Verification of Gravel: a CRUSHED brick on Stencils

Sameeran Joshi
University of Utah
Salt Lake City, Utah, USA

ABSTRACT

This report presents an overview as a part of the work done for the class project for CS6110, a software verification course at the University of Utah. The project tries to understand different tradeoffs for performance, accuracy, and precision for a highly performance-portable fine-grained data blocking library called Bricklib using compression and decompression techniques and further verifying it using Significand Decimal Digits from numerical methods. This work shows that compression, in fact, truncates double-precision data into single-precision. The work shows experimental results on different compression algorithms and uncompressed versions as a baseline on 3D7PT stencils. We show different challenges we overcame and end by mentioning future scopes, which might be worth looking at.

KEYWORDS

Stencils, HPC, Compression, Decompression, Numerical Analysis, Significant Decimal Digits, BrickLib, Crusher

1 INTRODUCTION

Today, modern scientific applications rely on verified components that should work as expected; for example, various scientific applications doing stencil computations rely on high-performance libraries. These libraries need to be sound and verified while doing all the floating point calculations as fast as possible on HPC machines. Scientists and engineers rely on such verified components to inform their daily activities and generate novel results.

Stencil computations are critical and widely used in applications like PDE solvers and image processing applications. The derivative of each point in the lattice is computed as a weighted sum of neighboring points. Depending on the structure of the lattice, different types of stencils, like 5-point 2D stencils, 7pt3D, and 27pt3D are possible. The data layout plays a vital role in these stencil computations to achieve the speedups by reducing the data movement. Higher-order stencils have more data reuse and are usually memory-bound. At the same time, achieving portability and performance on various hardware architectures is essential. BrickLib[5] has shown its effectiveness in achieving this performance portability across different architectures, types of stencils, and different applications.

Scientific applications need extremely large volumes of input data to run scientific simulations. Scientists are increasingly turning to high-order stencil schemes that perform more computations, leading to the use of higher-order stencil usage. However, high-order stencils need large volumes of data. Hence, It is essential to reduce the input data size so that the computations can be performed efficiently. Compression can be seen as one of the ways to solve the above problem. Lossless compressors suffer from limited compression ratios; hence, people have applied lossy compressors in the past with error-controlled techniques. LC[3], a successor of Crusher, is a framework for automatically generating high-speed lossless and guaranteed-error-bounded lossy data compression and decompression algorithms. It supports both CPUs and GPUs. Both tools can automatically synthesize compression pipelines from a library of data transformations. Applying lossy compression techniques on scientific input data has the benefit of efficiency by reducing the size of the input. However, it might lead to inaccuracies in floating point calculations. It is critical to verify the accuracy and precision of the scientific input data after applying lossy compression, as that might lead to the accumulation of errors in the output.

The goal of this project was a unique amalgamation where we aimed to verify the accuracy of a performance-portable fine-grained data-blocking library, Bricklib, after compressing and decompressing using CRUSHER/LC. We used techniques from numerical analysis, specifically Significand Decimal Digits(SDD), which measures the number of significant digits. We call this verification of Gravel as a Brick is crushed. This evaluation will help us understand the right amount of data compression ratio, performance and accuracy, and tradeoffs among them.

Following are the contributions of this work:

1. Using the Significant Decimal Digits technique, We verify the outputs from baseline bricks against Compression-Decompression (CDC) applied to input data.

2. We show this for 3D7Pt stencils for a brick size of 1 and domain size of 4.

3. We show results with lossless and lossy compression techniques and various compression strategies.

2 STATUS

Two major promises were made during the proposal submission

1. Integrate the tools together and get end to end working of the pipeline.

2. Apply the SDD verification technique and generate results.

As promised, both goals were accomplished with the application of 3D7PT stencils. Relative to ideal contributions, the project misses extending further to different types of higher-order stencils like 27PT and other star-shaped and box-shaped stencils. Currently, the end-to-end pipeline works where the data from stencil computations is extracted, compressed, and decompressed back using predefined algorithms using LC framework, reruns bricks with this CDC data, and compares the output generated from both using 2 different floating point verification techniques the first one using Floating point TOLERANCE(1e-6) check and the second using Significant Decimal Digits.

