RWTH Aachen University

Bachelor Thesis

Performance and Error Analysis of ASTC compressed Vector Fields in GPU accelerated Particle Advection

by

Jan Frieder Milke

RWTH Aachen University

Bachelor Thesis

Performance and Error Analysis of ASTC compressed Vector Fields in GPU accelerated Particle Advection

for the degree of B.Sc. in Computer Science

by

Jan Frieder Milke
Student Id.: 348 686

Prof. Dr. Torsten W. Kuhlen
Visual Computing Institute

Prof. Dr. Leif Kobbelt
Visual Computing Institute

Supervisor: Ali C. Demiralp, M.Sc.
Simon Oehrl, M.Sc.

Date of issue: August 26, 2021

Statement

Abstract

GPU-driven visualization becomes increasingly more popular due to its immense parallel compute power on even simple workstations. But the limited amount of dedicated video memory poses a severe bottleneck for the growing demands of visualization. In order to reduce the amount of data that needs to be uploaded to the GPU this work examines the effect of lossy ASTC texture compression on vector fields in steady and unsteady condition. The impact on memory and performance are measured in conjunction with the introduced error separately in geometric space and image space. This study discovered that ASTC texture compression is applicable for vector field compression in the context of visualization, however suffers a perceivable precision loss in turbulent vector fields at the benefit of an overall significantly faster computation performance. Normalization because of the encoder's supported value range showed to have additional effect on accuracy and runtime which indicates possible optimization potential for texture compression based vector field encoding.

Contents

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Scientific visualization processes are utilized and vital for many domains to perform feature detection within large scale volumetric datasets. Vector fields are a specifically challenging subject as for their richness in information which impacts the computational complexity of visualization methods and the sheer data size as well [47]. A common visualization method for these in flow visualization is streamline integration, i.e., the construction of trajectories from traveling particles whose flow is described by the vector field [23]. The need for high resolution data in order to capture even small features within the data [47] leads to an explosive growth of spatial and temporal dimension-size, simultaneously increasing the file size [6]. The compute power needed to process massively sized files in reasonable time resulted in large scale visualization traditionally being subject to supercomputers with limited accessibility only. However, during the last decades GPUs emerged as general-purpose processing tools which are capable of immense compute power and found in most consumer level desktop PCs today [31]. A whole branch of visualization is nowadays dedicated to accessible GPU-driven approaches [4]. However, the raw GPU compute power is cut off by a major pithole: on-board memory of GPUs is limited in size, fixed per model and not growing at the same pace as processing power. *Beyer et al.* even assume that technology will never catch up to fit the entirety of sophisticated datasets into the GPUs memory at once [4]. *Treib et al.* were concerned with bandwidth limitations of GPUs in recent works as well and discovered in case of visualizing 4D flows, that only 1% of the processing time was spent on transforming the data into a visualization and the remaining time was spent on I/O operations of the GPU paging data from memory or disk [46]. Many approaches to circumvent the GPUs memory limitations exist and some of them were described by *Beyer et al.* and *Rodríguez et al.* [3, 4]. A promising candidate is lossy compression, which allows for massive reduction of file size at the cost of small errors in the dataset. The GPUs native hardware accelerated support for certain decompression schemes could accelerate the performance additionally since bilinear and even trilinear interpolation comes at very small cost compared to manual implementation [1]. ASTC represents a modern technique which contrary to its competitors is capable of describing a great variety of data types in one single format. It comprises many modern advances in texture compression and is capable of high quality decompression for not only image data but also uncorrelated data [16, 29]. However, only limited research has been done about lossy vector field compression in the domain of GPU-driven streamline integration. Therefore, this work aims to implement ASTC compression for vector fields and analyze its applicability for visualization by investigating its error susceptibility and performance in the context of streamline integration.

This study will first present a selection of related work in the context of vector field compression and visualization in Chapter 2. The theoretical basics are provided next in Chapter 3 and encompass a quick overview of vector field visualization, GPU architecture, texture compression and error metrics. The implementation specifics of the ASTC particle advector and a description of the investigated datasets are given in Chapter 4. All results are then presented in Chapter 5, subdivided into the different aspects of analysis: geometric error, visual error and performance. This study then concludes with a discussion of the results in Chapter 6 and a summary in Chapter 7. The appendix provides alternative views of the presented data, additional findings which were not subject to analysis and all SSIM highlighted images of each datasets visualization scenarios.

Chapter 2

Related Work

Fitting large scale datasets into limited memory for visualization is an active research area [3, 4]. *Rodriguez et al.* presented already an extensive overview on this topic with focus on the task of scalar field visualization and direct volume rendering [3]. But once the perspective shifts towards higher dimensions contributions become sparse. Here, recent literature covering a range of methods tailored towards faster visualization of large scale vector fields is presented.

In 2001 *Lum et al.* used temporal encoding with lossy compression to bring interactive visual exploration of time varying scalar datasets to low cost desktop PCs [21]. Their approach summarizes multiple timesteps located at a fixed spatial position into a single scalar that is quantized and in the process also compressed by Discrete Cosine Transform (DCT). DCT is a coefficient based encoding that is applied in the frequency domain and uses cosine signals to model the original signal. It can be a lossy compression if, e.g., lower energy coefficients are quantized at a coarser resolution [3]. *Lum et al.* utilized the lossy approach to further reduce storage and bandwidth requirements, enabling the interactive investigation of 4D datasets at their time [21].

In 2003 a different approach was pursued by *Theisel et al.* as they focused on a topology preserving compression method for 2D vector fields [43]. They interpreted the vector field as a piecewise triangular mesh and performed half-edge collapses and other mesh reduction operations to compress it. But they enforced the condition that the global topology of source and compressed vector field must coincide with regards to critical points and separatrices through an iterative design which tests the topological consequences of a mesh reducing operation before it is applied permanently. Their results showed that a compression rate of 95.3% could be achieved while still retaining the same topological structure [43].

In 2010 *Golembiovskỳ* approached time varying vector field compression and presented a custom encoding method [11]. He used a logarithmic transformation to quantize the 3D grid of each component separately, thereby already performing lossy compression. Additionally, the encoded data was decomposed into blocks of equal size and then further compressed in temporal dimension by encoding temporal redundancy similar to the video compression schemes. With this method they were able to reduce the file size by 80-90% while still maintaining acceptable errors at the cost of very high decompression times per frame [11].

In 2016 *Treib et al.* presented a compression technique for turbulent vector fields that is suited for particle tracing on regular desktop GPUs [45]. Their main technique was Discrete Wavelet Transform (DWT) compression followed by a run-length and Huffman-encoding, achieving an efficiency of 3 bits per floating point vector. DWT is, similar to DCT, a compact coefficient-based representation in frequency domain but also preserving the spatial domain [3]. In combination with prior bricking to subdivide the dataset into smaller chunks, which can be easily uploaded to video memory as needed, they constructed a fast out-of-core method that was able to cut down visualization processing time by 90% [45]. A drawback of their method was additional overhead imposed by the need to fully decompress a brick inside video memory and transferring it into a texture before use.

In 2020 *Liang et al.* went for an error-bounded compression technique which preferred the topological preservation of critical points in the vector field [19]. Their core idea was to compute an error-bound for each vertex and feed it to an prediction-based lossy compressor which adapts compression based on the local error-bound. This output and the vertex-wise error bounds are then further encoded by a lossless compressor. The complete process is capable of in-situ processing which however is computationally more expensive than offline compression. The results of their work showed that they could compress a 3D vector field by approximately 87% and visualization of the topological features using the compressed dataset proved that all topological critical points could be preserved [19].

Research in this subfield of visualization seems limited though recent contributions have shown that the accuracy loss of lossy compression can be acceptable and in range of the error introduced by frequently used interpolation functions [44]. Especially the use of hardware accelerated texture compression methods is absent when it comes to vector fields. This work therefore aims to investigate the possible acceleration and file size savings if a regular GPU supported compression method is applied on vector field data and the error which accompanies this approach.

Chapter 3

Theoretical Background

This chapter explains the theoretical basis of this study. The main concepts of vector field visualization are provided first in Section 3.1. Characteristics of the GPU hardware and its API are considered next in Section 3.2. Extending on the GPU basics, Section 3.3 involves a description of texture compression in general and ASTC in particular. Finally, the applied error metrics used for geometric and visual analysis are provided in Section 3.4.

3.1. Vector Field Visualization

Vector fields are a non-trivial task in visualization as they hold complex information for each point in space and time. They are described by the mapping

$$\begin{aligned} V &: A^{n+k} \to B^n \\ V(\mathbf{x}, t) &\mapsto \mathbf{v}, \end{aligned} \tag{3.1}$$

for $n \in \mathbb{N}$ denoting the spatial dimension and $k \in \{0, 1\}$ signaling the temporal dependency [10, 17]. The resultant broad range of vector fields discourages a unified visualization approach and in practice there are often different methods desired for either 2D or 3D fields.

Scalar fields can be depicted a special case of 1D vector fields. They hold only a single scalar for each point in space and time and are therefore able to describe any vector field by construction of a separate scalar field for each component. Their structural simplicity and importance in visualization made them a well understood subject with custom visualization methods [4]. Because of their special relationship the concepts of scalar field visualization are applicable on vector fields as well. Accordingly, traditional methods such as direct volume rendering through raycasting into the data or isosurface construction by prior set thresholds are possible. But they are impractical as they require adaptation which has been shown to increase computation cost significantly [9]. Alternative methods specifically designed for vector fields are generally preferred, classified into direct, dense texturebased, geometric and feature-based approaches [15]. Most of these methods exploit the semantic vicinity of vector fields to flows by velocity. The velocity of a flow is a derivative quantity represented by a vector field [23]:

$$V(\mathbf{x}, t) = \frac{d\mathbf{x}}{dt} \tag{3.2}$$

Integration of this vector field will yield a flow and therefore, for the task of visualization, vector fields and flows are often pictured equivalent [22].

Direct flow visualization is concerned with the direct representation of the vector field through color coding or simple arrow glyphs. It is computationally cheap, but occlusion becomes a great concern in 3D space. Dense texturebased approaches mark the paths particles would take in a flow by integrating the whole domain of a texture, e.g. given as white noise. They are more suited for 2D fields, since they capture a lot of detail and therefore suffer heavy occlusion in 3D which requires difficult post-processing to resolve [13, 36]. Geometric flow visualization follows a similar approach, but integrates singular particles instead of the whole flow domain and constructs geometric objects upon the derived flow characteristics. This enables fine control over the general visualization and computational expenses, allowing to reduce clutter while still providing meaningful insight. Feature-based approaches investigate the topology of the vector field and extract flow important features like vortices. However, the entire domain needs to be analyzed beforehand and therefore imposes a significant computation overhead [15, 23].

At the core of this work will be particle advection as the main application of geometric flow visualization. It allows visualization at interactive framerates and is well suited for the case of 3D vector fields in steady and unsteady condition. But before this topic is further described, the discrete nature of the data will be addressed first.

3.1.1. Grids

Experimental or simulation-driven construction of a vector field is generally a time-demanding task and consequently performed offline prior to analysis. This rises need for a suiting data structure to fit the complete field in an efficient manner. This work will focus on regular grids where sampled elements are stored as nodes in a spatially ordered layout and whose correct position in space can be reconstructed through distance-

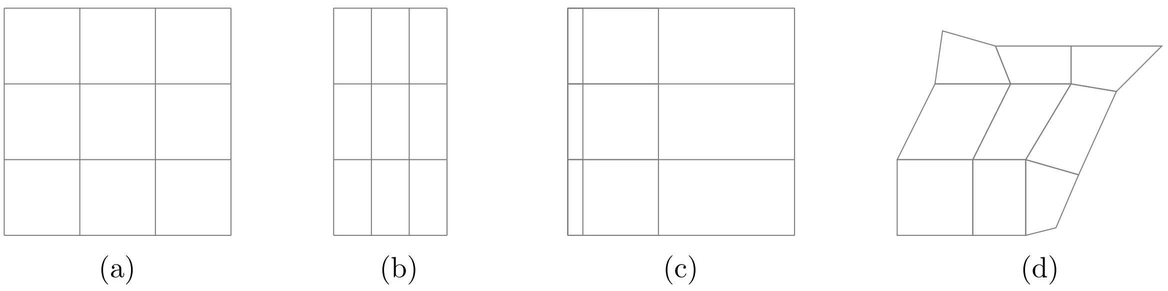


Figure 3.1.: (a) Cartesian grid, (b) regular grid, (c) rectilinear grid, (d) structured grid

weighted edges [52]. Structured grids, shown in Fig. 3.1, are common results of such discretization processes as they offer an intuitive indexing of each grid node [52]. These are composed of hexahedral cells in 3D space and maintain an implicit neighborhood connectivity which allows easy indexing of the individual grid cells. Special cases of structured grids are rectilinear, regular and Cartesian grids, which conform to rectangular cuboid cells [52]. Within scope of this work are regular and Cartesian grids, of which the former are composed of homogeneously sized cells and the latter are a special case of the regular grid with uniform sized cells.

