Vision, Modeling, and Visualization (2023)
T. Grosch and M. Guthe (Eds.)

Leveraging BC6H Texture Compression and Filtering for Efficient Vector Field Visualization

S. Oehrl, JF. Milke, J. Koenen, T. W. Kuhlen, and T. Gerrits

Visual Computing Institute, RWTH Aachen University, Germany

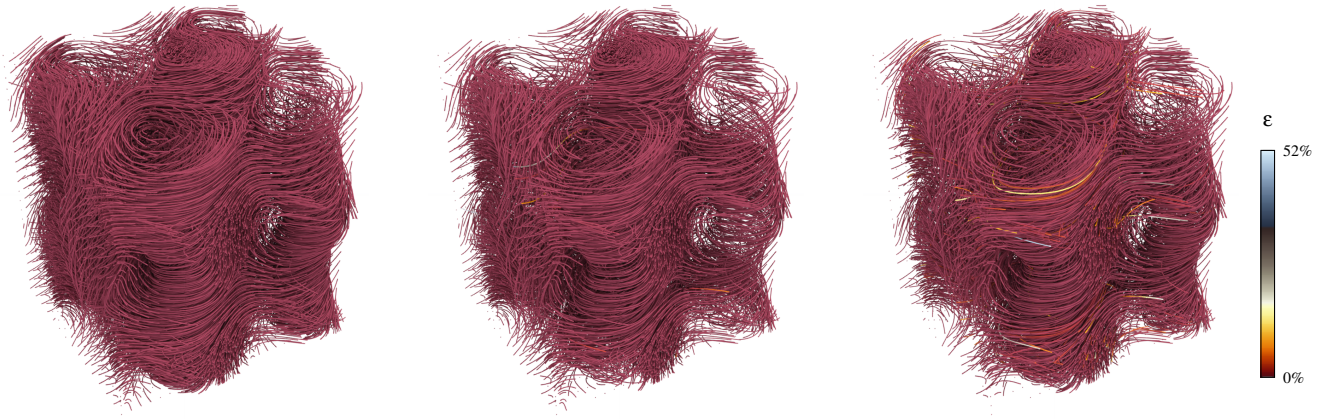


Figure 1: *Pathlines of the ABC flow. Left to right: Integrated using analytic, uncompressed, and BC6H compressed flow field data. The color indicates the per vertex distance* ε *to the ground truth (left) relative to the maximum possible distance.*

Abstract

The steady advance of compute hardware is accompanied by an ever-steeper amount of data to be processed for visualization. Limited memory bandwidth provides a significant bottleneck to the runtime performance of visualization algorithms while limited video memory requires complex out-of-core loading techniques for rendering large datasets. Data compression methods aim to overcome these limitations, potentially at the cost of information loss. This work presents an approach to the compression of large data for flow visualization using the BC6H texture compression format natively supported, and therefore effortlessly leverageable, on modern GPUs. We assess the performance and accuracy of BC6H for compression of steady and unsteady vector fields and investigate its applicability to particle advection. The results indicate an improvement in memory utilization as well as runtime performance, at a cost of moderate loss in precision.

CCS Concepts

• ***Computing methodologies*** → *Image compression; Graphics processors;* • ***Human-centered computing*** → *Visualization;*

1. Introduction

Vector fields describing phenomena such as fluid flows are a common source of data in scientific visualization, originating from various domains such as computational fluid dynamics, magnetic fields, or gradient fields. Visualization techniques, such as particle tracing and topology extraction are already well established today. Yet, measurement and simulation methods continue to create datasets of ever-increasing resolution, and hence size. This leads to several challenges to the visualization process: Limited memory of hardware, in particular graphics processing units (GPU), limits the amount of data that can be processed at a time. Although out-of-core methods can be used when the data size exceeds the hardware, this often comes at the cost of reduced interactivity and a higher development effort. In the last two decades, GPUs have evolved from graphics-specific hardware into general-purpose processors, capable of outperforming compute processing units (CPU) for special-

DOI: 10.2312/vmv.20231238

ized numerical operations. As a result, they are increasingly used for compute operations on visualization pipelines [BHP15]. Yet, the on-board memory of GPUs is limited in size, fixed per model, and does not grow at the same pace as compute power. Beyer et al. [BHP15] even propose that technology will never catch up to fit the entirety of sophisticated datasets into the GPUs memory at once. Treib et al. [TBR*12] showed, that in typical vector field visualization approaches, only 1% of the processing time is spent on transforming the data into graphics while the remaining time is spent on I/O operations of the GPU paging data from memory or disk. Therefore, several approaches were introduced in order to circumvent the memory limitations of GPUs [BRGIG*14, BHP15] such as using lossy compression of the data, which allows for significant reductions in data size at the cost of accuracy. Noticeably, modern GPUs provide explicit hardware acceleration strategies to specifically handle compressed data without the need for involved code development. Therefore, it sounds promising to leverage such functionality also for scientific data. To the best of our knowledge, this has not yet been explored within the scientific literature.

Therefore, in this work, we investigate if BC6H, a hardware-accelerated block-based compression format widely supported by modern GPUs, can be used to improve the utilization of limited memory of GPU hardware for vector field visualization. In particular, we study the effect of using compressed vector fields in terms of performance and accuracy and discuss the impact on the development process of visualization tools. Particle advection is chosen as a classical problem in flow visualization to assess the inaccuracies introduced by the lossy nature of the compression format. We produce visualizations for a variety of 3D time-dependent flow data sets with different resolutions and flow behavior. Using different code and parameter choices, we benchmark the I/O and compute performance, thus providing a thorough investigation of their impact on the visualization results. Finally, we discuss these results and derive, whether this technique represents a suitable candidate to improve runtime in addition to memory usage without requiring involved coding strategies.

