512 support, meaning support for 512-bit vector registers capable of operating on 16 32-bit floating-point values at a time. The AMD CPU only supports operations on up to 256-bit vector registers or 8 32-bit floating point values at a time. Moreover, the cache size of the Intel CPU is smaller than that of the AMD CPU. Although the Intel CPU can compute on twice as many values simultaneously, its smaller cache size means there is a higher probability of cache misses.

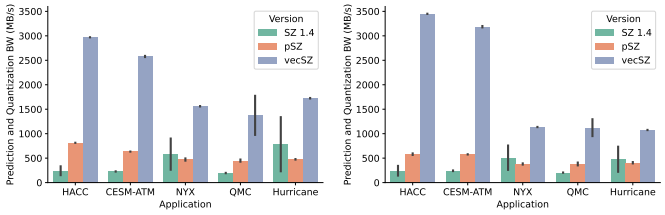
2) Test Datasets: We test vecSZ on real world HPC datasets (see Table II) from the SDRBench [18] that are representative of a wide range of HPC workloads. For the CESM, we use an absolute error bound of $1e{-}5$. For the others, we use an absolute error bound of $1e{-}4$.

Dataset	Domain	Type	Dimensions	Size (MB)
HACC (6)	Cosmology	fp32	280,953,867	1071.75
CESM (3)	Climate	fp32	1,800×3,600	24.72
Hurricane (20)	Climate	fp32	100×500×500	95.37
Nyx (6)	Cosmology	fp32	512×512×512	512.00
QMCPACK (2)	Quantum	fp32	288×115×69×69	601.52

TABLE II: Attributes of the datasets used in experiments. Number in parentheses indicates number of fields used.

3) Experimental Methodology: We use the C++ `ctime` library's high resolution clock for timing all runtimes. Each experiment is run on both the AMD and Intel CPUs and results are averaged across ten total runs of each dataset. We plot the standard deviation as error bars when appropriate. As a baseline for evaluating vecSZ we use pSZ, a serial version of SZ that uses the DQ method as opposed to the prediction and quantization method used by SZ-1.4.13.5 [7]. We compare the performance of vecSZ to SZ-1.4 as opposed to SZ-2.1 because the Lorenzo prediction and quantization method used in SZ-1.4 is directly comparable to the DQ in vecSZ. SZ-2.1 alternates between Lorenzo prediction and liner regression, which does not provide a fair performance comparison. For each version of SZ, we use the same config file that comes with SZ-1.4.13.5. GCC 9.3.1's `-O3` option compiles and optimizes all of our codes with OpenMP v4.5 and to vectorize, we use `-march=native` and the vector flags that correspond to AVX, AVX2, and AVX-512 if they are available (`-mavx`, `-mavx2 -mavx512`, respectively).

4) Comparison to SZ: Applying vectorization to the DQ method enables it to process data faster, leading to improvements in bandwidth. Figure 3 shows the prediction and quantization bandwidth of SZ-1.4 compared with the pSZ baseline, as well as the best performing configuration of vecSZ (see Section V-5). We break down the performance on the AMD and Intel CPUs in Figure 3a and Figure 3b, respectively.



(a) AMD (b) Intel

Fig. 3: Prediction and quantization bandwidth of SZ-1.4, pSZ, and vecSZ. Black bars are standard deviation.