It is likely that intermediate values need to be obtained from the sampled grid during visualization. A common way to do this is called linear interpolation [42]. Given two discrete values a and b, a linear combination of these according to some $\lambda \in [0, 1]$ can smoothly interpret the space between them [39]:

$$x = \lambda \cdot a + (1 - \lambda) \cdot b \tag{3.3}$$

Linear interpolation is performed multiple times for higher order dimensions, commonly referred to as bilinear in 2D and trilinear in 3D [1]. Other methods like Lagrangian interpolation exist and also promise a more realistic sampling of intermediate values in non-linear environments[44], but the simplicity and performance of linear interpolation is considered well enough [5].

3.1.2. Particle Advection

Particle advection is an integration based visualization method where the given vector field is interpreted as a velocity field to recreate the flow upon its properties. It follows a two-part process [23]:

1. Seeds, the initial particles, are placed inside the dataset according to a certain seeding strategy.

2. Trajectories are constructed by tracing the particles as they move through space and time, compliant to the prevailing velocity.

The first step, choosing a correct seeding strategy, is an open research topic itself. It greatly impacts the capture of important features, but also the generation of clutter. For instance, a dense seeding will capture even smallest features but they may be occluded by other trajectories, whereas a too sparse seeding would allow good visibility which however is likely to miss out interesting features [23]. Therefore, sophisticated solutions often prefer to place seeds within interesting regions instead of broadly spreading seeds over the whole domain [51]. For the scope of this work, an even seeding within a specific range is regarded the best compromise between complexity and the capture of many features. Occlusion will be a concern for evaluating the results at image level, but comparison of the geometric instances is independent of this problem and will receive a more representative set of samples.

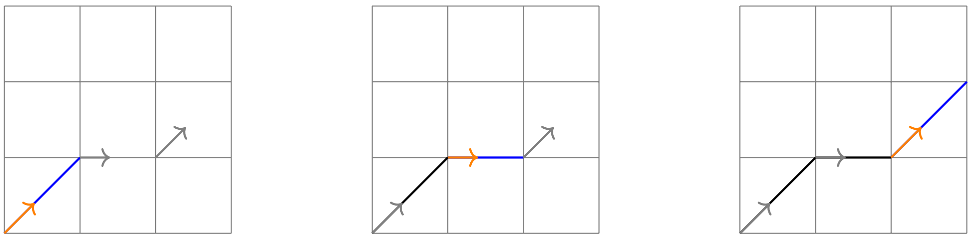


Figure 3.2.: Streamline integrated over three steps in a time-steady vector field. Euler integration scheme and unit-velocity scenario.

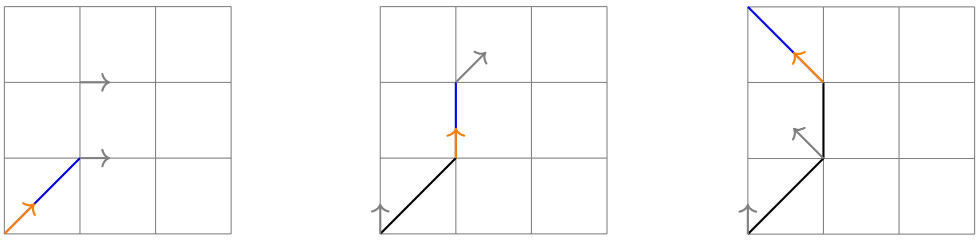


Figure 3.3.: Pathline integrated over three timesteps in a time-varying vector field. Euler integration scheme and unit-velocity scenario.

The second, computationally most demanding part is the construction of trajectories. There are three basic types of trajectories: streamlines, pathlines and streaklines [23]. For time constant steady fields all trajectories come down to streamlines which are everywhere tangent to the flow field. Figure 3.2 shows an exemplary streamline constructed from a single particle within a steady vector field. Unsteady fields vary in time and are generally described by streaklines or pathlines, since streamlines would only capture the flow at a single timestep [23]. Streaklines behave like dye, as particles are constantly ejected from the seeds initial position. Their trajectories are constructed from all particles which were spread from the respective seeding location, therefore resulting in a trace which stays affected by the flow field. Pathlines on the other hand describe the specific way a single particle travelled inside the changing field. These trajectories are constructed by the waypoints a particle traveled along and are not affected by the changing field [23]. The visualization of a pathline inside an unsteady vector field is shown in figure 3.3.

The algebraic technique to generate any trajectory is integration of the vector field by solving the ordinary differential equation (ODE) [17]

$$V(\mathbf{x}(t), t) = \frac{d\mathbf{x}}{dt} = \mathbf{v}, \tag{3.4}$$

where V represents the vector field, $\mathbf{x}(t)$ the time-dependent particle position at time t and $\frac{d\mathbf{x}}{dt}$ the tangent (velocity) at this position. The ODE is constructed and solved for each particle by setting its initial position as initial condition [17]:

$$\mathbf{x}(0) = \mathbf{x}_0 \tag{3.5}$$

However, V is most often represented by a discrete dataset $\widetilde{V}$ which requires interpolation for intermediate values and numeric integration. Therefore, the analytical solution to this problem is generally not available and the exact trajectory starting at x_0 can't be constructed [17]. Instead, the approximation $\tilde{\mathbf{x}}(t)$ is obtained by solving

$$\widetilde{V}(\tilde{\mathbf{x}}(t), t) = \frac{d\tilde{\mathbf{x}}}{dt} = \tilde{\mathbf{v}}, \tag{3.6}$$

whose behavior is strongly dependent on the chosen interpolation scheme and numeric integration method [17].

The core idea of numeric integration is to continuously add small displacements to the previous particle position based on the fields velocity. The Eulerian method implements this scheme straightforward by calculating the displacement based on the particles current positions velocity as [34]

$$d\tilde{\mathbf{x}} = dt \cdot \widetilde{V}(\tilde{\mathbf{x}}(t), t) \tag{3.7}$$

for a small dt. This method however is erroneous, as $\tilde{\mathbf{v}}$ evaluated at time t is assumed to be constant for the duration of dt. Instead, Runge Kutta 4th Order integration established as the de facto standard method, preserving good accuracy at small computational overhead [34]. It computes four intermediate velocities at three different timesteps and interpolates the result according to the following Equation: [34]

$$\begin{aligned}
k_1 &= \widetilde{V}(\tilde{\mathbf{x}}(t), t) \\
k_2 &= \widetilde{V}(\tilde{\mathbf{x}}(t) + dt \cdot \frac{k_1}{2}, t + \frac{dt}{2}) \\
k_3 &= \widetilde{V}(\tilde{\mathbf{x}}(t) + dt \cdot \frac{k_2}{2}, t + \frac{dt}{2}) \\
k_4 &= \widetilde{V}(\tilde{\mathbf{x}}(t) + dt \cdot k_3, t + dt) \\
d\tilde{\mathbf{x}} &= dt(k_1 + 2k_2 + 2k_3 + k_4)\frac{1}{6}
\end{aligned} \tag{3.8}$$

3.2. The GPU

As indicated by name, the Graphics Processing Unit (GPU) is a processor on its own traditionally designed to quickly process large amounts of geometry for presentation on a screen. Formerly, it was constructed as a closed configurable-only system limited to this type of tasks. However, during the last two decades, its interface evolved and permitted access to a wide range of programmers intents [1, 31]. This led to the emerge of General Purpose Computations on GPU (GPGPU) which refers to applications that moved their computational expenses of non-graphical tasks from the CPU to the GPU [26, 31]. The motivation to shift computations on the GPU lies within its orthogonal architecture design and execution model as compared to CPUs which allows for a significant acceleration of tasks if certain conditions are met [31].

This section aims to introduce the relevant aspects of the GPUs processing design and core mechanics which leverage its compute power. First, the physical architecture is described to show both the origin and pitfalls of the GPUs compute power. Then, the compute flow is addressed which explains how the GPU can be accessed and instructed for processing. Major contributions to this topic were gathered by *Ankenine-Moller et al.* and *Owens et al.* [1, 31], which can be compared with current specifications of the GPU manufacturers *NVIDIA* and *AMD* [14, 28].

3.2.1. GPU Architecture

Today's GPUs are driven by the power of massive parallelization, aimed for maximum throughput. For this purpose, they primarily employ a large amount of specialized processors that are designed for operating on a global task in concurrency.[1] These processors are called shader cores[1] and can amount up to multiple thousands on current generation GPUs [14, 28]. They are optimized to be synchronized in execution and to perform the same instructions, but each on their individually resident data, which is denoted Single Instruction, Multiple Dispatch (SIMD) function [1]. Accordingly, the respective task needs to be implemented core-invariant and core-independent to use the full potential of the GPUs compute power. Additionally, specialized chips are soldered to the GPU designed to perform specific tasks in very fast fashion [1]. Since these cannot be programmed, they are called the fixed-function hardware of the GPU. Within this work decoder chips and special texturing hardware become important which allow on-the-fly decompression of compressed data and accelerated interpolation of intermediate data values [1]. This allows GPUs to outpace certain parallelizable data tasks of CPUs by a large factor [31].

However, as pointed out by *Beyer et al.* and *Rodriguez et al.*, many contributions to scientific visualization discovered bandwidth to become the main limiting factor of

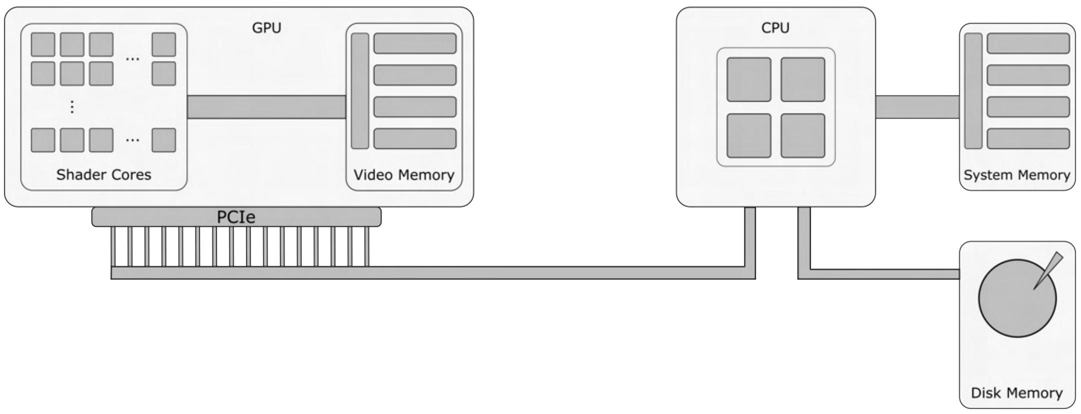


Figure 3.4.: Traditional model of a CPU/GPU system. Here, the GPU is accessing system memory and disk memory indirectly by the CPU, but new commercially available hardware is capable to allow the GPU direct access [14, 28].

GPU-driven data visualization [3, 4]. The reason lies within the GPUs limited amount of fast accessible video memory, whose bandwidth can reach a communication speed of 448 GB/s and even more [28]. If the video memory is exhausted the shader cores need to fetch data from the CPU managed system memory which can be slower by a factor of 14 through a reduction in communication throughput down to 31.51 GB/s in case of the currently standard PCIe Gen4 interface [1]. As a consequence, processing on the GPU is stalled until the data has been fetched into local memory. Fig. 3.4 shows an exemplary model of this CPU-GPU communication relation. Several approaches to mitigate the effect of latency introduced by communication with the system memory exist and are being examined by an active research area. But all present approaches require to decide whether to value information over latency or vice versa [3, 4]. Three basic techniques, ranging from one trend to the other, are the following:

Out-of-core algorithms are designed to process the whole dataset by chunks that fit into certain boundaries like the available video memory. They accept the cost of communication and try to hide the latency through smart optimizations in the code and, e.g., ignore currently irrelevant data regions or try to predict them [4].

Region of interest (RoI) selection is another viable approach and similar to out-of-core methods. Instead of processing the whole dataset, only preselected regions of the dataset are examined, reducing the information content of the visualization. These regions may be extracted by prior offline analysis or meta knowledge about the dataset to prevent missing informative features [18].

Compression can be performed lossless or lossy, the latter meaning a permanent reduction of information along with the filesize. Lossless methods are limited in efficiency through the concept of entropy itself [37] and may not offer enough reduction in size for a significant acceleration. Lossy methods on the other hand accept a configurable loss of information in form of reduced accuracy, thereby achieving much bigger savings [35, 45].

Out-of-core methods and RoI selection try to conceal the bandwidth limitations of the GPU, but they alone fail at effectively enabling investigation of the global dataset at near-interactive timings. A combined approach including compression is penalized by a loss in precision, but it includes multiple advantages. Next to a smaller communication throughput, the overhead of decompression is near zero since fixed-function hardware of the GPU is capable to decompress only local regions of the data, allowing for random-access traversal without prior decompression of the full dataset [29]. Additionally, linear interpolation comes at near zero cost [38].