3 DEMO

Fig.1 shows the system diagram. There are three components in the design of the system: one that deals with bricklib, another that handles the crusher part, and third, a wrapper python script that combines the previous two, has control over the schedule, and verifies the results. Such a modular design keeps each component

separate and can be extended easily with any other compressor tool or application in the future. An Xpt-2D/3D(5pt2D, 9pt2D, 13pt2D, 17pt2D, 21pt2D, 25pt2D) stencil computation is an input to bricklib. The algorithm shown in the system diagram is as follows: BrickLib divides the input computation domain(stencil problem size being solved in our case) into subdomains and allocates a brick data structure for each subdomain. First, various dimensions are initialized: the size of a brick, padding required by each brick, and strides. The input data needed for stencil computations are coefficients and the grid values in stencils. Both values are synthetically generated using uniform random distribution. These values are not representative of real-world data. We extract the synthetic input data from a brick before applying any computations. The data is passed as input to the compressor LC framework, and the tool is configured with various compression-decompression parameters, compressing a brick depending on the granularity we choose. This generates gravel; note that we preserve the layout of bricks. We decompress gravel back to bricks. Given that the compressor here is lossy, we assume the final compressed-decompressed brick (CDC) is not the same as the original brick. Further, the Compressed-Decompressed(CDC) data is passed to bricklib for another set of computations, and the outputs generated from the CDC pipeline and original brick computations are passed as input to the verification algorithm. We run the end-to-end bricks pipeline experiments with the baseline(without compression-decompression) and CDC bricks.

Verification: The following section discusses different ways that have been explored in the past and applies those techniques in this work. In machine learning, the loss of accuracy is acceptable to some extent, whereas in scientific applications, this might lead to incorrect calculations; hence, the accuracy has to be very well preserved. To verify such systems, there are 2 main techniques we rely on, 1. Numerical verification techniques: Khalid et al.[2] showed a way to reduce double-precision floating-point representations into single-precision and mixed-precision. They used a method called significant Decimal Digits(SDD) from numerical analysis methods to ensure that their conversion accuracy is preserved. We find the most significant digits in the output of original and CDC bricks. They are expected to have the same significant digits in a certain acceptable range or equal for the output to be verified. The significant digits method can be at a much higher granularity level than the verifier method inside CRUSHER, which is at a more bit-by-bit granularity. The verification from CRUSHER uses ways to verify if the compressed and uncompressed data are the same. For example, it has various algorithms implemented inside the verifier part of the LC framework like LOSSLESS, MAXR2R, MAXREL, MAXABS, MSE, PSNR, etc... The current state of this verifier is to verify compressed and uncompressed inputs and check if they are the same. I plan to take this verification for the original brick and the CDC brick. The challenging part here might be given that the brick layout for a stencil is a collection of many individual bricks. I would need to preserve the ordering of these bricks and apply verification on an individual brick at a time and eventually have a combined result of verification. I also need to understand if these algorithms used internally can be modified such that they take input from a brick.

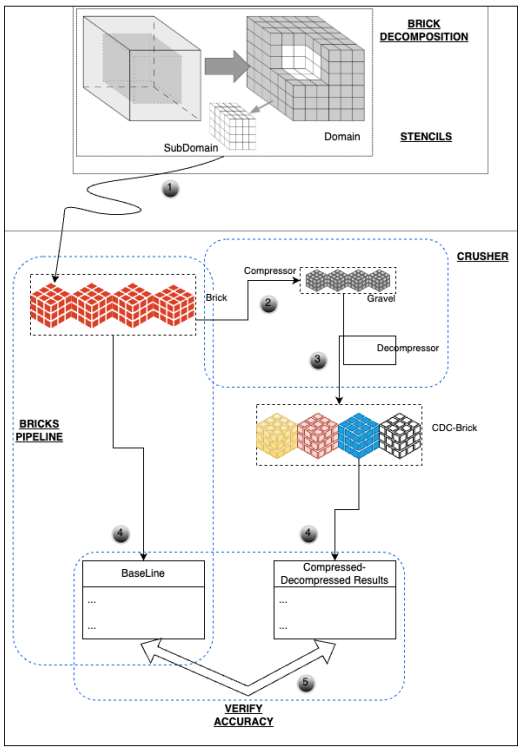

Figure 1: Example of a 7-point 3D BrickLib computation.