2. Related Work

Lossy compression methods for scientific visualization have been investigated in several works [CPW*19, TBR*12, LMG*18]. Among these, GPU-driven applications appear to benefit drastically from reduced I/O when operating on large datasets. The highest benefit from decompression is achieved when applied late in the visualization pipeline, most preferably in a local manner on element access [BRGIG*14]. A widely used approach in reducing data size at the expense of quality is transform coding. Often occurring as discrete wavelet transform (DWT), the main idea is to reconstruct the input signal via a combination of weighted basis functions. This method is typically lossless by design but can be made lossy when only a selection of the original basis functions is used for signal reconstruction. While these approaches allow for a variable compression-quality tradeoff, their non-deterministic compression ratio typically forbids random access in the compressed volume which thus needs to be decompressed completely before access. To ease this, DWT is often applied to a bricked representation of the input volume. An in-depth overview of wavelet transform in the context of 3D turbulent flows is given by Rinoshika et al. [RR20]. Treib et al. [TBR*12] presented a technique, where a 3D turbulent flow is first subdivided into equally sized bricks with some overlap to correctly interpolate at brick boundaries. GPU-accelerated DWT is then applied to the individual bricks whose coefficient stream is encoded by run-length (RLE) and Huffman coding. This led to an average compression rate of 32 : 1, reducing I/O time at the price of decompressing the data on the GPU. Li et al. [LGP*15] investigated different discrete wavelet transform (DWT) strategies in the context of turbulent flow data. Their test data was 256GB of turbulent flow for which they could achieve compression ratios ranging from 8 : 1 to 512 : 1 but with a high cost of introduced reconstruction time. Hoang et al. [HKB*18] explored different approaches to reduce the filesize in the context of data visualization, especially for isosurface extraction and gradient field visualization. Using several datasets, they found that a wavelet representation achieved the best results for task- and data-independent encoding. Liang et al. developed an error-bounded and feature-preserving lossy compression technique for vector fields in [LGD*20]. They applied a custom algorithm for on-the-fly and offline compression which used an interchangeable lossy compressor. Using the lossy SZ compressor, they achieved a 7.48 : 1 compression ratio for 3D vector data while preserving all first-order critical points where the Jacobian of the vector field is not equal to zero vector. Another approach for lossy compression is tensor decomposition. Tucker decomposition coins a popular approach in this field where the input data is considered a higher-order matrix which is decomposed via Singular Value Decomposition to a smaller approximation [LMG*18]. Ballester-Ripoll et al. [BRP16] applied Tucker truncation in combination with thresholding for lossy compression on multidimensional data focused on 8-bit CT scans. This approach, however, showed high element access times per tensor. Deep learning methods such as the use of convolutional neural networks (CNNs) are used as means to reduce the size of the flow field before analysis and increase the performance of the overall process. Kim et al. [KAT*19] presented a generative network explicitly designed to reconstruct fluid velocities, achieving fast visualizations and high compression. Glaws et al. [GKS20] presented a convolutional autoencoder able of factor 64 in-situ data compression, explicitly designed for 3D turbulent flow simulations. A detailed overview of other approaches in this area is given by Liu et al [LJW*22]. Making use of such networks is a promising field recently gaining popularity. As of now, however, these approaches are still less deterministic in behavior than other, more traditional methods and typically rely on training the neural network on the dataset first. Quantization schemes, e.g., scalar quantization or more complex vector quantization, remap values or groups of values such that their quantized representation has a smaller memory footprint. They tend to have limited compression capabilities and are therefore often composed with other techniques, e.g., uniform scalar quantization for encoding the coefficients of DWT coded data [LMG*18].

Reiterating current literature shows that most solutions require involved algorithms and development efforts to gain performance. Here, we want to investigate if similar results can also be achieved by relying on the built-in hardware support for accessing compressed textures.

3. Implementation

Followingly, we first describe the BC6H encoding and its properties, give a brief overview of the comparative cudaCompress encoding, and explain the applied visualization method.

3.1. Encoding

All existing GPU-native compression schemes are a form of block-based compression and are mostly used for encoding integer scalar fields in the context of scientific data [BRGIG*14]. For an encoding to be considered a suitable candidate for the given visualization method, two core specifications need to be fulfilled:

- Native support for encoding floating point data
- Support for hardware-accelerated decompression on modern GPUs

Currently, these requirements reduce the set of qualifying encodings to BC6H and ASTC as both are designed for high-quality floating-point data compression. The hardware support for ASTC is limited to weaker mobile GPUs and integrated graphics with smaller and slower video memory while dedicated GPUs support BC6H only. As, in our experience, the target group for scientific visualization software predominantly relies on desktop computers with dedicated GPUs, we decided to build our investigation on the BC6H compression method. The performance of the BC6H compression is compared against *cudaCompress*, a DWT-based encoder that was developed by Treib et al. [Tre14b] and is specifically tailored towards the compression of scientific data in the context of large scale visualization and the application on GPUs.