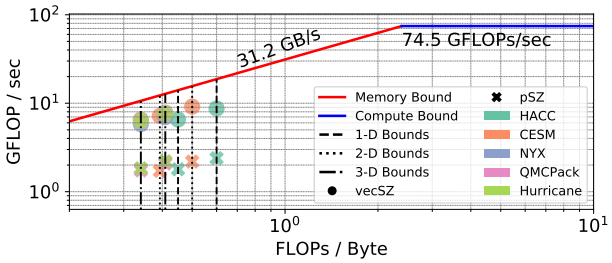
For the AMD CPU, the pSZ baseline shows better performance, on average, than SZ-1.4 in three of the five applications, with a maximal speedup of 3.4× for HACC. The AMD CPU shows its best performance for HACC and CESM, improving over the pSZ baseline by at least 2000 MB/s, resulting in a speedup of 3.7× and 4.1× respectively. Compared to SZ-1.4, vecSZ improves prediction and quantization bandwidth by 2700 MB/s for 1D and 2200 MB/s for 2D data, for a speedup of 13.1× and 12× for 1D and 2D data respectively. For 3D datasets, we outperform SZ-1.4 by 2.2–7.3× and pSZ by 3.2–3.7×, depending on the dataset. This corresponds to bandwidth increases of 1000–1100 MB/s for SZ-1.4 and 900–1200 MB/s for pSZ. 3D datasets result in increased prediction and quantization bandwidth on the AMD CPU as opposed to the Intel CPU due to its larger cache, resulting in an average of 30% fewer cache misses for higher dimensional datasets.

For the Intel CPU, pSZ has better performance than SZ-1.4 in the same three applications. Furthermore, for the 1D and 2D test datasets, vecSZ increases prediction and quantization bandwidth by greater than 2800 MB/s for SZ-1.4 and 2600 MB/s for pSZ, resulting in a 15.1× speedup over SZ-1.4 and 6× speedup over pSZ for 1D datasets, and 14.0× and 5.7× for SZ-1.4 and pSZ, respectively, on 2D datasets. 3D datasets exhibit less performance improvement compared to 1D and 2D, resulting in an increase of 600–900 MB/s or a 2.3–5.3× speedup. The decrease in speedup when performing DQ for 3D data is due to an increase in cache misses, up to 6×. We attempted to lower the cache miss rate by prefetching the blocks, but it did not yield substantial performance improvements.

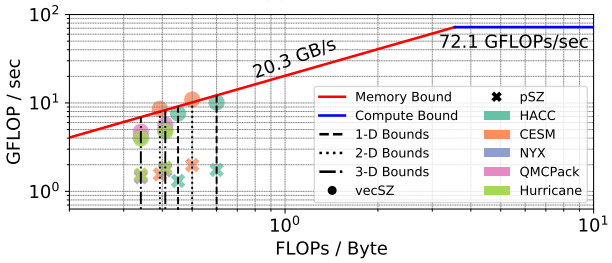
Both the AMD and Intel CPUs display similar trends in their bandwidth across datasets of different dimensions. Comparing the CPUs for 2D datasets, the larger vector registers of the Intel enable it to outperform the AMD. However, the larger cache of the AMD leads to higher bandwidths for 3D datasets than the Intel. Thus, for performance portable codes that operate on multidimensional arrays, data locality and caching are central. Overall, we find that vecSZ improves upon the bandwidth of SZ-1.4 by 8.7× for AMD and 9.2× for Intel on average, with a peak prediction and quantization bandwidth in excess of 2.9 GB/s for both AMD and Intel CPUs.

Roofline Analysis. To quantify how much of peak performance vecSZ obtains, we compare our results to the expected performance from the Roofline model. Using the peak GFLOPs attained by the optimal configuration of block size and vector length, we plot the DQ performance of pSZ and vecSZ in Figure 4.

The AMD CPU's performance results in peak percentage of DRAM memory bandwidth between 47–61%, improving over the baseline code by 3.2–4.2×. The Intel CPU shows between 57-107% of peak DRAM memory bandwidth. HACC and CESM result in the closest to peak performance. CESM shows a GFLOP/sec value over the peak DRAM memory bandwidth because it is able to fit within the 22 MB L3 cache, whereas larger datasets cannot. We do not see this behavior for CESM on the AMD CPU because it is able to sustain a higher peak memory bandwidth and does not have support for 512-bit vector registers that provide the best performance on the Intel CPU. The combined effect of these two features results in the performance difference between Figures 4a and 4b. For all