4 IMPLEMENTATION

BrickLib is a library written in C++. This library can be used out-of-tree and incorporated into external projects, providing flexibility for those projects to run their workflows without building the library every time. The 3D 7-point stencil code utilizes an external interface to interact with BrickLib. It sets various parameters such as brick dimensions (BDIM), domain size (N), strides, the ghost zone (GZ), and their padding sizes. The code supports multithreading using OpenMP. The function `init_grid` sets the layout of the Grid, a structure that points to individual bricks. The 7-point stencil kernel allocates a variety of brick layout-based data structures using the highly templatized `Brick<Dim<BDIM>, Dim<VFOLD>` API from BrickLib, which requires the dimensions to be specified for each brick. `init_grid` allocates space for grids and initializes them with random values. This data is then captured into files, which are passed as inputs to Crusher/LC. BrickLib is a hierarchical structure where a Grid is a 2D structure, and each grid element points to a 3D brick. This design provides flexibility for managing the data layout, as the elements are eventually laid flat as a 1D array in memory. For example, a 7-point 3D BrickLib computation is shown in Fig. 2. The code retrieves a brick from a grid via simple array indexing and then accesses the brick `bIn1` using an extra fourth dimension `bIn1[b][i][j][k]`, where `i`, `j`, `k` are the indices for accessing a brick. Each brick output element (`bElem`) is a weighted sum of coefficients (`coeff`) and `bIn1` (input brick elements).

Crusher/LC comprises of three integral parts—the preprocessor library, a component library featuring various compression

```
auto brick_func1 = [&grid1, &bIn1, &bOut1]() -> void {
  PARFOR
  for (long tk = GB; tk < STRIDEB - GB; ++tk)
    for (long tj = GB; tj < STRIDEB - GB; ++tj)
      for (long ti = GB; ti < STRIDEB - GB; ++ti) {
        unsigned b = grid1[tk][tj][ti];
        for (long k = 0; k < TILE; ++k)
          for (long j = 0; j < TILE; ++j)
            for (long i = 0; i < TILE; ++i) {
              bOut1[b][k][j][i] = coeff1[5] * bIn1[b][k + 1][j][i] + coeff1[6] * bIn1[b][k - 1][j][i] +
                                  coeff1[3] * bIn1[b][k][j + 1][i] + coeff1[4] * bIn1[b][k][j - 1][i] +
                                  coeff1[1] * bIn1[b][k][j][i + 1] + coeff1[2] * bIn1[b][k][j][i - 1] +
                                  coeff1[0] * bIn1[b][k][j][i];
            }
      }
};
```

Figure 2: Example of a 7-point 3D Stencil computation in BrickLib.

```
double kor = fabs(brick1 - brick2); // original - CDC
// if (kor<=0.0000000001) ki_l++;
if (kor<=0.0000000000000001) k15_l++;
else if (kor<=0.000000000000001) k14_l++;
else if (kor<=0.00000000000001) k13_l++;
else if (kor<=0.0000000000001) k12_l++;
else if (kor<=0.000000000001) k11_l++;
else if (kor<=0.00000000001) k10_l++;
else if (kor<=0.0000000001) k9_l++;
else if (kor<=0.000000001) k8_l++;
else if (kor<=0.00000001) k7_l++;
else if (kor<=0.0000001) k6_l++;
else if (kor<=0.000001) k5_l++;
else if (kor<=0.00001) k4_l++;
else if (kor<=0.0001) k3_l++;
else if (kor<=0.001) k2_l++;
else if (kor<=0.01) k1_l++;
else k0_l++;
```

Figure 3: Example of `verifyBrick_numerical` code for SDD verification.

algorithms, and the framework, which seamlessly chains preprocessors and components into pipelines to construct compression and decompression algorithms. The framework can automatically search for effective algorithms for a given input file or set of files by testing user-selected sets of components in each pipeline stage. The data captured in files from BrickLib is passed to a compressor-decompressor pipeline in CRUSHER/LC and again goes through the above brick computation. Furthermore, both `bIn` and `bOut` is passed to `verify_brick` and `verifyBrick_numerical`, performing floating-point TOLERANCE check and Significant Decimal Digits (SDD) verification.