BC6H Compression BC6H is a block-based compression that reaches high encoding/decoding rates on GPUs due to special decoding chips. It is designed to encode HDR three-channel color spaces with an internal precision of 16 bits and a mantissa length of 10 to 11 bits depending on whether a sign bit is necessary. The basic idea behind BC6H is to spatially subdivide the data into 4x4-sized 2D blocks with a fixed memory footprint of 128 bits and encode each block individually. This design enables random-access traversal of the data as the position of an encoded texel can be computed and only needs the respective block to be decompressed. Inside a block, each texel is replaced by an interpolant weight with limited precision which can be used with additionally saved color endpoints to reconstruct the former values. This block-based approach might introduce discontinuities when transitioning from one block to another. BC6H employs additional measures like partitions to reference different color gradients per weight and multiple encoding modes to best describe the input data. The computational complexity lies in finding the best mode, partitioning, and color endpoints to best describe the content of the respective block which is often solved by an iterative process encoding the data and comparing it using some error function. Due to this nature, this is a lossy compression that effectively achieves constant $6:1$ compression ratios for 3-channel half-float data. Any more precise floating-point data will be converted to half float before compression.

cudaCompress The cudaCompress compression library was developed by Treib et al. [Tre14b] for efficient large scale visualization on GPUs in 2014. It uses a combination of DWT, Huffman coding, and RLE to achieve a high compression ratio at only a small cost in precision for floating point data or even no cost for integer data. The implementation is based on C++ and CUDA to allow efficient utilization of the GPU. Contrary to BC6H, the cudaCompress encoding needs to be decompressed completely before it can be used for integration. This implies the presence of auxiliary data buffers which need to allocate enough memory to hold the uncompressed data and the need to brick the data into smaller subvolumes, e.g., individual time-slices, to circumvent too high memory usage. Depending on the chosen approach, halo regions around the subvolumes containing information about the neighboring subvolumes are necessary to allow interpolation at the border regions, reducing the compression efficiency. These problems are common to compression techniques used for scientific data.

3.2. Data Compression

Applying BC6H compression to existing data is made very easy, as several open-source compression frameworks are available. The *NVIDIA Texture Tools Exporter* [NVI21] proved to be the best-achieving candidate concerning quality and coverage of the official BC6H specification. We consider this a pre-processing step that is often not as time-critical compared to efficient loading and visualization. Nonetheless, several parameters influence the compressed data: The time spent on finding a good encoding of the data is tuned via five presets ranging from “fast” to “highest”. At the time of the experiments, the implementation supported CUDA-accelerated compression up to the “medium” preset. As the availability and achievable quality of BC6H are strongly implementation dependent and may improve in the future, we are interested in both the “medium” and “highest” presets as they were the best available presets for GPU- and CPU-based compression. We input 32-bit floating-point data to the encoder which is then transformed to 16-bit floating-point precision first and subsequently encoded which leads to a total file size reduction ratio of $12:1$.

For the cudaCompress encoding only the C++/CUDA implementation by Treib [Tre14a] exists. Using the library is less easy as the exposed parameters require a certain level of domain knowledge for the applied compression techniques at the benefit of more control over the compression. Meaning, the encoding is steered more directly via the quantization step size and the number of decomposition levels, compression iterations, and huffman bits. We use default values of the implementation for each parameter in our approach except for the number of huffman bits for which we need to choose either 14 bits or 16 bits, depending on the dataset. The achievable compression ratio of cudaCompress is hardly predictable and dependent on the chosen parameters, as well as the dataset itself, but is capable of a file size reduction ratio of 100:1. Also, contrary to BC6H, it can encode and reconstruct 32-bit floating-point data with a small error owed by the lossy compression.

3.3. Integral Line Computation

We developed a prototype particle tracer in C++ and Vulkan designed to load the entire dataset into video memory before start-

ing the integration process, i.e., no out-of-core loading was implemented. A dataset is stored on the GPU as a set of 3D textures where each texture stores a single timestep using either compressed BC6H or uncompressed 16/32-bit floating point texture formats. Hardware-accelerated texture sampling is used to efficiently interpolate the data within single time slices. However, texture sampling is performed with less spatial precision on most GPUs, leading to larger errors compared to manually interpolating neighboring values. In the following, the former will be called *implicit interpolation* while the latter will be called *explicit interpolation*. For compressed BC6H data, initial tests indicated a performance benefit of using 2D texture arrays instead of 3D textures, even when performing implicit interpolation. This limits the hardware-accelerated interpolation to the x,y-plane while performing manual interpolation in the z and t dimensions. For uncompressed textures, this was not the case, thus, implicit interpolation for BC6H will utilize 2D texture arrays while implicit interpolation for uncompressed textures will use 3D textures. Additionally, in the case of uncompressed texture formats, the data for one time slice was split across three textures, one for each component of the vector, as GPU hardware does not support 3D textures with 3-component formats. An alternative approach would be to use 4-component texture formats, however, this would waste 25% of memory which is critical in memory bandwidth-bound applications and was slower in our tests. Integration was performed solely on the GPU using compute shaders where each thread is assigned a single particle which it traces until it terminates by reaching either the target length or by colliding with the field boundaries. The employed integration method was Runge-Kutta 4th-Order with fixed stepsize in conjunction with linear interpolation in each spatial and temporal dimension.