(a) AMD

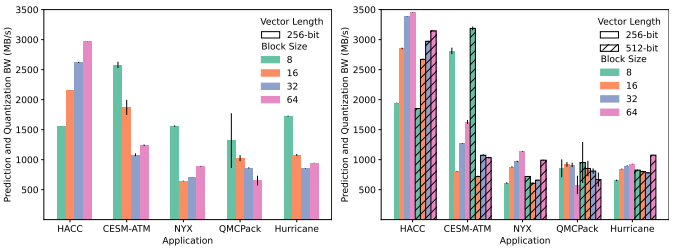


(b) Intel

Fig. 4: Roofline Model showing vecSZ performance for the DQ algorithm and the DQ no vectorization baseline, pSZ.

applications and both CPUs, we consistently improve over the baseline, pSZ, by 2.5–5.6× which highlights the performance portability of vecSZ's optimizations.

5) Understanding Vectorized Performance: The performance of the DQ algorithm is largely dependent on the block size used for chunking input datasets, and the vector register length used in computation. To determine the optimal configuration of block size and vector length for each dataset, we perform an exhaustive run of all possible block size and vector length configurations. We show the average prediction and quantization bandwidth per application for each of the possible configurations in Figure 5.



(a) AMD (b) Intel

Fig. 5: Impact of vector length and compression block size.

6) Block Size and Vector Length: Block size has the largest impact on performance of DQ. In some cases, adjusting the block size improves the prediction and quantization bandwidth by 354% for Intel and 139% for AMD. The differences in block size performance also vary across applications. HACC's performance improves as block size increases, whereas the general trend of QMCPack tends to decrease as block size increases. For AMD, we find that a block size of 64 performs best for 1D datasets, while a block size of 8 leads to optimal performance for 2D and 3D datasets. The Intel performance (Figure 5b) is more ambiguous, varying depending on vector length and the dataset. To account for this variability, we explore autotuning in Section V-7.

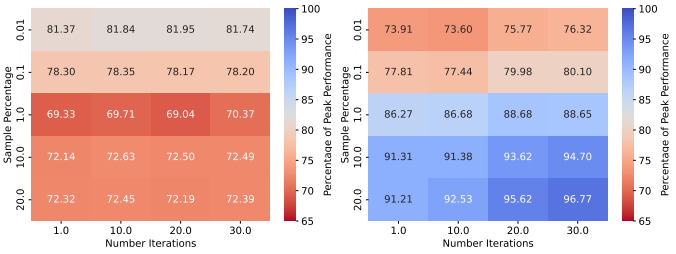
Since the AMD CPU supports up to 256-bit vector registers, choice of vector length only exists on the Intel CPU. In most cases, 256-bit vector registers perform slightly better for the Intel CPU. However, for CESM, QMCPack, and Hurricane, 512-bit vector registers is best. A potential reason for the improved performance of 256-bit over 512-bit registers is that the size of a cacheline in the Intel CPU is 64 bytes. This means that one 512-bit vector contains an entire cache line worth of data. If data is misaligned, it results in degraded performance.

Block size should be a multiple of the vector register in use. For example, a block size of 8 will not fill a 512-bit vector register. In this case, autovectorization performed by the compiler uses 512-bit vector registers, while our manual vectorization reverts to a vector size of 256-bits in order to utilize the entire vector register. This creates a hybrid of 512-bit and 256-bit vector operations that attribute to the performance improvements between 512-bit and 256-bit vector lengths for block size 8. This technique is applicable to other vectorizable codes, where data does not fit perfectly in vector registers, to increase the efficiency of each vector instruction.

7) Autotuning Block Size and Vector Length: When tuning, the Intel CPU has 8 configurations of block size and vector length, while the AMD CPU has 4. vecSZ's autotuning determines the best configuration based on the best average performance of the DQ operation for a subset of all blocks in a dataset completed across multiple sampling iterations.