The `verify_brick` functionality checks by iterating over each brick element if the difference is below the expected threshold of 1×10^{-6}; if not, the verification fails. `verifyBrick_numerical` is a bit more complicated and is shown in Fig. 3. The code checks if the difference is within a specific range denoted by the conditions in Fig. 3. These conditions attempt to calculate the number of significant decimal digits and generate a histogram from it. Bricks are verified based on the number of SDDs present. We explain the interpretation of these SDD values in the results sections.

5 RESULTS

In this section, we discuss various experiments that were performed. For the experimental setup, we used machines from CHPC[1]; GPUs were not used in this experiment as we chose the CPU version of LC. We build each submodule(Crusher and BrickLib) in the repository and use BrickLib as an external library by importing the appropriate

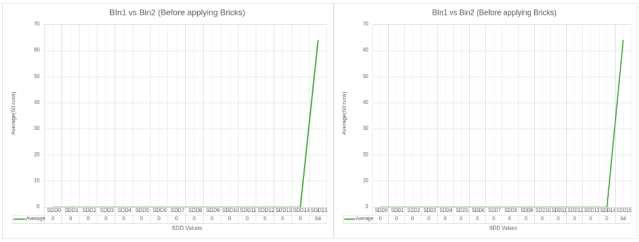

Figure 4: Lossless compression averaged over 50 runs with Component="TUPL4_1 RRE_1 TUPL2_1 RLE_4"

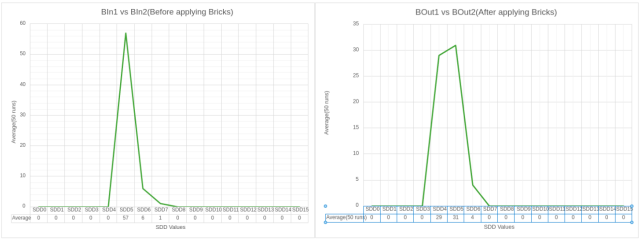

Figure 5: Lossy compression averaged over 50 runs with Preprocessor="QUANT_ABS_0_f64(0.000001)" and Component="RZE_4"

paths. The following modules were prerequisites for building the submodules gcc 11.2.0, openmpi, cuda, python 3.11.7. The Vector Scatter Optimization was turned off in BrickLib.In order to find a good compression algorithm, the user first needs to choose the best-suited preprocessor and component from crusher/LC and then pass it statically to this framework. Crusher/LC also provides a 'CR' mode, which searches for the best compressing algorithm. Preprocessors are needed to generate lossy algorithms with LC. They must be fully specified (no regular expressions are allowed) and cannot be searched for automatically as they require user-specified parameters such as the error bound. We performed 2 experiment runs, one with Lossless Compression and another with Lossy Compression, using 2 different compression techniques, and we averaged over 50 runs.

Fig. 4 shows the results for lossless compression without using Crusher applied on both input and output data after BrickLib computation; the histogram values except SDD15 are zeros. SDD values in the range of SDD14 and SDD15 signify that the input data is double precision Floating Point. Hence, the results conclude that there is no change in the precision of the input data, and the output remains the same after performing stencil computations.

Fig. 5 shows the results for Lossy compression used. The Input chart shows spike at SDD5 and some values for SDD6, this is a shift from SDD15 to SDD5/SDD6. SDD5/SDD6 values represent Single Precision Floating point numbers. This shift from SDD15 to SDD5/SDD6 indicates that there is a loss of precision from Double precision to Single Precision. We believe this comes from the Lossy Compressor as evidences from the input data shows a shift. The output data in bOut shows the accumulation of errors leading to

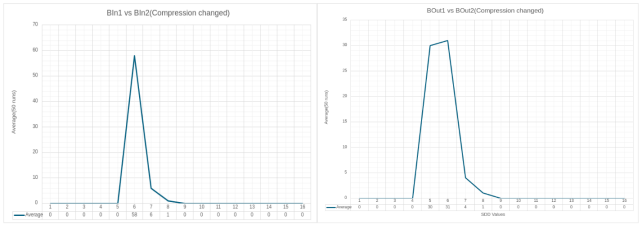

Figure 6: Lossy compression averaged over 50 runs with Preprocessor="QUANT_ABS_0_f64(0.000001)" and Component="BIT_4 CLOG_4 DBEFS_4 DIFFMS_4 DIFF_4 HCLOG_4 RLE_4 RRE_4"

more shifts towards lower SDD values. Applying different compression configurations almost generates similar charts, as shown in Fig. 6.