4. Experimental Setup

	ABC Small	Tangaroa	Half Cylinder
Resolution	$225 \times 250 \times 200 \times 151$	$300 \times 180 \times 120 \times 201$	$640 \times 240 \times 80 \times 151$
FP32 Size	20.38GB	15.63GB	22.26GB
BC6H Size	1.73GB	1.30GB	1.85GB
cC Size	151MB	186MB	236MB
Seeding Density	$25 \times 25 \times 15$	$20 \times 25 \times 15$	$25 \times 25 \times 12$
Path Lines	9375	7500	7500
Stepsize Δt	0.01	0.01	0.01
Steps	15,100	20,100	15,100

Table 1: *The investigated datasets with core descriptors and applied integration parameters.*

Three datasets describing time-dependent 3D vector fields at 32-bit floating point precision were investigated within the scope of this work. An overview of each dataset's core descriptors and the applied integration parameters are given in Table 1. All investigated datasets had time-varying characteristics which introduced a very high memory demand. The *Tangaroa* and *Half Cylinder* denoted datasets both exhibited an overall relaxed flow except for a localized region of strong turbulence. The Tangaroa dataset describes the airflow distortion caused by turbulence around the research vessel Tangaroa [PSS04]. The second dataset refers to an incompressible flow around a Half Cylinder [BRG19]. Finally, we use the analytical ABC Flow to compare the effects of compression against a true solution. The selected region shows five cylindrical vortexes with especially dynamic behavior between shared vortex boundaries. Using the formula provided by Shi et al. [STW*08], we used an ABC flow field with $A = \sqrt{3} \cdot 0.05t \cdot \sin(\pi \cdot 0.01t)$, $B = \sqrt{2}$, $C = 1$ and $s = 0.05$.

To investigate the impact of the compression on runtime and accuracy, we compute several standard metrics during and after the integration of path lines at fixed and evenly distributed seeding locations in every dataset. This is done for BC6H compressed, as well as uncompressed data with the parameters listed in Table 1. This process is repeated 100 times to compensate for fluctuations in performance due to system work and allow for standard statistical analysis. Performance is described by measuring compression, I/O, and integration times. To further provide a comparison to existing solutions, we implemented the *cudaCompress* approach by Treib et al. [TBWW15] and recorded the same metrics. As cudaCompress encoded data needs to be fully decompressed on the GPU before it can be used, we process it similarly to uncompressed data.

Two workstations were used within the scope of this work. The preprocessing stage, which mainly consists of compressing the input data, happened on a powerful workstation with two Intel Xeon E5-2680v3, an NVIDIA Quadro RTX 6000 with 24GB video memory, and 128GB DDR4 system memory. The visualization part of this study was performed on a regular desktop PC with a single Intel Core i9-10900X, an NVIDIA RTX 3090 with 24GB video memory, 32GB DDR4 system memory, and an SSD. We provide our application and error calculation as open-source code available at `https://github.com/VRGroupRWTH/bc6h-integrator`.

4.1. Error Metrics

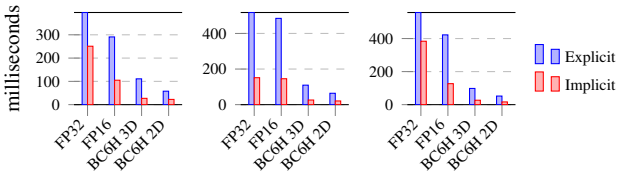
As integral lines were chosen as a standard visualization problem, we investigate the geometric error by aggregating the unique trajectory pairs between the source and compressed representation into curve similarity metrics. The most apparent approach is to measure the average and maximum Euclidean distances between each unique vertex pair of two corresponding trajectories. However, this measure does not account for the similarity curve *shapes* as the compared trajectories may follow a similar path at different velocity magnitudes. This effect can be mitigated when taking time-parameterization along the trajectory into account, e.g., to allow matching a single vertex along one trajectory with multiple others on the second trajectory if this provides a lower distance cost compared to strictly relying on the given input series of points. Concerning a time-aware error alternative, we, therefore, measure the similarity of trajectories by the distance under *Dynamic Time Warping (DTW)* and *Fréchet Distance* [EFV07]. DTW is a sum-based measure that calculates a semantic average distance between two curves if divided by the number of matches. It matches each point along an input trajectory with the closest one on the compared trajectory adhering to certain heuristics, i.e., a guaranteed comparison from the first until the last input vertex and respecting the monotonic increasing nature of time for indexing the vertices during comparison. Similarly to DTW, the Fréchet Distance adheres to certain heuristics which guarantee to start comparison from the first vertex until the last while respecting time progression, not taking past vertices into account for matching. Additionally, we applied the *Clamped Divergence Rate* introduced by Treib

et al. [TBWW15]. It computes the rate at which two trajectories diverge instead of their distance and is clamped when a certain distance Δs is met. The Δs limits the impact a critically diverged trajectory has on the obtained error, as it is not tied to the field error anymore once it diverged too much from the ground truth. Here, the Δs was set to the same value as the authors of the Clamped Divergence Rate, equal to the grid spacing.

5. Results

In the following, we present and analyze the results of our experiments. While visually, the resulting visualizations are very similar as shown in Figure 1, there are significant differences in the process that the data points out. First, we investigate possible performance gains by looking at the impact of data transfer and data access on computation times. After that, we analyze the error introduced by the compression.