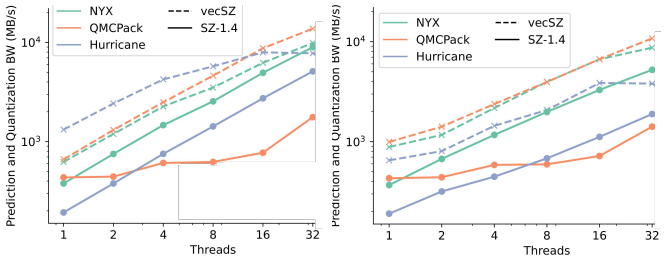
Figure 6 states the percentage of the peak performance of a configuration achieved by each pair of autotuning settings, averaged across all applications. Results for individual applications yield similar performance. In Figure 6, the more frequently an autotuning run selects the best configuration, the closer that configuration's portion of the heatmap is to the peak performance. Results for AMD yield a higher percentage of peak performance for smaller sample percentages of blocks. Since across iterations the same blocks may be used, the larger cache of the AMD CPU has many blocks resident in cache, although a random sample of data is being taken. Since the cache of the Intel Gold CPU is much smaller, this issue is not present. This allows a more accurate configuration to be found as more samples are taken and averaged over a larger number of trials. We conclude that caching affects the accuracy of autotuning with small datasets. However, as dataset size increases, the cost of the autotuning increases, as we discuss below. Clearing or polluting the cache between iterations is an option, but incurs additional time overheads.

For both CPUs, the percentage of sampled blocks and number of iterations measured for autotuning affect the percentage of time spent during the parameter sweep. Increasing the blocks sampled and iterations measured increases time spent autotuning. The performance cost of increasing the number of iterations is lower than the cost of increasing number of blocks sampled, however we find that increasing the percentage of blocks sampled results in performance closer to that of the



(a) AMD (b) Intel

Fig. 6: Autotuned percentage of peak performance.



(a) AMD (b) Intel

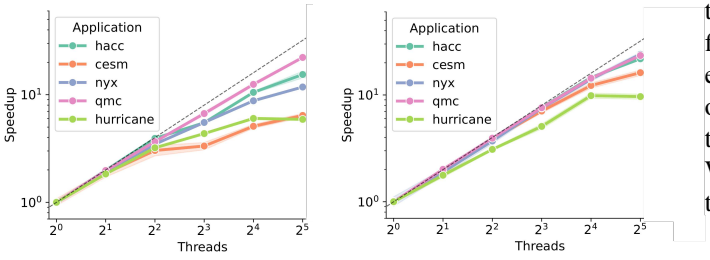
Fig. 8: OpenMP performance for vecSZ and SZ-1.4

best configuration. The AMD processor completes the sweep more quickly because it has a smaller set of vector capabilities available and, as a result, fewer parameters to sweep.

When choosing a reasonable percentage of blocks to sample and iterations to average before autotuning, it is also important to consider the time required to autotune for a given configuration. Depending on the application, a trade-off to consider is a balance between gaining closer to peak performance and spending less time tuning the configuration.

The Intel CPU exhibits a trend corresponding with the result that if a larger percentage of blocks from a dataset are sampled, the tuning configuration chosen by our autotuning algorithm is closer to optimal. Additionally, if we repeat our experiment for a larger number of iterations, we achieve a value within 5–10% of peak performance. The high-cost of autotuning for larger percentages and larger number of iterations can be amortized if we reuse the configuration for the next several time-steps. Prior work shows, for fields in a scientific application, their data properties are similar through time and an optimal compressor configuration remains optimal or nearly optimal for several adjacent time-steps [19]. We find that for each field of the Hurricane dataset, when examined across all 48 time-steps, results in one of two configurations for an average of 80% of autotuning runs. Using this knowledge, autotuning for each time-step based on the top two configurations drastically reduces the overhead by up to 6× for Intel. Moreover, reusing a single tuning for multiple time-steps further lowers the cost.

8) Thread-level Parallelization: We perform thread-level parallelization at the block level for vecSZ's DQ operation using OpenMP. Figure 7 scales the tread count from 1–32 and presents the speedup of vecSZ for each application over the single-threaded performance of vecSZ. Overall, we see a maximal speedup of 24× at 32 threads.



(a) AMD (b) Intel

Fig. 7: Single node OpenMP scaling performance.