The GPUs hardware is therefore well suited for the task of particle advection, but measures to overcome the bandwidth bottleneck need to be taken in order to perform interactive analyses. Here, the focus will be on lossy compression, which is described in greater detail later in Section 3.3. The next question to be answered is how the GPU is instructed for operation and what is special about its compute flow.

3.2.2. Rendering Pipeline

GPUs are traditionally exposed to the programmer by a limited set of graphics APIs: OpenGL, DirectX and Vulkan to name a few [1]. They own a logical and a physical model, of which the first is accessed by the programmer and the latter implemented by the GPU manufacturer, offering a unified GPU interface in a heterogenous hardware environment [1].

The canonical execution model of the GPU is called the Rendering Pipeline and is shown in Fig. 3.5 [1]. It is primarily meant to transform 3D scene geometry comprised of geometric primitives (i.e. points, lines, triangles) into a discrete 2D representation. These primitives are given to the GPU as vertices that describe positions in 3D space and later are transformed into fragments which each resemble the potential content of a single pixel. The execution flow of the traditional rendering pipeline has been highly optimized by GPU manufacturers and contains the following tasks [1, 31]:

Vertex Processing: Vertices describing the primitives are processed according to a shader program to transform or add vertex-local information.

Primitive Assembly: Vertices are assembled into geometric primitives.

Rasterization: Primitives are transformed into fragments, a discrete pixel representation.

Pixel Processing: Each generated fragment is processed individually as instructed by a shader program.

Composition: The final fragments are composed into an image.

A more subdivided description is given by Figure 3.5. It emphasizes the unidirectional workflow of the rendering pipeline where each stage feeds its output as input to the next stage, thereby dictating the computational domain on which threads are invoked for processing.

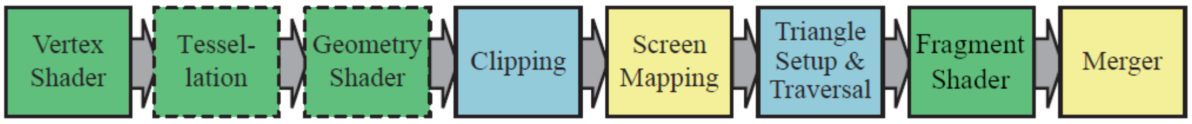


Figure 3.5.: Model of the Rendering Pipeline. Green stages are programmable shader stages, the remaining boxes are fixed function. Yellow stages allow configuration. Courtesy to [1].

The vertex shader therefore initiates threads based on the complete set of vertices resembling the input geometry. But the input gets thinned out, i.e. during clipping, omitting vertices outside the cameras view, and is finally transformed into fragments during rasterization. The fragment shader resembles the last programmable stage and is executed upon a resolution and geometry dependent number of fragments which survived the prior stages [1].

Though by design it is optimized for rendering graphics, the real behavior is up to the programmer's implementation of the shader programs and how they process their input and output.

3.2.3. Compute Shader

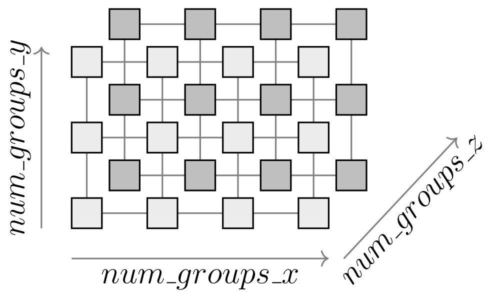


Figure 3.6.: To invocate the Compute Shader a domain must be defined in 3D space. Each node represents a threadblock, which itself holds a scaleable amount of threads.

On modern devices exists a second way to access the GPUs compute power, primarily meant for GPGPU aside traditional rendering tasks. It is an alternative execution model which only consists of a single shader program called Compute Shader within OpenGL [1]. As explained by *Owens et al.* it is primarily a more natural and direct interface for the programmer to access the GPU for general purpose computations [31]. As it does not adhere to a multi-stage model, contrary to the Rendering Pipeline, it has no output interface such that the task of data management is completely up to the programmer. Thread invocation is abstract as well and not dependent on vertices anymore, but replaced by a spatial computation domain that must be defined by the programmer before execution [26, 31]. A visualization of the Compute Shaders invocation domain is shown in Figure 3.6.

The Compute Shader execution model offers the same programmability as the rendering pipeline. However, the interface exhibits easier access to the GPUs compute power for non-rendering tasks [31]. A split approach is therefore suggested by *Owens et al.* [31], which can perform shape generation from raw data on the Compute Shader and shape visualization as a natural rendering approach by the Rendering Pipeline.

3.3. Texture Compression

Sellers et al. and *Shreiner et al.* define textures as shader accessible storage types that are designed to hold image type data [38, 39]. Their elements are arranged accordingly on a grid structure ranging from 1D to 3D [1] where each node holds one to four values, depending to the assumed color model and existence of an alpha channel. They resemble a specialized type of data buffer in video memory which was optimized for fast access timings by the GPU manufacturers due to the importance of image data in the context of rendering [1]. As a consequence they own two unique features compared to regular data buffers: support for hardware-accelerated linear interpolation and decompression.[38]. The presence of hardware support for linear interpolation allows to directly look up intermediate data values at nearly zero cost in all dimensions [38]. Therefore, this design makes them a preferable choice for volumetric visualization and is also applied by *Nagayasu et al.* [24].

The second feature is hardware support for compression, as it is a common technique to reduce image type data in size [37]. A wide variety of compression techniques to encode images into alternative descriptions exists [37], however only few of them profit of GPU acceleration. Those who profit are generally based on the lossy S3TC scheme: Independent blockwise encoding of texels, independent and fast decoding of blocks, fixed size compression [1]. The texels within a block shaped region are encoded by interpolant weights $\lambda_i \in [0, 1]$ which calculate a weighted average of 2 reference colors c_1 and c_2. Each block holds some control information and has a fixed size memory

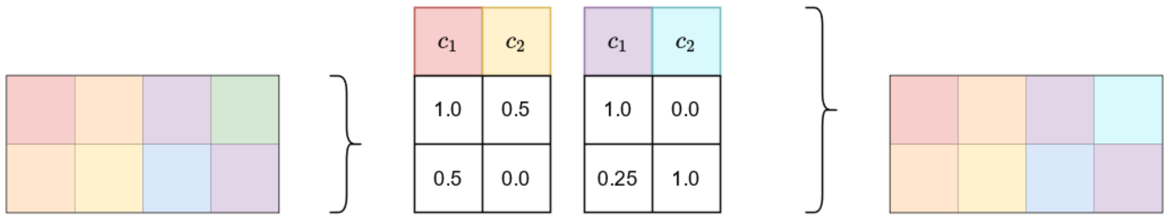


Figure 3.7.: Example compression and decompression of an image using the S3TC scheme with 2x2 block size. L.t.r.: Original image, compressed representation, decompressed image with erroneous texel. c_i denotes a reference color for each block.

footprint irrespective of its content [1]. This allows to decompress blocks individually and infer the exact block that is holding a certain texel. Figure 3.7 shows an exemplary compression and decompression of an image using the S3TC scheme for a block size of 2x2. In general, GPU compression techniques encode texels only to 2D blocks of size 4x4 and restrict the interpolation factor precision to 2 or 3 bits.[1] 3D compression can be imitated by a sliced 3D approach where 2D compressed slices are stacked upon each other [27]. However, a limitation inherent to block-based compression additionally to the inevitable precision loss is the generation of artifacts. In image data, these are visible as discontinuities between adjacent blocks, especially at low bit rates [3]. Well known S3TC-based techniques are e.g. BC7 for low dynamic range (LDR) colors with 8bit precision and BC6H for high dynamic range (HDR) profiles and 16bit precision [1].

3.3.1. ASTC

ASTC is an elaborated lossy compression technique which was published 2012 by *Nystad et al.* [29] and is precisely specified for OpenGL [16]. At core ASTC is an S3TC based technique with a block memory footprint of 128bit. Besides that, it offers great versatility by multiple block sizes, native 3D compression and LDR as well as HDR profile support within a single codec [29]. The supported block resolutions range from 4x4 to 12x12 texels in 2D and 3x3x3 to 6x6x6 texels in 3D [29]. Most information is saved on a per block basis and can be locally optimized by the encoder, the only global state variables are the block resolution and color profile. The block itself is a highly optimized construct which employs an encoding that allows the use of fractional bits and therefore more compact representation. It can hold up to four partitions which each describe a pair of color endpoints, assignable on a per-texel basis [29]. Additionally, a second weight can be saved per texel to better represent data which has uncorrelated values across their color channels [16].

Comparative analysis has shown that ASTC compression was slightly behind BC6H and BC7 regarding LDR and respectively HDR image quality, but outperformed any other technique [8, 29]. The application of ASTC compression on non-image data however was not subject to thorough investigation yet though it offers a rich potential by its adaptive design.

3.4. Error Analysis

A performance metric is inherently given by the time a certain operation needs. But in order to analyze the error that emerges from compression a wide range of metrics for various purposes exist. *Verma et al.* described the key aspects of comparative flow analysis and subdivided it into three essential levels of comparison: data, feature and image level [48]. The data level refers to any meaningful metric on the raw data, feature level operations investigate the visualized entities and image level examination compares two pictures with identical camera parameters [23, 50]. Here, the main emphasis lies on feature and image level analysis and will be explained in the following Sections.

3.4.1. Geometric Evaluation

Subject to comparative analysis on a feature level are the generated trajectories of the advection process. A commonly used accuracy metric is to calculate the Euclidean distance between the approximate object and the ground truth.[48] For points p and q in n-dimensional space Euclidean distance is defined as

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{3.9}$$

and hence penalizes any deviation [41]. Since the trajectories are complex objects that are spatially defined by their comprising vertices, their distance is interpreted as the average vertex distance. Here, the actual error is then obtained as the Root Mean Square Error (RMSE) of the vertex distances. Given observations o_i resembling the measured vertex distance, predictions s_i set to the optimal distance of zero for $i \in [1, n]$, the RMSE is defined as [25]:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(s_i - o_i)^2} \tag{3.10}$$

$$= \sqrt{\frac{1}{n}\sum_{i=1}^{n} o_i^2} \tag{3.11}$$

The accuracy of the complete dataset is calculated by the weighted average of each trajectories RMSE measure where the weights account for entities with varying amount of vertices. Additionally, a global maximum RMSE metric is calculated for each dataset by the average of each trajectory's maximum RMSE measure.

3.4.2. Visual Evaluation

Images used for visual analysis resemble an information reduced 2D representation which more directly reflect the perceived amount of information. Visual methods are solely pixel-based and side-by-side comparison depicts the most basic method to obtain an error value if no supplementary data is incorporated [48]. Two established visual error estimates are the peak signal-to-noise ratio (PSNR) and structural similarity index measure (SSIM) [12]. The PSNR is noteworthy as it is a steadily reoccurring metric throughout literature. However, it is an unbound measure which approaches infinity for vanishing differences and is calculated for the whole image at once. The SSIM on the other hand was designed as an improvement over PSNR for better correlation with the human visual system (HVS) [12]. It operates on a normalized scale and is calculated locally and averaged globally into a summarizing quality score. For these reasons the SSIM resembles the core metric of visual analysis within this work.

The SSIM is calculated by combining the results of luminance l, contrast c and structure comparison s using the following formula [49]:

$$\mathit{SSIM}(g,h) = l(g,h)^{\alpha} \cdot c(g,h)^{\beta} \cdot s(g,h)^{\gamma} \tag{3.12}$$

where g and h are two image signals and $\alpha, \beta, \gamma > 0$. The measure is taken locally using a sliding window approach for which the original authors suggested a size of 11x11 pixels. The complete derivation of the SSIMs subformulas is given by *Wang et al.* [49].

To derive the PSNR score for an image, it is aggregated into the global Mean Square Error (MSE) of pixel differences first and then processed by the following equation [12]:

$$\mathit{PSNR}(g,h) = 10 \cdot \log_{10} \cdot \left(\frac{255^2}{\mathit{MSE}(g,h)} \right) \tag{3.13}$$

The MSE is calculated analogous to the RMSE without applying the square-root and the numerator of 255 depicts the maximum possible pixel value for a regular image with 8bit channels. A more in depth presentation of the PSNR and its comparison to SSIM has been conducted by *Horé and Ziou* [12].

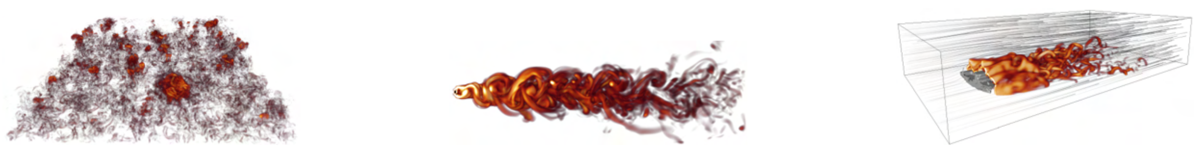
Chapter 4

Research Approach

Here, the implementation of the visualization pipeline, error analysis and the investigated datasets are presented. Section 4.1 provides an overview of the analyzed data, followed by the implementation in Section 4.2. The implementation is further subdivided into three stages of processing which cover the data pre-processing stage, the visualization process and post-processing.