5.1. Performance Analysis



(a) *ABC* **(b)** *Tangaroa* **(c)** *Half Cylinder*

Figure 2: *Average complete integration time for all path lines.*

	BC6H		
Dataset	Medium (GPU)	Highest (CPU)	cudaCompress
Half Cylinder	108.2	3204.6	49.5
Tangaroa	135.0	2474.2	48.5
ABC	170.2	4339.6	52.2

Table 2: *Time in seconds needed to compress the vector field data.*

Table 2 shows the time demand of dataset compression for both encodings and both BC6H presets. The BC6H encoder used only offers a CPU-based compression at the highest quality preset, which introduced a significant initial overhead, while medium quality preset and cudaCompress encoding can be performed much faster on the GPU. The cost of CPU-based compression arced above one hour for the ABC dataset, marking it unsuitable for a fast preview of the input data in a compressed representation. The GPU-based compression achieved a significantly faster encoding in less than 3 minutes for the same data. The implementation of the cudaCompress encoder performed fastest, needing less than half the time than the GPU-accelerated BC6H encoding. Generally, for BC6H the encoding time was very dependent on the capability of the encoder to find a good solution. When looking at measurements taken on the compressed data, two results are clearly visible in the recorded data: A strong increase in performance during integration as well as in I/O operations when using BC6H compression. Figure 3 shows the time spent on file and memory I/O. Unsurprisingly, it

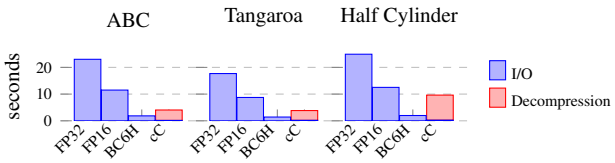


Figure 3: *Time spent on file and memory IO for the different datasets and formats as well as the decompression in case of cudaCompress.*

scales with the size of the data and favors the compressed over the uncompressed data, which is why cudaCompress significantly outperforms BC6H, due to the higher compression ratio shown in Table 1. Once the data is stored on the GPU, cudaCompress needs an additional decompression step which is not needed for BC6H. This not only drastically increases processing times as shown in Figure 3, such that when adding I/O and decompression times, data compressed with cudaCompressed performs slower than BC6H but still faster than raw 16-bit floating point values. As the decompression of BC6H textures is hardware-accelerated and happens locally during texture access only, this introduced no measurable overhead, which is a major advantage of this compression method. It also means that if uncompressed cudaCompress data exceeds the GPU memory, involved streaming strategies have to be applied. Figure 2 shows the average time spent on pathline integration. The observed speedup of using BC6H in the integration depended on the investigated dataset and ranged from factor 11.14× up to 23.28× compared to integration using the original data. Even if compared to raw 16-bit floating-point precision data, which equals the precision of BC6H, the encoded representation achieved a 7.7× faster result. The nature of cudaCompress encoded data does not allow for any further comparison than to stage BC6H against raw floating-point precision performance. This already indicates that the BC6H architecture achieves an additional acceleration of the integration next to the IO reduction. Due to the combination of both faster I/O and integration times, the process of creating a path line visualization for the Half Cylinder data set could be reduced from two minutes down to a second. The choice of explicit or implicit accelerated linear interpolation had a significant impact on the integration times as well. If the slightly more precise explicit interpolation is favored, the integration time grew by a factor of 3.41× compared to implicit interpolation.

Dataset	FP32	FP16	BC6H
Half Cylinder	3	4	10
Tangaroa	3	4	10
ABC	5	5	15

Table 3: *Empiric optimal number of parallel threads for best integration performance.*

Particle advection is trivially parallelizable as the individual trajectories do not depend on each other. However, as the algorithm's performance is primarily limited by memory bandwidth and not compute power it is not feasible to trace a trajectory with every available hardware thread in parallel. Doing this causes more cache

invalidation and results in weaker performance compared to using fewer threads which we could also observe in our application. Reducing the data size through compression not only reduces video memory requirements but also reduces the required memory bandwidth and improves caching behavior. In our application, we observed that we could use more threads when integrating datasets that were compressed using BC6H compared to raw datasets. Table 3 shows the number of threads that lead to the best result during our integration benchmarks. This clearly shows a significant increase when using smaller data, which contributes to its improved performance.

5.2. Accuracy Analysis

The results of the proposed error metrics for the presented data sets are summarized in Figure 4, showing distances aggregated over all computed trajectories. As noted in Section 4, implicit interpolation in z-direction could not be performed for encoded data because of the chosen data structure. Implicit temporal interpolation was impossible due to the lack of a native temporal data structure for textures. However, favoring the texture array over other data structures that allow implicit interpolation in the z-dimension still proved to be faster and offered a higher precision when translating the world position to the field position in our tests. As we only had an analytic solution for the ABC dataset and used presampled data as ground truth for all other observations, we obtained and differentiated the error introduced by linear interpolation for the ABC field only.

An observation common across all datasets is the decrease in accuracy growing with the density of turbulence in the source data. The ABC dataset being the most turbulent flow field investigated indicates that linear interpolation already introduces a notable error in the integration. While cudaCompress shows a comparable error for all metrics, BC6H performs significantly worse. While Figure 1 shows that the overall flow is still preserved, it produces more stray trajectories, especially in highly dynamic regions. Using the "highest" instead of the "medium" preset for compressing the dataset only had a small impact on the resulting error. The ABC and Half Cylinder datasets even preferred the medium preset when considering the maximum vertex error, DTW, or Frechet distance.

6. Discussion

The conducted experiments aimed to shed light on the question of how a GPU-native compression format performs in the domain of vector field visualization. Particle tracing with line integration was chosen as the visualization setup due to its commonness in that domain and the ability to effectively catch and progress any error introduced by the chosen integration method. As mentioned in Section 3, BC6H was investigated instead of ASTC which was the only other viable candidate for float vector input data. Although ASTC compression allows for more configuration options in their specification and is a valid competitor to BC6H, it is not widely supported on discrete desktop GPUs if at all. To enable comparison to non-native GPU encodings that benefit from optimizations for scientific data we selected cudaCompress from a wide range of options, as it was developed specifically for the use case of large-scale visualization on GPUs.