For AMD, DQ operation scales nearly linearly up to 4 threads. At 8 threads, CESM and Hurricane begin to reach their peak speedup, while HACC, Nyx, and QMCPack continue scaling linearly. For all applications, except Hurricane, the Intel CPU scales nearly linearly until reaching 16 threads. Hurricane begins to not scale well at 4 threads, plateauing at 16. On the Intel node, moving from 16 to 32 OpenMP threads utilizes the second CPU.

We compare vecSZ's prediction and quantization bandwidth to that of SZ-1.4 in Figure 8. We only show the performance for the 3D datasets, as SZ-1.4 does not support OpenMP compression of 1D or 2D datasets. When performing the prediction and quantization operation on up to 32 threads, vecSZ outperforms SZ-1.4 by as much as 11.6× for 16 threads for QMCPack on the AMD CPU.

9) Overall Impact: The above results focus on optimizations performed on the prediction and quantization operations of the compression pipeline. We now address the impact of our optimizations on the overall performance of vecSZ.

	CPU	
	AMD EPYC Rome	*Intel Xeon Gold*
Dual-Quant % of Runtime	46.9%	42.9%
Theoretical Max Speedup	1.70×	1.67×
Actual Speedup	1.51×	1.47×
% of Theoretical Achieved	88.9%	87.6%

TABLE III: Theoretical and actual speedup for vecSZ over it's serial implementation, pSZ, averaged for all applications.

The DQ operation takes an average of 46.9% and 42.9% of the total sequential runtime for the AMD and Intel CPUs, respectively. Using Amdahl's Law, theoretical maximum speedup is computed by: $S = \frac{1}{(1-p)+p/s}$, where p is the proportion of the sequential runtime that an operation takes and s is the speedup of the operation being parallelized. For the AMD CPU, we set $s = 8$ because a 256-bit vector register fits 8 32-bit floating point values. For Intel, $s = 16$. Using this equation, we find the theoretical speedup shown in Table III of the total runtime possible by performing vectorization of the DQ operation to be 1.7× for AMD and 1.67× for Intel. We achieve 89% of the theoretical maximum by reaching a total speedup of 1.51× for AMD and 88% for Intel at 1.47×.

10) Impact of Dynamic Padding: Across all block sizes and error-bounds, the use of dynamic padding values results in a decrease in unpredictable values by an average of 10%. The best cases results in 100% elimination of all outlier

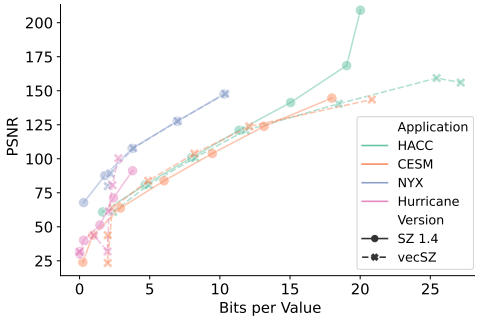


Fig. 9: Rate distortion comparison of vecSZ and SZ-1.4.

data. For larger error-bounds, the prediction of border values becomes more important because fewer unpredictable values exist in the rest of the block, therefore a larger percentage of the unpredictable values come from border regions that remain unpredictable regardless of error-bound due to the use of zero padding. In these cases, application of a global or block average as dynamic padding results in elimination of all outliers because we make it possible to predict the border values. We also find that use of minimum and maximum values for padding do not perform as well as average value because they tend to be outliers in the dataset.

Using a global average value for padding to provide padding that is representative of the data without incurring excess overhead of storing additional padding values, we generate Figure 9, which shows the rate-distortion of vecSZ and SZ-1.4. For reasonable error-bounds, we perform equally, or better than SZ-1.4. For CESM and Hurricane data, we improve rate-distortion up to 18.9% and 32% respectively. Thus, dynamic padding could be useful for other prediction based compressors to improve rate-distortion.