4.1. Datasets

To test for a broader set of use-cases, three datasets were obtained from *ETH Zürich* to test ASTC compression in different environments [7]. Accordingly, the representative vector fields show each different characteristics in terms of flow, topological features and application. Figure 4.1 shows exemplary visualizations of the data. All vector fields are set in 3D spatial space with 32bit floating point precision per scalar. They all are saved in the netCDF file format and designed as contiguous blocks for each component. A short description of each dataset is given hereafter.



(a) Clouds (steady)

(b) Half Cylinder (unsteady)

(c) R/V Tangaroa (unsteady)

Figure 4.1.: Example visualizations of the datasets used in this thesis [7].

4.1.1. Research Vessel Tangaroa

Popinet et al. investigated the airflow distortion caused by turbulence around the research vessel *Tangaroa*. They obtained experimental and simulation data for this cause, of which the latter has been saved onto an unstructured grid holding relative wind speeds as measured from the ships position [33]. The acquired dataset is reduced to a region of interest of the original and has been resampled onto a regular grid using *Gerris Flow Solver* [32].

Filesize	15.629 GB
Grid Resolution (X x Y x Z x T)	300 x 180 x 120 x 201
Simulation Domain	[-0.35, 0.65] x [-0.3, 0.3] x [-0.5, -0.3] x [0, 2]
Spatial University	Dimensionless
Temporal Unit	Dimensionless
Element Unit	Dimensionless

4.1.2. Half Cylinder Ensemble

Rojo et al. were concerned with visualization of unsteady vector fields which aimed to provide a complete picture of the topology for every time slice. They constructed a fluid simulation where the obstacle is a half cylinder for several Reynolds numbers on an unstructured grid [2]. The investigated dataset holds the resampled version of the source data for the Reynolds number 6400 on a regular grid.[32]

Filesize	22.265 GB
Grid Resolution (X x Y x Z x T)	640 x 240 x 80 x 151
Simulation Domain	[-0.5, 7.5] x [-1.5, 1.5] x [-0.5, 0.5] x [0, 2]
Spatial Unit	Dimensionless
Temporal Unit	Dimensionless
Element Unit	Dimensionless

4.1.3. Cloud-Topped Boundary Layer

Stevens generated a cloud resolving boundary layer simulation with the UCLA-LES model, holding some rising cumulus clouds lying on a regular grid [40].

Filesize	230.044 MB
Grid Resolution (X x Y x Z)	384 x 384 x 130
Simulation Domain	[0, 10] x [0, 10] x [0, 3.2]
Spatial Unit	kilometers
Element Unit	meters per second

4.2. Implementation

To investigate the applicability of ASTC compression in the context of particle advection I developed a software tool that allows to load binary or HDF5 formatted data and as a first step to pre-process it for further use with the *arm ASTC Encoder*[20] and rendering pipeline in general. Secondly, the prepared data is visualized in real-time through an integration process whose output is forwarded to the rendering pipeline. A C++ based approach using OpenGL to communicate with the GPU has been chosen as the underlying architecture to obtain reasonable performance and exact timings of the individual sub-processes. Two PC systems were used for investigation: A powerful workstation and a regular laptop, both followingly denoted as such. This selection was done because of hardware compatibility restraints imposed by the ASTC compression. The workstation works on two Intel Xeon CPU E5-2680 v3 processors, 128GB DDR4 system memory and an 512GB system SSD in combination with a 7TB data HDD. The laptop is a Lenovo T450s powered by an Intel i5-5200U processor with an Intel HD Graphics 520 integrated GPU and 512MB shared video memory, 8GB DDR3 system memory and a 500GB SSD.

The pipeline used in this work is constructed as two big conceptual steps:

Pre-process data

1. Convert source data into RGBA texture format
2. Optionally normalize & compress data texture

Visualize data

1. Configure integration properties
2. Advect particles
3. Render result until new advection

Pre-processing happens on the more powerful workstation while the complete visualization stage is performed on the home laptop. Evaluation happens both as an accompanying runtime process in case of performance measures and as an offline process to investigate the geometric and visual error.

4.2.1. Pre-Processing

The visualization pipeline expects vector data as either tightly packed RGB(A) image data or ASTC compressed textures. Therefore, the pre-processing stage comprises the operations of data reordering and optionally ASTC compression. Thereby, the preparation of source data for direct advection is straightforward and only needs data reordering if it does not already come in a tightly packed vector format. If the data will be compressed as ASTC, the reordering will also encompass downsampling into 16 bit half float and a normalization procedure. Normalization is necessary due to the LDR color profile clamping the values to the range $[0, 1]$ during compression. Though the HDR color profile allows to exceed this boundary it is not supported by the available platforms. The compression preceding normalization is a range based method using minimum and maximum values of a certain region as defined below:

$$x' = \frac{x - min}{max - min} \tag{4.1}$$

with x' denoting the normalized value and x the original one. Accordingly, the denormalization process during advection will need to reuse these minimum and maximum value pairs.

The region heuristic within which to localize these minima and maxima was set to a per depth layer basis. Two different approaches have been implemented to allow for an initial estimation of the effect of normalization on the results:

Scalar Normalization (SN) that localizes a scalar maximum and minimum value pair across all vector components and normalizes the original values using these scalars.

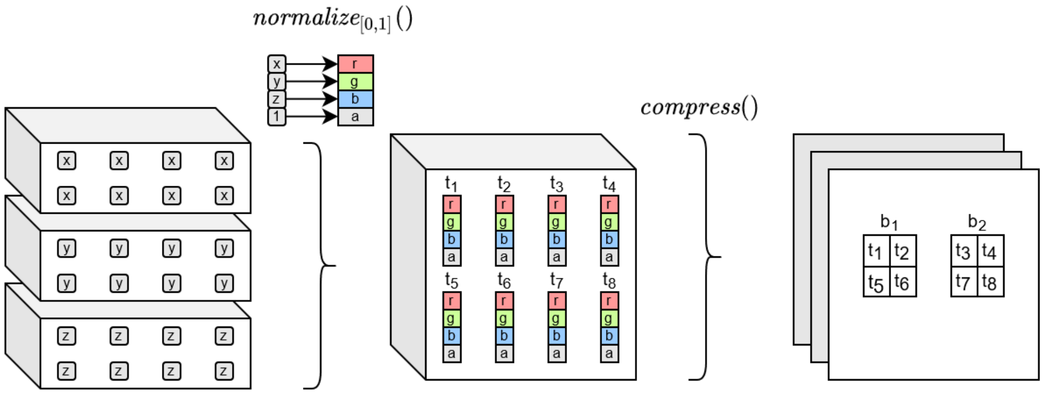


Figure 4.2.: L.t.r.: Conversion from source data to an ASTC encoded representation. t_i denotes texel i, b_k an ASTC block.

> *Vector Normalization (VN)* which normalizes using a maximum and minimum vector comprised of each components extrema.

Compression is then invoked via the ASTC encoder on each depth layer in parallel. This means the compressor will encode all 2D depth layers separately and output a sliced 3D ASTC encoded representation of the input. To ensure staying within the maximum texture size, time varying data is compressed to multiple files where each one refers to a 3D chunk describing a single timestep. While the ASTC encoder is also able of native 3D compression, the available systems are not capable of native 3D decompression and therefore the approach focuses on 2D compression. The whole preprocessing stage can be seen in Figure 4.2.

4.2.2. Visualization

Goal of the visualization stage is to perform advection upon the pre-processed data textures and some configuration. Finally, it will render the resulting trajectories until a new advection process is invocated. While the render loop resembles a simple vertex forward operation to the traditional rendering pipeline, presented in Section 3.2.2, the advection process runs through a more thorough processing.

The particle tracer was implemented in iterative fashion on the GPU using at core a Compute Shader within OpenGL. The shader program performs multiple steps: It reconstructs vectors from the data textures, postprocesses the vectors as needed (e.g. denormalization), performs the integration steps and finally saves the current particles position and velocity. The iterative nature of the approach originates from the need to interpret the vector field between two discrete timesteps. This problem was solved by

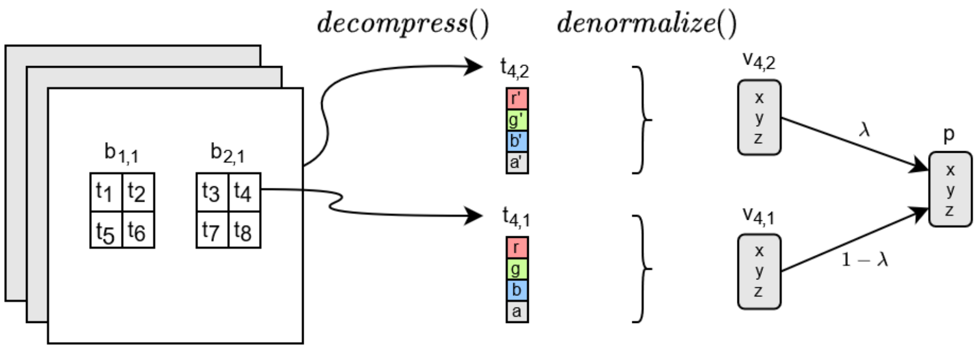


Figure 4.3.: Vector look-up & interpolation during integration of ASTC compressed data. $t_{i,j}$ denotes texel i at depth j, $b_{i,j}$ an ASTC block, $v_{i,j}$ the reconstructed vector, p the interpolated result at a certain position.

linear interpolating the vectors of two timesteps s_i and s_{i+1} according to current time t_{curr} relative to s_i and s_{i+1}. However, the GPU imposes a limit on the maximum number of active textures additionally to the limited video memory. Therefore, only the lower bound of two data textures holding adjacent timesteps are uploaded to the GPU at a time. Consequently the Compute Shader needs to be executed once for each passed timestep pair. If the integration is applied over the complete time domain of n discrete timesteps this will lead to a maximum of $n-1$ invocations.

But prior to time interpolation, vectors are constructed in space for timesteps s_i and s_{i+1} by reading the respective data textures based on the particles current position and interpolating them each in their local 3D space. Based on the underlying data structure the GPU already interpolates intermediate texels very fast using its built-in hardware support. This enables full trilinear interpolation of the source data that can be uploaded as a 3D texture. ASTC compressed data however relies on the 2D texture array representation due to the sliced 3D approach which only allows for GPU accelerated bilinear interpolation in the plane. Depth interpolation is therefore implemented manually, increasing the number of operations needed to construct a vector at the given position. The process of vector reconstruction from an ASTC compressed texture is also shown in Figure 4.3

Here, the ASTC compressed textures also need to be denormalized due to the LDR color profile. A buffer holds the denormalization parameters in video memory and is read 2 or 6 times per vector, based on the normalization approach. Source data textures on the other hand are processed directly. The total count of read operations for any setting applied in this work can be seen in Table 4.1.

Table 4.1.: Texture and buffer read operations needed during advection to retrieve a trilinearly interpolated vector from a single texture. Time interpolated vectors effectively need double the amount of read operations.

Texture	Normalized	Texture Reads	Buffer Reads	Total Reads
Source	-	1	0	1
ASTC compressed	SN	2	4	6
	VN	2	12	14

4.2.3. Evaluation

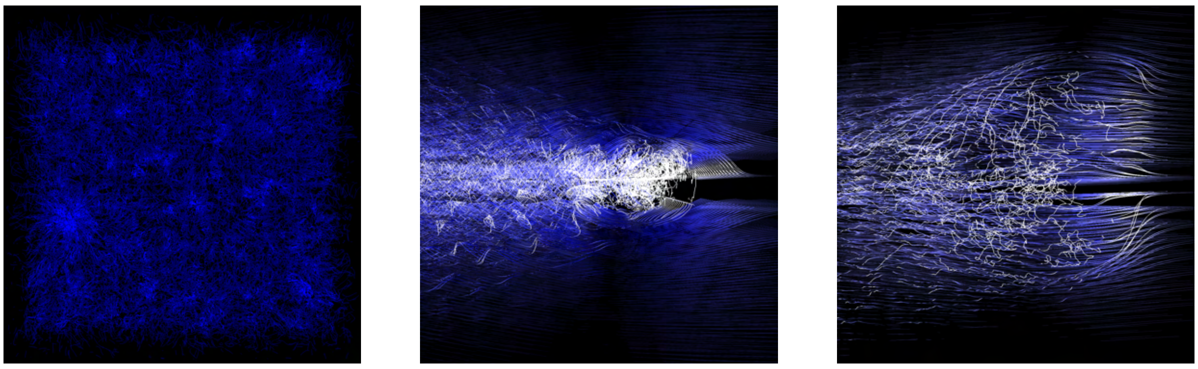
Finally, the data generated during the previous steps was evaluated according to the measures described in Section 3.4. Performance ratings were obtained during runtime at several intervals. To avoid artificial stalling, GPU timings were fetched in asynchronous fashion. Geometric interpretation was implemented as an offline approach on the saved vertex buffer contents generated during advection. Visual analysis happened on image data at different perspectives on both source and compressed data. The SSIM was calculated as a measure of visual distortion using a sliding non-overlapping window with square size 11 as proposed in the original paper. Exponent parameters α, β, γ of the SSIM formula were set to one. The individual SSIM values were used for visualizing bad regions in the processed images and averaged to obtain a global SSIM score for the respective image.