The first observation across the chosen datasets was BC6H's error susceptibility to regions with lots of dynamic behavior. For example, the ABC dataset created the most dynamic trajectories and showed to be the worst-case scenario for both encodings and linear interpolation itself. While cudaCompress performed similarly erroneously during interpolation, BC6H introduced an error that was larger by one magnitude across all metrics. This error already suggests traces to BC6H's architectural design, where a single block region describes 16 3D vectors approximated via 2 pairs of 3D vectors, leading to hard limitations when this region also contains dense turbulence. Prior investigations of the datasets, 2D image representations also showed few random block artifacts in otherwise smoothly encoded regions which additionally suggest a non-optimal implementation of the applied encoder, which also contributes to the perceived errors. The Tangaroa and Half Cylinder datasets contained large areas of laminar flow around a strictly contained turbulent subarea and exhibited the lowest error scores for BC6H which, however, still remained a magnitude greater relative to cudaCompress and linear interpolation. The spatial distribution of the error indicates that highly laminar flow could be preserved well and the error is constrained to regions of strongly diverging vectors. Due to the lack of an analytical solution for these two datasets, the selected ground truth for comparison inherited the error from linear interpolation and constrained the expressiveness of these findings.

When analyzing the *Medium* and *Highest* encoding preset of BC6H, it shows that the optimizer of the chosen encoder could not reliably find a better encoding under all metrics. In the case of the ABC dataset, the true error is even higher in the case of the presumed higher encoding across all metrics except the divergence rate. A more thorough investigation is needed to see the effect of the encodings. Concerning performance, a massive overall reduction in runtime could be achieved by applying BC6H compression. The achieved speedups of near factor 39× enabled interactive exploration of the data and reduced waiting times for the completion of an integration run from 9 seconds down to 250 milliseconds. As the achievable performance was primarily memory-bound, the reduction of file size was most responsible for the speedup of BC6H compression. The compute stage itself could also be cut down by a factor of 3× in the most expressive scenario.

When compared to similar studies, other approaches regularly investigate more conservative methods which use adaptive encodings that result in different file sizes depending on locally encoded data regions [GK11, TBWW15]. This benefits by restricting the memory reduction in regions where the precision loss would be too great and vice versa. Resulting in a generally more precise reconstruction, this design also comes with a cost. E.g., these encodings typically are unable to perform random access traversal on the compressed data volume since the flow field position can't be mapped to its encoded position without an accompanying lookup structure. Additionally, they require larger streams of data to be decoded first and have them copied into another data buffer before. To counter the need to decompress the complete data volume in the GPU's limited video memory, the source data is often bricked into smaller regions before compression. This, on the other hand, increases the file size as each block then requires the data of neighboring blocks around at its boundaries to correctly encode these neighboring re-

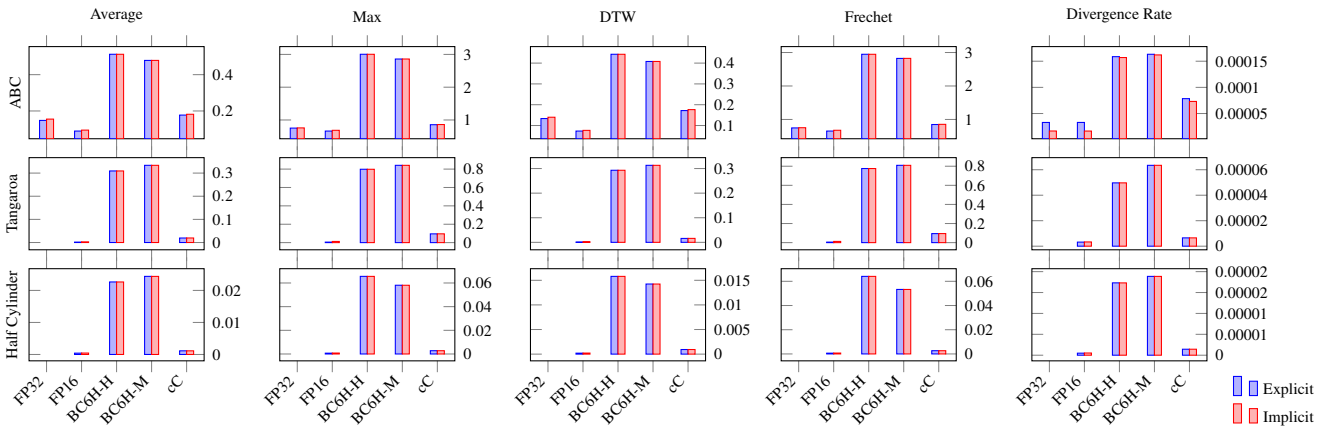


Figure 4: *Accuracy of the pathlines generated based on compressed and uncompressed datasets. The ground truth for the ABC dataset was the analytic solution while the ground truth for all other datasets is the sampled 32-bit floating point with explicit interpolation. The plots show the RMS of the average vertex error, maximum vertex error, DTW, and Frechet distance as the percentage of the maximum possible distance defined by the diagonal through the three-dimensional domain. The final row shows the divergence rate defined by Treib et al [TBWW15]. Each plot shows the error metric for 16 and 32-bit floating point numbers (FP16 and FP32), BC6H compression using the highest and medium preset (BC6H-H and BC6H-M), and cudaCompress (cC) for implicit and explicit interpolation (blue and red respectively).*

gions without any discontinuities. Thus, while BC6H has a fixed compression ratio and limited accuracy due to internal precision and design, it accelerates the decompression by the lack of translation procedures in the Compute Shader. The cost of BC6H is more imminent if compared to other fast encodings like cudaCompress. In the most dynamic scenario, the trajectories of BC6H are off by up to 0.4% under the DTW average measure. This is significantly greater than the error of linear interpolation or cudaCompress but may be acceptable in certain scenarios, as the trajectories are still near the true trajectory.