VI. Conclusions

As HPC grapples with larger volumes of data, compression techniques are needed to effectively reduce the data. In this paper, we present and optimize vecSZ, a threaded and vectorized version of the dual-quantization algorithm for CPU architectures. We find that best performance depends on dataset, compression block size, and vector register length. vecSZ's autotuner selects the best configuration and improves the prediction and quantization bandwidth by up to 15.1× compared to SZ-1.4, improving peak DRAM bandwidth by as much as 61–107%. Our novel non-zero block padding reduces the number of outlier data on the block's border by up to 100% yielding up to a 32% improvement in rate distortion.

Acknowledgment

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations – the Office of Science and the National Nuclear Security Administration, responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, to support the nation's exascale computing imperative. The material was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357. This material is based upon work supported by the National Science Foundation under Grant No. SHF-1910197, SHF-1943114, OAC-2003709, OAC-2042084, OAC-2104023, and OAC-2104024.

References

[1] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, V. Vishwanath, T. Peterka, J. Insley *et al.*, "HACC: Extreme scaling and performance across diverse architectures," *Communications of the ACM*, vol. 60, no. 1, pp. 97–104, 2016.

[2] S. W. Son, Z. Chen, W. Hendrix, A. Agrawal, W. keng Liao, and A. Choudhary, "Data compression for the exascale computing era - survey," *Supercomputing frontiers and innovations*, vol. 1, no. 2, 2014. [Online]. Available: http://superfri.org/superfri/article/view/13

[3] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, 2016, pp. 730–739.

[4] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec 2014.

[5] J. Calhoun, F. Cappello, L. N. Olson, M. Snir, and W. D. Gropp, "Exploring the feasibility of lossy compression for pde simulations," *The International Journal of High Performance Computing Applications*, vol. 33, no. 2, pp. 397–410, 2019.

[6] "Reducing disk storage of full-3d seismic waveform tomography (f3dt) through lossy online compression," *Computers & Geosciences*, vol. 93, pp. 45–54, 2016.

[7] 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 2017, Orlando, FL, USA, May 29 - June 2, 2017*, 2017, pp. 1129–1139.

[8] J. Tian, S. Di, K. Zhao, C. Rivera, M. H. Fulp, R. Underwood, S. Jin, X. Liang, J. Calhoun, D. Tao, and F. Cappello, "Cusz: An efficient gpu-based error-bounded lossy compression framework for scientific data," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 3–15.

[9] A. H. Baker, H. Xu, D. M. Hammerling, S. Li, and J. P. Clyne, "Toward a multi-method approach: Lossy data compression for climate simulation data," in *High Performance Computing*, ser. Lecture Notes in Computer Science, J. M. Kunkel, R. Yokota, M. Taufer, and J. Shalf, Eds. Springer International Publishing, 2017, vol. 10524, pp. 30–42.

[10] 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 *IEEE International Conference on Big Data, Big Data 2018, Seattle, WA, USA, December 10-13, 2018*, 2018, pp. 438–447.

[11] T. Drijvers, C. Pinto, H. Corporaal, B. Mesman, and G.-J. Van den Braak, "Fast huffman decoding by exploiting data level parallelism," 08 2010, pp. 86 – 92.

[12] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009.

[13] Lawrence Berkeley National Laboratory, https://crd.lbl.gov/departments/computer-science/par/research/roofline/software/ert/, 2020, online.

[14] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," *Proceedings of the CGO 2006 - The 4th International Symposium on Code Generation and Optimization*, pp. 281–294, 2006.

[15] Intel, "Intel intrinsics guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide, online.

[16] Victor Eijkhout, "Openmp topic: Affinity," https://pages.tacc.utexas.edu/eijkhout/pcse/html/omp-affinity.html, 2011, online.

[17] A. Akella, "Experimenting with next-generation cloud architectures using cloudlab," *IEEE Internet Computing*, vol. 19, no. 5, pp. 77–81, 2015.

[18] "Scientific Data Reduction Benchmark," https://sdrbench.github.io/.

[19] R. Underwood, S. Di, J. C. Calhoun, and F. Cappello, "Fraz: A generic high-fidelity fixed-ratio lossy compression framework for scientific floating-point data," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 567–577.