CHAPTER 5

RESULTS

In this chapter the results of the runtime and error measurements will be presented. The examined datasets, underlying system and compression configurations were prior explained in Sections 4.1 to 4.2.2. Followingly, the geometric error is investigated in Section 5.1 first and will use distance based error metrics to ratify the error imposed by the ASTC compression with regards to the chosen block size, preset and compression time. Secondly, in Section 5.2 the visual error is described by methods of the field of image comparison. In Section 5.3 the performance of the compression and integration processes is presented.

Figure 5.1 shows the results of the integration process conducted on the source datasets. The presented images are subsets of the originals focusing on the high dynamic regions which were most sensitive to the ASTC compression.



(a) Clouds (b) Half Cylinder (c) Tangaroa

Figure 5.1.: Rendered integration results of the source datasets. Displayed are subimages showing the highly dynamic regions.

5.1. Geometric Error

The geometric interpretation of the error was measured by aggregating the unique vertex pairs between source and compressed trajectory into Euclidean distances. These were collected as the average RMS and maximum RMS over all trajectories. Since the type of normalization prior to compression proofed to have considerate impact on the results, all measurements were taken for the scalar normalized and vector normalized cases, abbreviated as *SN* and *VN*. Details of the applied range based normalizations were described in Section 4.2.1. In short, SN refers to the case where each vector is normalized with the same values across each component and VN dictates the scenario in that each vector is normalized with distinct values for each component.

In case of the trajectory error, a certain set of vertices called “boundary points” were excluded from the results. A vertex was defined as a boundary point if it was integrated beyond the vector fields boundaries, which therefore became ill-defined and was prematurely stopped from further integration. Since comparing a slightly displaced, eventually prematurely stopped trajectory with a continued one does not give any meaningful insight on the loss of accuracy but only introduces a growing penalty, the comparison of these vertices has been limited to the well-defined instances only. Both source and compressed trajectories were tested for boundary points and Table 5.1 shows how many vertices were skipped and how many trajectories were considered prematurely terminated.

Table 5.1.: Overview of the number of vertices skipped and trajectories counted as early terminated if both source and compressed trajectories were tested for ill-defined vertices beyond the vector fields boundaries. Measurements are given relative to the total vertex and trajectory count. For compressed trajectories the exhaustive preset at 4x4 block size using vector normalization was used as reference.

Dataset	Skipped	#Total Vertices	Early	#Total Trajectories
Clouds	5.9%	33,750,000	9.2%	13,500
Halfcylinder	7.2%	54,000,000	7.6%	18,000
Tangaroa	20.0%	13,500,000	35.5%	6,750

5.1.1. Block Size Impact

Block sizes have been evaluated at different resolutions for each dataset, ranging from 4x4 to a maximum of 12x12, as depicted in Table 5.2. Measurements were taken for the average and maximum RMS distance error using the Exhaustive compression preset. The results are in grid space, where a distance magnitude of 1 corresponds to the distance between two cells, allowing for limited comparability.

Table 5.2.: RMS of the average and maximum trajectory error for each dataset and normalization method using different block size configurations. Values were obtained using the Exhaustive preset for compression.

Dataset	Method	Error	4x4	5x5	6x6	12x12
Clouds	SN	AVG	6.739	7.044	7.268	8.824
		MAX	12.280	12.876	13.110	15.562
	VN	AVG	5.410	5.902	6.355	8.175
		MAX	10.587	11.419	12.054	14.687
Halfcylinder	SN	AVG	0.207	0.218	0.274	
		MAX	0.408	0.432	0.562	
	VN	AVG	0.203	0.224	0.257	
		MAX	0.406	0.460	0.535	
Tangaroa	SN	AVG	0.330	0.327	0.351	
		MAX	0.605	0.611	0.647	
	VN	AVG	0.330			
		MAX	0.615			

For all three datasets the average error ranges from 0.203 to 8.824 and the maximum error resides between 0.406 and 15.562. The most prominent observation was the difference of the Clouds dataset to the unsteady datasets. The measured error of the former is consistently greater as opposed to the unsteady datasets, ranging from 5.410 to 15.562 for both error scores. Halfcylinder and Tangaroa on the other hand exhibit error scores between 0.203 and 0.647 at most, therefore being consistently lower by at least one magnitude. Choosing either normalization method had a measurable impact on the error for all datasets with varying effect. The error distribution from the Clouds dataset was most strongly affected by the applied method and choosing VN over SN resulted in consistently lower error ratings, able to decrease the error by up to 19.8%. However, the effect diminishes with increasing block size. The unsteady datasets error scores reacted less severe to the applied normalization method, achieving a maximum error decrease of 2.7%. In case of the Halfcylinder 5x5 and Tangaroa 4x4 scenario the error even grew by up to 6.4% when applying VN as opposed to SN before compression. A more consistent effect was observed when comparing the effects of increasing block size resolution, meaning a decrease in bits spent per vector (bpv). Except for the Tangaroa SN 5x5 scenario, lowering the bpv also lowered the error. The intensity of precision loss varied for each dataset, but in all scenarios the loss was smaller than the relative decrease in bpv. I.e., increasing block size from 4x4 to 5x5 lowers the bpv by 20%, but only introduces an additional error of 10.3% at most. In case of the formerly mentioned Tangaroa scenario, the average error decreased by 0.1% upon increasing block size to 5x5.

Table 5.3.: Filesize of each dataset in source and ASTC representation using different block sizes.

Dataset	Source	4x4	5x5	6x6	12x12
Clouds	219 MB	18.2 MB	11.7 MB	8.12 MB	2.03 MB
Halfcylinder	20.7 GB	1.72 GB	1.1GB	0.79 GB	
Tangaroa	14.5 GB	1.21 GB	0.79 GB	0.55 GB	

The storage savings enabled by the application of ASTC compression is depicted in Table 5.3. They are only influenced by the block resolution since each block has a fixed memory footprint. Consequently, the compression ratios are equal across the datasets and therefore it is safe to express a general statement on efficiency. The highest bitrate blocksize 4x4 achieves already a compression ratio of 8.3% which alone easily fits the Halfcylinder dataset into a regular GPUs video memory. Following square blocksizes reduce it further to 5.3%, 3.7% and 0.9% at coarsest resolution.

5.1.2. Preset Impact

Staying with the most precise block size setting of 4x4, different compression presets ranging from Fast to Exhaustive have been evaluated. The steady Clouds dataset was tested for all available presets, whereas the unsteady datasets were limited to the two most optimizing presets Exhaustive and Thorough, as denoted in Table 5.4. The evaluated error scores are again the average and maximum RMS distance in grid space, where a distance magnitude of 1 corresponds to the distance between two cells for all datasets.

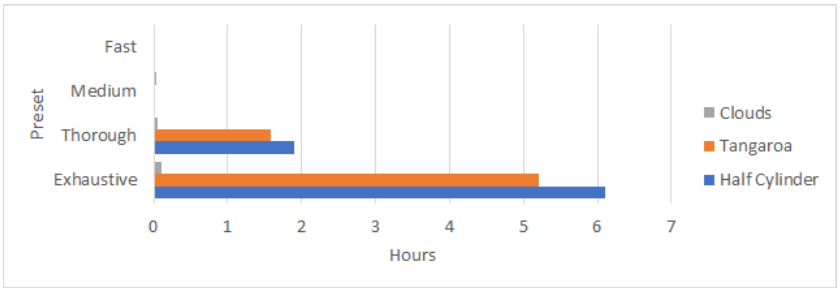


Figure 5.2.: Timings of each datasets compression process in hours. Values were obtained using the 4x4 block size and VN method. Measurements were taken on the high performance workstation.

The results in Table 5.4 extend the observations from prior Section 5.1.1 that have been described at block size 4x4. The observed range of error scores across the investigated presets is lower compared to changes in block size resolution. It resides within 5.41 to 13.164 for the Clouds dataset and 0.203 up to 0.615 for the unsteady datasets. A lower preset generally lowered the accuracy, except for one mentionable scenario. The Fast preset resembles the least optimized scenario and also generally introduced the greatest amount of error compared to the Exhaustive preset when investigated for the Clouds VN scenario. However, in case of the overall worse performing Clouds SN scenario, the measured error decreased at the Fast preset. Here, the error was lower by up to 7.9% as compared to the Exhaustive preset and therefore coined the highest precision across all presets tested for this scenario. Similar to prior block size investigation, the Tangaroa dataset performed better by 0.5% when using a less optimized preset.

Table 5.4.: RMS of the average and maximum trajectory error for each dataset and normalization method using different compression presets. Values were obtained using the 4x4 block size configuration for compression.

Dataset	Method	Error	Exhaustive	Thorough	Medium	Fast
Clouds	SN	AVG	6.739	6.912	7.273	6.209
		MAX	12.280	12.581	13.164	11.560
	VN	AVG	5.410	5.565	5.754	5.907
		MAX	10.587	10.835	11.118	11.450
Halfcylinder	SN	AVG	0.207	0.211		
		MAX	0.408	0.429		
	VN	AVG	0.203	0.209		
		MAX	0.406	0.415		
Tangaroa	SN	AVG	0.330	0.329		
		MAX	0.605	0.611		
	VN	AVG	0.330	0.330		
		MAX	0.615	0.612		

The spatial distribution of the error inferred from the VN method can be seen in Figure 5.3. It shows the individual average trajectory error projected to the XY plane for each dataset and preset, using a fixed block size of 4x4. The visualization happened in seed space, which refers to an evenly spaced regular grid where each node represents the trajectory's initial seeds position. The projection shows, that the consistently dynamic Clouds dataset has an evenly high error score within the whole integration domain. The Half Cylinder and Tangaroa datasets on the other hand show comparably diminishing small errors except for few regions. These regions are close to the obstacle within each simulation where the flow is most chaotic and exhibits peaks in the error score of the same magnitude as the average error of the Clouds dataset.

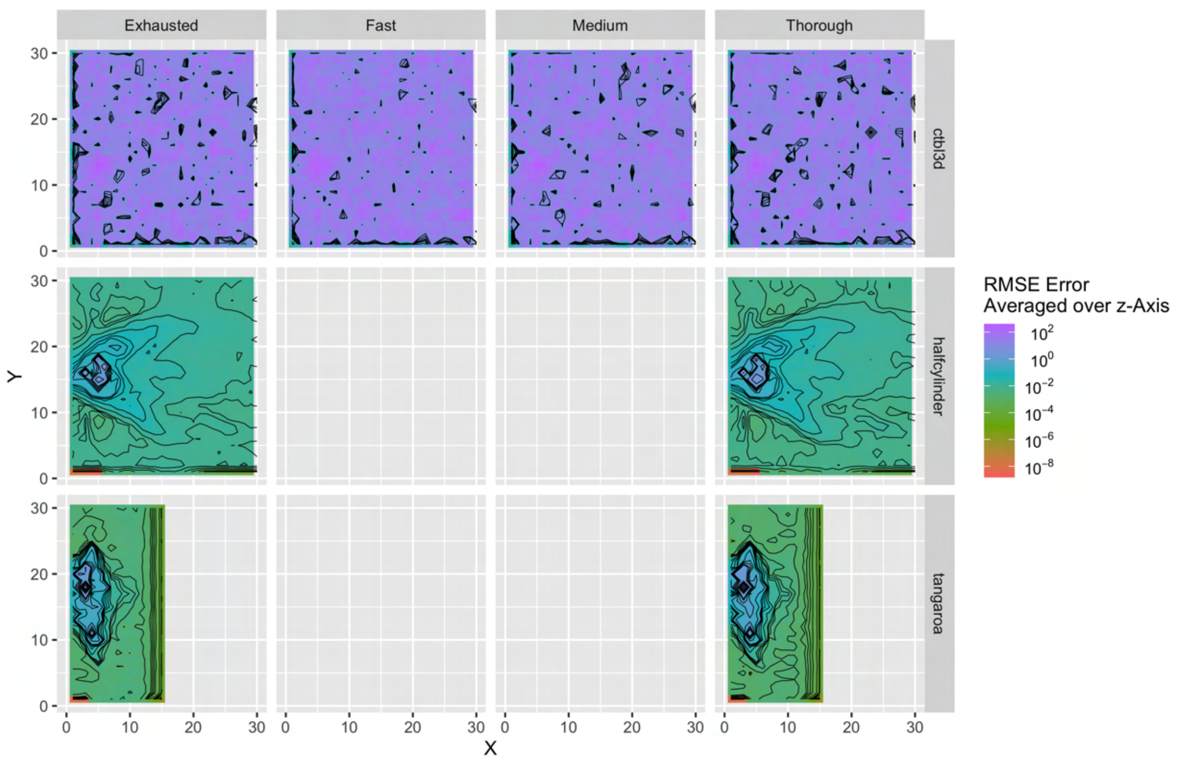
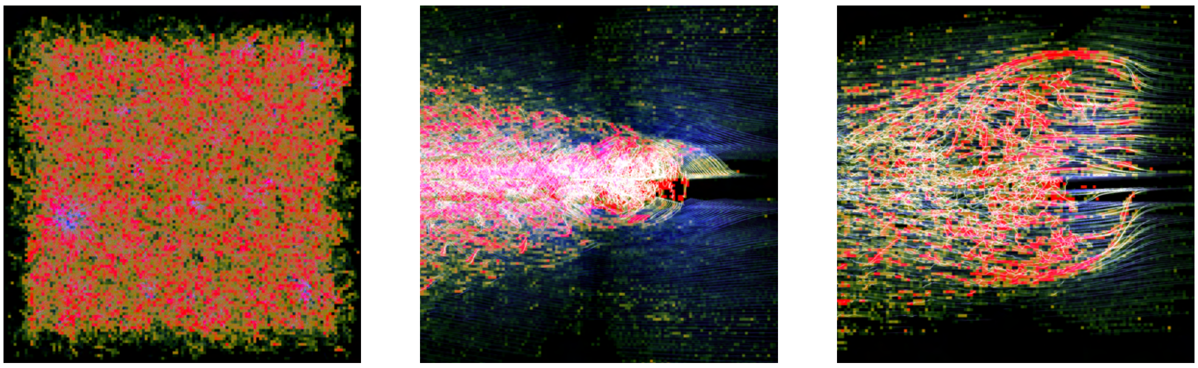