An additional impact to be mentioned is the applied encoder. Here, the NVIDIA Texture Tools Exporter [NVI21] which builds upon the NVIDIA Texture Tools [CN20] was used to encode the data into BC6H, but other open source candidates like Intels ISPC Texture Compressor [Int22] or AMDs Compressonator [AMD22] also provided implementations of BC6H encoders. All encoders use a quality measure to control the encoding effort, which, e.g., in the case of the ASTC encoder, is the Peak Signal-to-Noise Ratio (PSNR) metric [Arm22]. The PSNR metric is specifically tuned toward image quality and it is not clear which measure was implemented for the encoder which was applied to our studies. It is, however, highly possible to better retain features of the flow field if a custom encoder optimizing for vector fields is employed. Especially in consideration of the avoidable blocky artifacts which could be observed in the image representation of the datasets, a more stable implementation alone suggests better results. The mentioned internal precision of 16bit half float with only 10 or 11 bits for the mantissa is, however, a major drawback. The smaller representable maximal value is probably negligible, but losing 12 bits of precision in the mantissa can be considered quite major. To eliminate the error introduced by the smaller precision, we also sampled and evaluated 16-bit precision datasets. In the case of the ABC dataset, we observed lower error ratings in aggregated representation for the 16-bit datasets compared to the full 32-bit precision. This is a phenomenon we could not explain as no changes in the experimental setup between these two cases were made except for the data format. This needs to be evaluated in further studies.

In the light of specifically tailored encodings like cudaCompress, the applied BC6H encoding performs significantly worse concerning precision and compression ratio. It does, however, show significant speedups for the integration itself and more compute-heavy visualizations can benefit from this. The possibility to hold the compressed dataset in the VRAM for caching purposes, without the need for out-of-core decompression, is a huge advantage with respect to interactive visualization and incentivizes a more in-depth analysis under a broader range of parameters. The most potential for better performance for a GPU-native encoding is to be expected in two directions. First, a custom encoder with vector-optimized heuristics and more stable results has the potential to greatly benefit the results. Further, Vaidyanathan et al. [VSW*23] published a preprint of a next-generation GPU encoding that elevates the use of neural networks for compression. While this specific publication yet only covers integer-based datasets, they achieved promising results with respect to accuracy which is to be expected to map to floating-point-based datasets in the future as well.

7. Conclusion

In this work, we investigated if the hardware-accelerated BC6H format can be used to overcome memory limitations and improve performance of vector field visualization. To do so, we applied a number of error and performance metrics on a variety of time-dependent vector fields and provided a compact study on possible parameter and interpolation impact. We showed that using this widely-supported compression, allows for several performance improvements without the need for involved coding strategies, as the

format is widely supported on modern GPUs. For one, these improvements include faster data transfer due to the reduced memory gained from the 6 : 1 compression rate. Further, our experiments showed, that the hardware-accelerated texture access drastically increased particle tracing performance on BC6H-compressed vector fields. While these are promising results, we also showed that there is a noticeable error introduced by the compression. While all tested error metrics showed an introduced deviation from the original values, we were also able to show that the accuracy of the format is highly correlated with the complexity of the dataset, with smoothly varying regions being less prone to error than highly turbulent ones. Based on these findings, we conclude that BC6H represents a simple-to-implement technique to reduce data transfer and access, improving computation times and allowing even large data sets to be processed efficiently that would normally not fit on the GPUs memory or can now be used on GPUs with smaller memory. The loss in accuracy, however, is clearly noticeable in applications such as particle advection with long integration times, where small errors accumulate quickly, such that we discourage the use of the compression when high accuracy even after long integration times is needed. Overall, we established BC6H as a tool to trade accuracy for memory and runtime performance in the context of vector field visualization. In the future, we want to investigate custom error functions for encoding vector fields, which could further improve the accuracy. Although BC6H is capable of optimizing for angular error, the implementation is limited to normal maps, rather than general vector fields. Another direction is packing 32 and 64-bit numbers to short floats to overcome the channel limitations of BC6H. Comparison of BC6H to alternative formats also remains as future work.