There are 2 possible conclusions from these results:

1. The shift from double precision to single precision shows that the compression tool performs lossy compression and reduces the precision and accuracy of the values in 2 brick elements.

2. The higher the count of SDD values below SDD5/SDD6, the more concerning the errors are. In our results, we don't see any values below SDD5 except in the output charts, as they are accumulated errors.

6 FUTURE WORK

The current state of the project completes all the proposed tasks. Following might be possible tasks if we end up extending this work for a publication:

1. Crusher/LC supports using GPUs for parallelly compressing and decompressing the inputs; extending it, and exploring more different types of compression configurations might be possible.

2. The data captured from BrickLib is currently in files which is an input to Crusher/LC, file I/O is slow, using in-memory data structures can speedup the interfacing between these tools for large input sizes.

3. Experimenting with large domain sizes and brick dimensions for BrickLib might be interesting to understand the results and error accumulations.

4. Extending to more higher order stencils like (5point, 9point, 13point, 17point, 21point, 25point, 27point) star and Box shaped stencil patterns.

5. For achieving performance benefits from BrickLib, preserving the data layout is important. Future work should preserve the layout of bricks during the CDC pipeline.

7 DISCUSSION

Learnings:

1. This project helped me explore the working and source code of BrickLib, this would help in future related work of applying it to other domains.

2. Learnt how to apply various floating point verification techniques including Significand Decimal Digits.

Challenges overcame:

1. Initially reading and understanding complex and highly Templatized C++ BrickLib code was challenging. I divided the codebase into sections and started with the most basic section, building up from there.

2. Understanding and applying SDD techniques was difficult as I lacked domain knowledge from Numerical methods references from Khalid et. al's work.

3. Making sure the verification output is correct was challenging as the output data generated from BrickLib is quite large, and usually, c++ output streams don't print with such large precision unless specified; I reduced the domain sizes and verified the floating point values by printing the precision till 15 places.

8 CONCLUSION

In this work, we explored interdisciplinary techniques of applying verification to stencil computations, specifically on a performance-portable fine-grained data-blocking library called BrickLib. We started with the fundamental question by asking what would be the performance-accuracy and precision tradeoffs when applying compression-decompression techniques on input data to stencils. We verified CDC floating point 3D 7point Stencils with the baseline for various compression techniques and found that the lossy compression causes loss of precision, leading to a shift in outputs from Bricks. The code base consists of 51+ commits with a delta of 1429 additions and 87 lines of deleted code. The experiments are open-sourced[4], along with instructions to reproduce results.

ACKNOWLEDGMENTS

Thanks to Professor Ben Greenman for teaching the class and for all the insights and feedback provided. Discussion with Khalid helped to implement and understand SDD techniques, and thanks for open-sourcing the codes.

REFERENCES

[1] [n. d.]. Center for High-Performance Computing. https://www.chpc.utah.edu/. Accessed: April 20, 2024.

[2] Khalid Ahmad, Hari Sundar, and Mary Hall. 2019. Data-driven mixed precision sparse matrix vector multiplication for GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 16, 4 (2019), 1–24.

[3] Martin et al. Burtscher. [n. d.]. LC-framework. https://github.com/burtscher/LC-framework/ Accessed: April 22, 2024.

[4] Sameeran Joshi. [n. d.]. Verify-gravel. https://github.com/Sameeranjoshi/verify_gravel/tree/main Accessed: April 22, 2024.

[5] Tuowen Zhao, Samuel Williams, Mary Hall, and Hans Johansen. 2018. Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 59–70. https://doi.org/10.1109/P3HPC.2018.00009