Figure 5.3.: Individual average trajectory error projected to the XY plane. Visualization in seed space, where a grid node represents a trajectories initial seed. Projection was performed by averaging over the Z axis and contour lines are established to separate regions of different error magnitude. Values were obtained using the 4x4 block size and VN method.

5.2. Visual Error

Next, the visual error is evaluated for each dataset at the highest compression settings. Multiple images were taken for each dataset from unique viewpoints of the source and compressed integration results. The error was measured in a sliding-window based SSIM approach as described in Section 3.4. Figure 5.4 shows selected visual samples for each dataset's local SSIM values. The images were obtained from the following perspectives: (a) front, (b) back & (c) back. Visual examination shows similar behavior as the previous analysis, where the error is lower in regions of laminar flow and higher in regions of turbulent flow. The global SSIMs of the scenarios shown in Figure 5.4 lie within the range 0.658 and 0.79 which indicate a fairly high amount of distortion. Also, it must be noted that the shown images display subregions of the original, which exhibit a certain amount of empty space that dampens the global SSIM error.

The global SSIM varies considerably between datasets and is highly dependent on the chosen viewpoint. Table 5.5 shows the global SSIM values for each scenario. The



(a) Clouds
Avg. SSIM: 0.711

(b) Half Cylinder
Avg. SSIM: 0.658

(c) Tangaroa
Avg. SSIM: 0.79

Figure 5.4.: Visualized local SSIM error calculated at a square window size of 11. Red regions mean high deviations from ground truth. Images are subregions of the originals showing the high dynamic regions, whereas the SSIM was calculated on the whole image.

corresponding images are shown in the appendix. Observations exhibit lower errors in scenarios which capture a macroscopic view of the whole dataset whereas the results deteriorate quickly for detailed close-up captures of singular features.

Table 5.5.: SSIM measures of image pairs between source and compressed trajectories. The letters A-H resemble the different unique viewpoints from which each image was evaluated. The compressed representatives were obtained using Exhaustive compression, 4x4 block size and the VN method. All images can be found in the Appendix.

Dataset	A	B	C	D	E	F	G	H
Clouds	0.711	0.236	0.920	0.399	0.454	0.349		
Halfcylinder	0.818	0.836	0.658	0.422	0.866	0.294	0.232	0.960
Tangaroa	0.918	0.790	0.806	0.924	0.440	0.509	0.589	

5.3. Performance

Compared to prior observations, the investigation of advection performance yielded the most prominent results. The measures referenced in this section were taken for the source and compressed datasets. The latter ones are represented by the most optimized Exhaustive preset, 4x4 block resolution and both SN and VN method.

For large unsteady datasets the data upload processes to GPU consumed the most amount of time compared to the compute process. Thereby the source datasets of Half Cylinder and Tangaroa had accumulated integration timings of 1.5 seconds or less, which

Table 5.6.: Timings of the data upload processes. Values obtained from compressed datasets with Exhaustive preset and 4x4 blocksize configuration.

Dataset	Method	Avg [ms]	Max [ms]	Min [ms]	Total [s]	Ratio
Clouds	Source	38	38	38	0.038	
	SN	2.5	2.5	2.5	0.0025	6.5%
	VN	2.7	2.7	2.7	0.0027	6.9%
Halfcylinder	Source	1033.9	4511.9	49.5	155.1	
	SN	239.7	6372.2	2.9	35.9	23.2%
	VN	225.7	5714.4	2.8	33.9	21.8%
Tangaroa	Source	537.2	4622.8	24.2	107.44	
	SN	121.9	3348.6	1.6	24.37	22.7%
	VN	122.5	3216.3	1.6	24.5	22.8%

Table 5.7.: Timings of the particle advection (compute) processes. Values obtained from compressed datasets with Exhaustive preset and 4x4 blocksize configuration.

Dataset	Method	Avg [ms]	Max [ms]	Min [ms]	Total [ms]	Ratio
Clouds	Source	244.3	244.3	244.3	244.3	
	SN	406.1	406.1	406.1	406.1	166.2%
	VN	448.1	448.1	448.1	448.1	183.4%
Halfcylinder	Source	6.7	8.7	5.6	1013.1	
	SN	7.8	16.0	7.2	1170.6	115.5%
	VN	9.6	11.4	8.4	1443.7	142.5%
Tangaroa	Source	1.2	2.3	1.0	247.3	
	SN	1.3	2.4	1.0	270.6	109.4%
	VN	1.5	2.9	1.1	303.3	122.6%

were two magnitudes lower compared to accumulated upload timings of 155.1 seconds for the Half Cylinder dataset. Tables 5.6 and 5.7 show the obtained timings for all datasets normalization methods. With compression applied, the data upload time could be reduced by 86.8% in case of the Half Cylinder dataset and similarly for the Tangaroa scenario. The Clouds dataset was even able to reduce the upload time by 93.5%. However, the Clouds dataset also had significantly smaller upload times compared to the unsteady candidates. The compression also imposed an overhead to the integration process. Invocation of the Compute Shader lasted 9.4% to 83.4% longer, depending on the dataset and the applied normalization method. VN normalized datasets needed 12 Shader Storage Buffer accesses per vector for denormalization which were responsible for on average 19.1% higher timings compared to SN normalized datasets with only 2 buffer accesses per vector.

Table 5.8.: Timings of each datasets preprocessing before compression in seconds. Measurements were taken on the high performance workstation.

Dataset	Method	Load File [s]	Find Peaks [s]	Convert [s]
Clouds	SN	0.502	0.314	0.908
	VN	0.598	0.436	1.627
Halfcylinder	SN	54.180	34.417	90.787
	VN	57.723	38.522	158.605
Tangaroa	SN	41.767	25.381	64.195
	VN	39.031	23.988	113.916

The pre-processing imposes an additional runtime overhead to be executed once per dataset. Besides the implementation and dataset dependent steps of loading the file, gathering information for normalization and converting the dataset into a suiting representation, as presented in Table 5.8, especially the encoding itself created a large time overhead. The encoding process performance was heavily influenced by the choice of preset and the encoder's ability to quickly find a good encoding. In Table 5.9 the compression behavior across the datasets and presets is presented. Timings decrease with lower presets in each scenario, but there was a gap between the observations of the steady dataset and the unsteady ones. Compared to the Clouds dataset, the unsteady datasets showed a significantly higher encoding throughput which was able to process three times the data on average. Lower presets were investigated for the Clouds dataset only, but these showed also great acceleration potential over the Thorough preset. The lowest preset promised the highest average encoding throughput by a ratio of 3 compared to the preceding Medium preset.

Table 5.9.: The size of each dataset in Megabyte divided by the timings of each datasets compression process in seconds. Measurements were taken on the high performance workstation.

Dataset	Method	Exhaustive $[\frac{MB}{s}]$	Thorough $[\frac{MB}{s}]$	Medium $[\frac{MB}{s}]$	Fast $[\frac{MB}{s}]$
Clouds	SN	0.614	1.065	2.089	7.565
	VN	0.623	1.064	1.859	6.268
Halfcylinder	SN	0.987	3.184		
	VN	0.968	3.103		
Tangaroa	SN	0.804	2.828		
	VN	0.795	2.612		

CHAPTER 6

DISCUSSION

The conducted study pursued the answers to two critical questions regarding volumetric compression in the context of scientific visualization: How accurate are the results and how fast can they be computed? The preceding observations have shown that the global precision differs strongly between a consistently dynamic behaving dataset and a mostly laminar flow system. A chaotic flow as present in the Clouds dataset resulted in an average deviation of 5.4 cells distance which can already result in strongly altered trajectories as eventually no original grid node is used for integration anymore. The spatial error distribution obtained per trajectory seen in Figure 5.3 and the visual examination demonstrate both that this error is not raised by outliers but can be localized throughout the visualization. The unsteady datasets comprised of mostly laminar flow achieved much better global error scores around 0.2 and 0.33, but also show strong local deterioration in regions of chaotic flow behavior imposed by the presence of obstacles. The local SSIM measures support this observation and detect strong image-level alterations from the source dataset in regions where the trajectories show high curvature. Next to the mandatory precision loss of lossy compression, there are multiple possible reasons to these observations: The incapability of the encoder to adequately represent a fixed region of grid nodes with a limited number of per-texel weights and color endpoints, a possibly negative impact of the applied normalization method or bad optimization heuristics of the encoder itself. Fixed-size region encoding captures many grid nodes, 16 at a minimum, but the number of partitions holding different color endpoint pairs and weights assignable per texel are strongly limited. Only two color channels can be compressed uncorrelated due to the maximum of two weights and if uncorrelated encoding is active the maximum number of partitions is limited to three[30] which may not be enough to encode 16 3D vectors of quickly changing vector field regions. A significant impact of normalization on accuracy can be detected in the Clouds dataset and was seen as a trend across all datasets. This observation rises the assumption that the normalization heuristics especially influence regions of dynamic behavior which is more visible in the Clouds dataset and suppressed for the unsteady datasets. An extended investigation of this approach using an HDR color profile capable texture compression is needed to rule out any error imposed by normalization in chaotic regions. Finally, the encoder showed a bad optimization habit for the Clouds SN scenario which had best error scores at the least optimizing Fast preset. Though the vector field is being interpreted as colors, a PSNR based optimization approach may not be suited for the input data. An angular based error metric is also offered by the encoder, however only for normalized input vectors with unit magnitude. In this case the encoder omits the third component as it can be recomputed manually later, given the remaining available information. However, the chosen implementation approach did not normalize the vector to unit magnitude and therefore this error metric was not available for block optimization. Comparative contributions for 3D vector field visualization all chose custom algorithms which needed a shader dependent decompression algorithm but also offered more freedom for choosing or implementing a custom encoder [11, 19, 45]. Despite different visualization approaches were investigated, all reported a successful acceleration of the whole visualization process while still maintaining acceptable accuracy. A fair comparison of error scores for different datasets and implementations is not possible, but it can give a rough estimate about the relative accuracy. *Treib et al.* investigated the subject of particle tracing in turbulent 4D vector fields with respect to the trajectory RMS distance error as well. Their observations translated into uniform grid space showed comparable accuracy error which was slightly lower with a score of 3.2 compared to the Clouds datasets respective value of 5.4 [45].

Since a loss in accuracy is deemed acceptable for lossy compression techniques a significant acceleration of the visualization process is expected in exchange. The presented approach achieved impressive results with the underlying test system and was able to reduce the advection runtime down to 22%. This however only holds true for large datasets as upload timings could be reduced whereas the compute stages recorded an increase in runtime. The compute stage slowdown was expected due to the numerous adjustments needed to enable ASTC encoding in the first place, i.e. normalization and additional texture accesses because of the sliced 3D approach. But it is also seen as negligible for realistically sized datasets since here the compute stage needed around 1 second at most compared to 155 seconds of pure upload times. Comparative studies achieved 10% higher performance ratios for their data upload stage and less penalty for their compute stage despite a custom shader programmed decompression.[45] But exact performance comparisons are difficult, since this approach was limited to an integrated GPU which does not share the advantages of dedicated fast video memory and a huge amount of shader cores.