References

[AMD22] AMD: Compressonator, 2022. URL: https://github.com/GPUOpen-Tools/Compressonator. 7

[Arm22] ARM: Astc encoder, 2022. URL: https://github.com/ARM-software/astc-encoder. 7

[BHP15] BEYER J., HADWIGER M., PFISTER H.: State-of-the-art in gpu-based large-scale volume visualization. *Computer Graphics Forum 34*, 8 (2015), 13–37. 2

[BRG19] BAEZA ROJO I., GÜNTHER T.: Vector field topology of time-dependent flows in a steady reference frame. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Scientific Visualization)* (2019). 4

[BRGIG*14] BALSA RODRÍGUEZ M., GOBBETTI E., IGLESIAS GUITIÁN J., MAKHINYA M., MARTON F., PAJAROLA R., SUTER S.: State-of-the-art in compressed gpu-based direct volume rendering. *Comput. Graph. Forum 33*, 6 (Sept. 2014), 77–100. 2, 3

[BRP16] BALLESTER-RIPOLL R., PAJAROLA R.: Lossy volume compression using tucker truncation and thresholding. *The Visual Computer 32*, 11 (2016), 1433–1446. 2

[CN20] CASTAÑO I., NVIDIA: Nvidia texture tools, 2020. URL: https://github.com/castano/nvidia-texture-tools. 7

[CPW*19] CHEN J., PUGMIRE D., WOLF M., THOMPSON N., LOGAN J., MEHTA K., WAN L., CHOI J. Y., WHITNEY B., KLASKY S.: Understanding performance-quality trade-offs in scientific visualization workflows with lossy compression. In *2019 IEEE/ACM 5th International Workshop on Data Analysis and Reduction for Big Scientific Data (DRBSD-5)* (2019), pp. 1–7. 2

[EFV07] EFRAT A., FAN Q., VENKATASUBRAMANIAN S.: Curve matching, time warping, and light fields: New algorithms for computing similarity between curves. *Journal of Mathematical Imaging and Vision 27*, 3 (2007), 203–216. 4

[GK11] GOLEMBIOVSKY T., KRENEK A.: Compression of vector field changing in time. In *Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10)–Selected Papers* (2011), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 6

[GKS20] GLAWS A., KING R., SPRAGUE M.: Deep learning for in situ data compression of large turbulent flow simulations. *Physical Review Fluids 5*, 11 (2020), 114602. 2

[HKB*18] HOANG D., KLACANSKY P., BHATIA H., BREMER P.-T., LINDSTROM P., PASCUCCI V.: A study of the trade-off between reducing precision and reducing resolution for data analysis and visualization. *IEEE transactions on visualization and computer graphics 25*, 1 (2018), 1193–1203. 2

[Int22] INTEL: Fast ispc texture compressor, 2022. URL: https://github.com/GameTechDev/ISPCTextureCompressor. 7

[KAT*19] KIM B., AZEVEDO V. C., THUEREY N., KIM T., GROSS M., SOLENTHALER B.: Deep fluids: A generative network for parameterized fluid simulations. In *Computer graphics forum* (2019), vol. 38, Wiley Online Library, pp. 59–70. 2

[LGD*20] LIANG X., GUO H., DI S., CAPPELLO F., RAJ M., LIU C., ONO K., CHEN Z., PETERKA T.: Toward feature-preserving 2d and 3d vector field compression. In *PacificVis* (2020), pp. 81–90. 2

[LGP*15] LI S., GRUCHALLA K., POTTER K., CLYNE J., CHILDS H.: Evaluating the efficacy of wavelet configurations on turbulent-flow data. In *2015 IEEE 5th symposium on large data analysis and visualization (LDAV)* (2015), IEEE, pp. 81–89. 2

[LJW*22] LIU C., JIANG R., WEI D., YANG C., LI Y., WANG F., YUAN X.: Deep learning approaches in flow visualization. *Advances in Aerodynamics 4*, 1 (2022), 1–14. 2

[LMG*18] LI S., MARSAGLIA N., GARTH C., WOODRING J., CLYNE J., CHILDS H.: Data reduction techniques for simulation, visualization and data analysis. In *Computer graphics forum* (2018), vol. 37, Wiley Online Library, pp. 422–447. 2

[NVI21] NVIDIA: Nvidia texture tools exporter, 2021. URL: https://developer.nvidia.com/nvidia-texture-tools-exporter. 3, 7

[PSS04] POPINET S., SMITH M., STEVENS C.: Experimental and numerical study of the turbulence characteristics of airflow around a research vessel. *Journal of Atmospheric and Oceanic Technology 21*, 10 (2004), 1575–1589. 4

[RR20] RINOSHIKA A., RINOSHIKA H.: Application of multi-dimensional wavelet transform to fluid mechanics. *Theoretical and Applied Mechanics Letters 10*, 2 (2020), 98–115. 2

[STW*08] SHI K., THEISEL H., WEINKAUF T., HEGE H.-C., SEIDEL H.-P.: Visualizing transport structures of time-dependent flow fields. *IEEE computer graphics and applications 28*, 5 (2008), 24–36. 4

[TBR*12] TREIB M., BÜRGER K., REICHL F., MENEVEAU C., SZALAY A., WESTERMANN R.: Turbulence visualization at the terascale on desktop pcs. *Visualization and Computer Graphics, IEEE Transactions on 18* (12 2012), 2169–2177. 2

[TBWW15] TREIB M., BÜRGER K., WU J., WESTERMANN R.: Compression and heuristic caching for gpu particle tracing in turbulent vector fields. In *International Joint Conference on Computer Vision, Imaging and Computer Graphics* (2015), Springer, pp. 144–165. 4, 5, 6, 7

[Tre14a] TREIB: cudacompress, 2014. URL: https://github.com/m0bl0/cudaCompress. 3

[Tre14b] TREIB M.: *Gpu-based compression for large-scale visualization*. PhD thesis, Technische Universität München, 2014. 3

[VSW*23] VAIDYANATHAN K., SALVI M., WRONSKI B., AKENINE-MÖLLER T., EBELIN P., LEFOHN A.: Random-access neural compression of material textures. *arXiv preprint arXiv:2305.17105* (2023). 7