The encoding process itself showed that choosing the lowest accurate Fast preset can have benefits over the more optimizing presets. Compression duration strongly depended on the applied preset determining the optimization expenses and the ability of the encoder to find a good encoding fast. Flow fields with mostly laminar behavior were easier to encode and achieved a three times faster encoding throughput compared to the Clouds dataset at Thorough preset. However, encoding the larger scale datasets with higher precision presets needed an immense amount of time even on a powerful workstation with concurrent encoding of each timeslice. The Fast preset on the other hand allowed a very high throughput efficiency for the Clouds dataset while exhibiting only slightly higher error scores. An investigation of the unsteady datasets at this preset is not available, but if the same amount of throughput is assumed the encoding duration could be reduced by 75% compared to the Exhaustive preset. Even faster results can be expected if the encoder still benefits from the better optimizable behavior of laminar flow. If the additional error for the unsteady datasets resides in the same range as for the Clouds scenario, the encoding performance could prove a suitable compromise to enable fast visualization of the dataset.

Chapter 7

Conclusion

In this study the applicability of ASTC compression on 3D and 4D vector fields in the context of particle advection has been investigated. Subject of analysis was the accuracy by which the trajectories could be reconstructed using the compressed dataset and the performance of the visualization and encoding processes. It could be shown that ASTC compression is well applicable if an accuracy loss in regions of very dynamic flow behavior is acceptable. The runtime of visualization processes could be reduced though the computation itself experienced an additional runtime overhead due to normalization and manual depth interpolation. Since computation runtime resided in the range of one second the absolute overhead imposed by the additional operations was negligible. The encoding performance was dependent on the encoders ability to find good encodings fast and the chosen preset. It was shown that choosing the best optimized preset does not always guarantee the best encoding and does not justify the additional runtime needed to fully encode the dataset. As the error stayed in the same magnitude for all scenarios that have been tested choosing the least optimizing preset is suggested as the best compromise for good performance of the complete visualization pipeline including the pre-processing at the cost of accuracy. Choosing a greater block size resolution on the other hand is generally not advised as the accuracy decreases significantly and fast. Further investigation of texture compression in the context of scientific visualization is advised as the results are overall promising in the face of the optimizations that can and should be made in future works. Major benefits are expected by the choice of a different sophisticated texture compression like BC6H which is fully supported by todays GPUs and is designed for the HDR color profile that offers a greater range of values in the encoding.

Chapter 8

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, et al. *Real-time rendering.* AK Peters/CRC Press, 2018.

[2] Irene Baeza Rojo and Tobias Günther. "Vector Field Topology of Time-Dependent Flows in a Steady Reference Frame". In: *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE Scientific Visualization)* (2019).

[3] M. Balsa Rodríguez et al. "State-of-the-Art in Compressed GPU-Based Direct Volume Rendering". In: *Comput. Graph. Forum* 33.6 (Sept. 2014), pp. 77–100. ISSN: 0167-7055.

[4] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. "State-of-the-Art in GPU-Based Large-Scale Volume Visualization". In: *Computer Graphics Forum* 34.8 (2015), pp. 13–37.

[5] T. Blu, P. Thevenaz, and M. Unser. "Linear interpolation revitalized". In: *IEEE Transactions on Image Processing* 13.5 (2004), pp. 710–719.

[6] J. Chen et al. "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.

[7] *Computer Graphics Laboratory of ETH Zürich.* URL: `https://cgl.ethz.ch/research/visualization/data.php`.

[8] Dan Dolonius et al. "Compressing Color Data for Voxelized Surface Geometry". In: *IEEE Transactions on Visualization and Computer Graphics* PP (Aug. 2017), pp. 1–1.

[9] T. Frühauf. "Raycasting vector fields". In: *Proceedings of Seventh Annual IEEE Visualization '96.* 1996, pp. 115–120.

[10] Antonio Galbis and Manuel Maestre. *Vector analysis versus vector calculus.* Springer Science & Business Media, 2012.

[11] Tomáš Golembiovský. "Compression of vector field changing in time". PhD thesis. Masarykova univerzita, Fakulta informatiky, 2010.

[12] Alain Horé and Djemel Ziou. "Image Quality Metrics: PSNR vs. SSIM". In: *2010 20th International Conference on Pattern Recognition.* 2010, pp. 2366–2369.

[13] V. Interrante and C. Grosch. "Strategies for effectively visualizing 3D flow with volume LIC". In: *Proceedings. Visualization '97 (Cat. No. 97CB36155).* 1997, pp. 421–424.

[14] *Introducing RDNA Architecture.* Tech. rep. Advanced Micro Devices, Inc., 2019.

[15] Johannes Kehrer and Helwig Hauser. "Visualization and Visual Analysis of Multifaceted Scientific Data: A Survey". In: *IEEE Transactions on Visualization and Computer Graphics* 19 (Mar. 2013), pp. 495–513.

[16] *Khronos Data Format Specification.* Version 1.3.1. Apr. 2020. URL: https://www.khronos.org/registry/DataFormat/specs/1.3/dataformat.1.3.inline.html.

[17] J. Kruger et al. "A particle system for interactive visualization of 3D flows". In: *IEEE Transactions on Visualization and Computer Graphics* 11.6 (Nov. 2005), pp. 744–756. ISSN: 1941-0506.

[18] E. LaMar, B. Hamann, and K. I. Joy. "Multiresolution techniques for interactive texture-based volume visualization". In: *Proceedings Visualization '99 (Cat. No.99CB37067).* 1999, pp. 355–543.

[19] Xin Liang et al. "Toward Feature-Preserving 2D and 3D Vector Field Compression". In: *2020 IEEE Pacific Visualization Symposium (PacificVis).* 2020, pp. 81–90.

[20] Arm Limited. *ASTC Encoder.* 2020. URL: https://github.com/ARM-software/astc-encoder.

[21] E. B. Lum, Kwan-Liu Ma, and J. Clyne. "Texture hardware assisted rendering of time-varying volume data". In: *Proceedings Visualization, 2001. VIS '01.* 2001, pp. 263–563.

[22] N Max. "Progress in Scientific Visualization". In: *The Visual Computer* 21.12 (Nov. 2004).

[23] Tony McLoughlin et al. "Over Two Decades of Integration-Based, Geometric Flow Visualization". In: *Computer Graphics Forum* 29.6 (2010), pp. 1807–1829.

[24] Daisuke Nagayasu, Fumihiko Ino, and Kenichi Hagihara. "Two-stage compression for fast volume rendering of time-varying scalar data". In: Nov. 2006, pp. 275–284.

[25] Simon P. Neill and M. Reza Hashemi. “Chapter 8 - Ocean Modelling for Resource Characterization”. In: *Fundamentals of Ocean Renewable Energy.* Ed. by Simon P. Neill and M. Reza Hashemi. E-Business Solutions. Academic Press, 2018, pp. 193–235. ISBN: 978-0-12-810448-4.

[26] J. Nickolls and W. J. Dally. “The GPU Computing Era”. In: *IEEE Micro* 30.2 (2010), pp. 56–69.

[27] *NV_texture_compression_vtc.* URL: `https://www.khronos.org/registry/OpenGL/extensions/NV/NV_texture_compression_vtc.txt`.

[28] *NVIDIA Ampere GA102 GPU Architecture.* Tech. rep. V2.0. NVIDIA Corporation, 2021.

[29] J. Nystad et al. “Adaptive Scalable Texture Compression”. In: *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics.* EGGH-HPG’12. Paris, France: Eurographics Association, 2012, pp. 105–114.

[30] *OpenGL Graphics System: A Specification.* Version 4.5 (Core Profile). June 2017. URL: `https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf`.

[31] John Owens et al. “GPU computing”. In: *Proceedings of the IEEE* 96 (May 2008), pp. 879–899.

[32] S. Popinet. “Free Computational Fluid Dynamics”. In: *ClusterWorld* 2.6 (2004). URL: `http://gfs.sf.net/`.

[33] Stéphane Popinet, Murray Smith, and Craig Stevens. “Experimental and Numerical Study of the Turbulence Characteristics of Airflow around a Research Vessel”. In: *Journal of Atmospheric and Oceanic Technology* 21.10 (2004), pp. 1575–1589.

[34] William H Press et al. *Numerical recipes 3rd edition: The art of scientific computing.* Cambridge university press, 2007.

[35] P. Ratanaworabhan, Jian Ke, and M. Burtscher. “Fast lossless compression of scientific floating-point data”. In: *Data Compression Conference (DCC’06).* 2006, pp. 133–142.

[36] C. Rezk-Salama et al. “Interactive exploration of volume line integral convolution based on 3D-texture mapping”. In: *Proceedings Visualization ’99 (Cat. No.99CB37067).* Oct. 1999, pp. 233–528.

[37] Khalid Sayood. “Data Compression.” In: *Encyclopedia of Information Systems* 1 (2002), pp. 423–444.

[38] Graham Sellers, Richard S. Wright, and Nicholas Haemel. *OpenGL Superbible: Comprehensive Tutorial and Reference.* 7th. Addison-Wesley Professional, 2015.

[39] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1.* 7th. Addison-Wesley Professional, 2009.

[40] Bjorn Stevens. *Introduction to UCLA-LES.* 2013.

[41] John Tabak. *Geometry: the language of space and form.* Infobase Publishing, 2014.

[42] Christian Teitzel, Roberto Grosso, and Thomas Ertl. “Efficient and reliable integration methods for particle tracing in unsteady flows on discrete meshes”. In: *Visualization in Scientific Computing’97.* Springer, 1997, pp. 31–41.

[43] H. Theisel, Ch. Rössl, and H.-P. Seidel. “Compression of 2D Vector Fields Under Guaranteed Topology Preservation”. In: *Computer Graphics Forum* 22.3 (2003), pp. 333–342.

[44] Marc Treib et al. “Analyzing the Effect of Lossy Compression on Particle Traces in Turbulent Vector Fields.” In: *IVAPP.* 2015, pp. 279–288.

[45] Marc Treib et al. “Compression and Heuristic Caching for GPU Particle Tracing in Turbulent Vector Fields”. In: vol. 598. Feb. 2016, pp. 144–165. ISBN: 978-3-319-29970-9.

[46] Marc Treib et al. “Interactive Editing of GigaSample Terrain Fields”. In: *Computer Graphics Forum* 31 (May 2012), pp. 383–392.

[47] Marc Treib et al. “Turbulence Visualization at the Terascale on Desktop PCs”. In: *Visualization and Computer Graphics, IEEE Transactions on* 18 (Dec. 2012), pp. 2169–2177.

[48] Vivek Verma and Alex Pang. “Comparative flow visualization”. In: *Visualization and Computer Graphics, IEEE Transactions on* 10 (Dec. 2004), pp. 609–624.

[49] Zhou Wang et al. “Image quality assessment: from error visibility to structural similarity”. In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612.

[50] P. Williams, S. Uselton, and Marisa K. Chancellor. “Foundations for Measuring Volume Rendering Quality”. In: 1997.

[51] Xiangong Ye, D. Kao, and A. Pang. “Strategy for seeding 3D streamlines”. In: *VIS 05. IEEE Visualization, 2005.* Oct. 2005, pp. 471–478.

[52] R. Yagel et al. “Hardware assisted volume rendering of unstructured grids by incremental slicing”. In: *Proceedings of 1996 Symposium on Volume Visualization.* Oct. 1996, pp. 55–62.

Appendices

Appendix A
Additional Results

A.1. Geometric Error

Table A.1.: Normalized RMS of the average vector component error for each data set with different compression presets and normalization methods. Range based Normalization of the RMS by componentwise global maximum and minimum of the dataset. Values taken from 4x4 block size configuration.

Dataset			Exhaustive	Thorough	Medium	Fast
Clouds	SN	X	2.321e-3	2.521e-3	2.672e-3	3.075e-3
		Y	1.878e-3	2.044e-3	2.164e-3	2.521e-3
		Z	1.480e-3	1.512e-3	1.538e-3	1.820e-3
	VN	X	2.012e-3	2.157e-3	2.295e-3	2.592e-3
		Y	1.681e-3	1.814e-3	1.932e-3	2.235e-3
		Z	1.797e-3	1.919e-3	1.995e-3	2.223e-3
Halfcylinder	SN	X	4.826e-4	5.461e-4		
		Y	4.695e-4	5.436e-4		
		Z	6.665e-4	7.408e-4		
	VN	X	5.610e-4	6.564e-4		
		Y	5.072e-4	5.932e-4		
		Z	3.585e-4	4.027e-4		
Tangaroa	SN	X	4.799e-4	5.449e-4		
		Y	5.819e-4	6.781e-4		
		Z	7.267e-4	8.290e-4		
	VN	X	5.485e-4	6.410e-4		
		Y	5.437e-4	6.272e-4		
		Z	6.069e-4	6.829e-4		

Table A.2.: Normalized RMS of the average vector component error for each data set with different compression presets and normalization methods. Range based Normalization of the RMS by componentwise global maximum and minimum of the dataset. Measurements at lower presets are set relative to the Exhaustive preset. Values taken from 4x4 block size configuration.

Dataset	Method	Dim.	Exhaustive	Thorough	Medium	Fast
Clouds	SN	X	2.321**e-3**	108.6%	115.1%	132.5%
		Y	1.878**e-3**	108.8%	115.2%	134.2%
		Z	1.480**e-3**	102.2%	103.9%	123%
	VN	X	2.012**e-3**	107.2%	114.1%	128.8%
		Y	1.681**e-3**	107.9%	114.9%	132.9%
		Z	1.797**e-3**	106.8%	111%	123.7%
Halfcylinder	SN	X	4.826**e-4**	113.2%		
		Y	4.695**e-4**	115.8%		
		Z	6.665**e-4**	111.1%		
	VN	X	5.610**e-4**	117%		
		Y	5.072**e-4**	116.9%		
		Z	3.585**e-4**	112.3%		
Tangaroa	SN	X	4.799**e-4**	113.5%		
		Y	5.819**e-4**	116.5%		
		Z	7.267**e-4**	114.1%		
	VN	X	5.485**e-4**	98.8%		
		Y	5.437**e-4**	115.3%		
		Z	6.069**e-4**	112.5%		

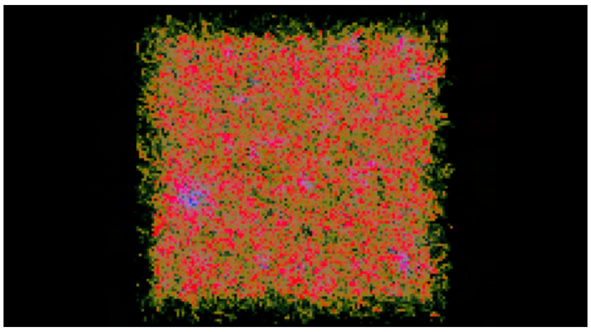
Table A.3.: RMS of the average and maximum trajectory error for each dataset and normalization method using different block size configurations. Measurements at lower resolutions are set relative to the 4x4 block size resolution. Values were obtained using the Exhaustive preset for compression.

Dataset	Method	Error	4x4	5x5	6x6	12x12
Clouds	SN	AVG	6.739	104.5%	107.8%	130.9%
		MAX	12.280	104.8%	106.6%	126.7%
	VN	AVG	5.410	109.1%	117.5%	151.1%
		MAX	10.587	107.8%	113.6%	138.7%
Halfcylinder	SN	AVG	0.207	105.3%	132.4%	
		MAX	0.408	105.9%	137.7%	
	VN	AVG	0.203	110.3%	126.0%	
		MAX	0.406	113.3%	131.8%	
Tangaroa	SN	AVG	0.330	99.99%	106.4%	
		MAX	0.605	101.0%	106.9%	
	VN	AVG	0.330			
		MAX	0.615			

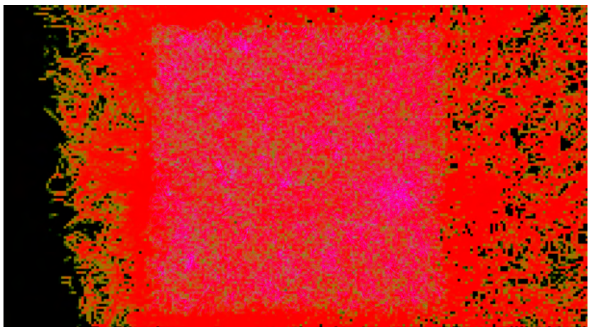
Table A.4.: RMS of the average and maximum trajectory error for each dataset and normalization method using different compression presets. Measurements at lower presets are set relative to the Exhaustive preset. Values were obtained using the 4x4 block size configuration for compression.

Dataset	Method	Error	Exhaustive	Thorough	Medium	Fast
Clouds	SN	AVG	6.739	102.6%	107.9%	92.1%
		MAX	12.280	102.4%	107.2%	94.1%
	VN	AVG	5.410	102.9%	106.4%	109.2%
		MAX	10.587	102.3%	105.0%	108.1%
Halfcylinder	SN	AVG	0.207	101.9%		
		MAX	0.408	105.1%		
	VN	AVG	0.203	102.9%		
		MAX	0.406	102.2%		
Tangaroa	SN	AVG	0.330	99.7%		
		MAX	0.605	100.1%		
	VN	AVG	0.330	0.0%		
		MAX	0.615	99.5%		

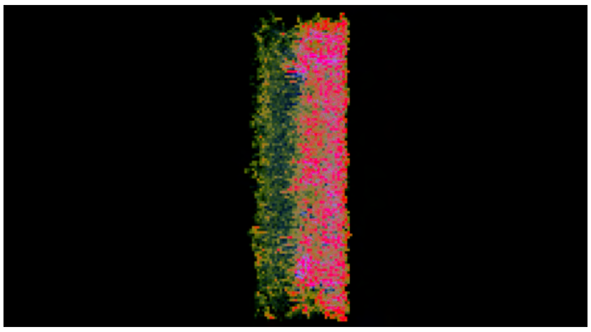
A.2. Visual Error



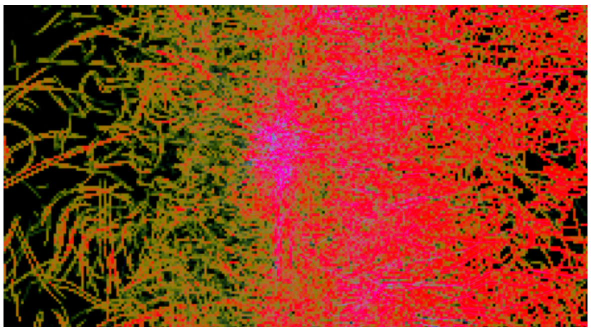
(a) Clouds A
Avg. SSIM: 0.711



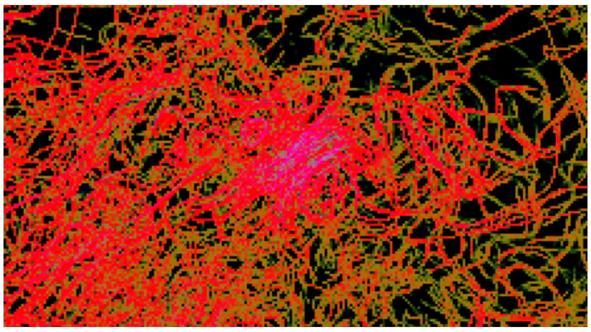
(b) Clouds B
Avg. SSIM: 0.236



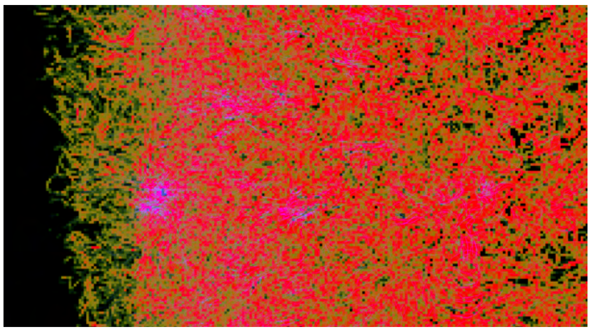
(c) Clouds C
Avg. SSIM: 0.920



(d) Clouds D
Avg. SSIM: 0.399

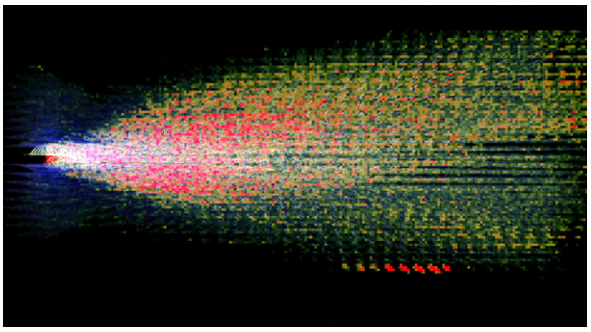


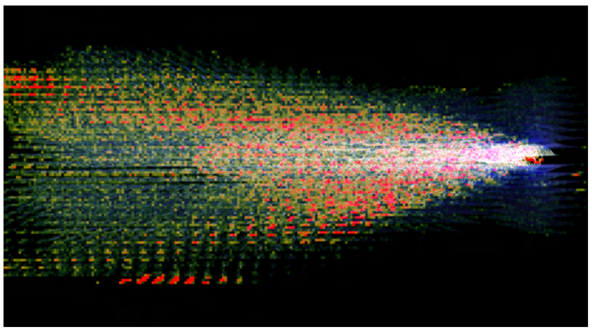
(e) Clouds E
Avg. SSIM: 0.454

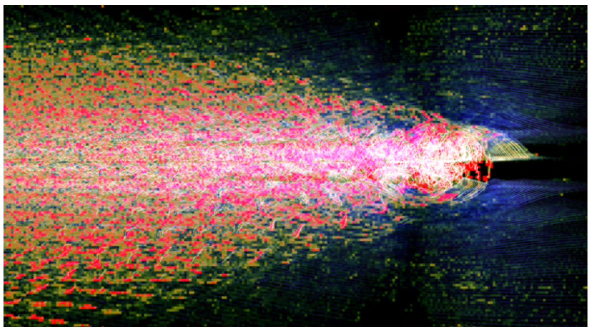


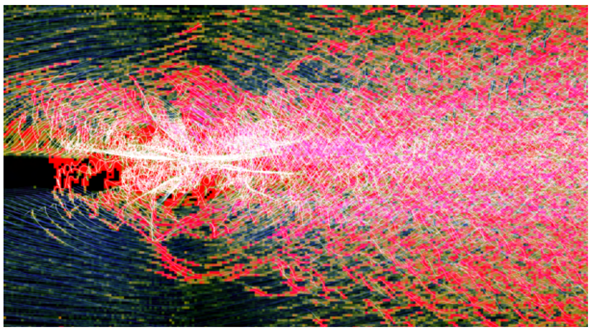
(f) Clouds F
Avg. SSIM: 0.349

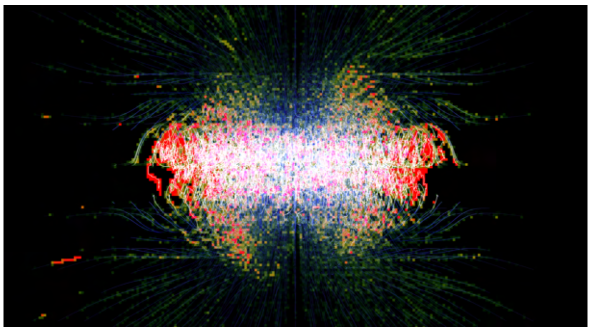
Figure A.1.: Visualized local SSIM error calculated at a square window size of 11. Red regions mean high deviations from ground truth.

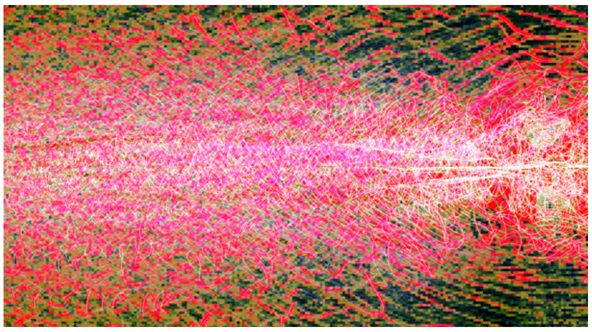

(a) Half Cylinder A
Avg. SSIM: 0.818

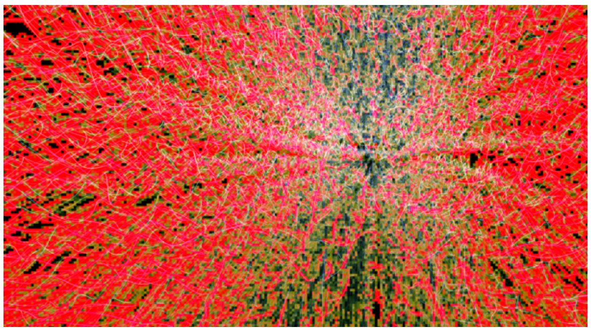

(b) Half Cylinder B
Avg. SSIM: 0.836


(c) Half Cylinder C
Avg. SSIM: 0.658


(d) Half Cylinder D
Avg. SSIM: 0.422


(e) Half Cylinder E
Avg. SSIM: 0.866


(f) Half Cylinder F
Avg. SSIM: 0.294


(g) Half Cylinder G
Avg. SSIM: 0.232

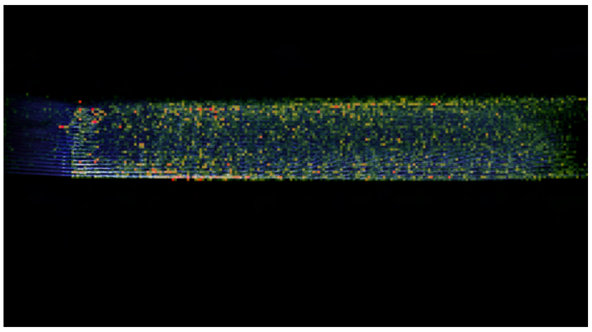
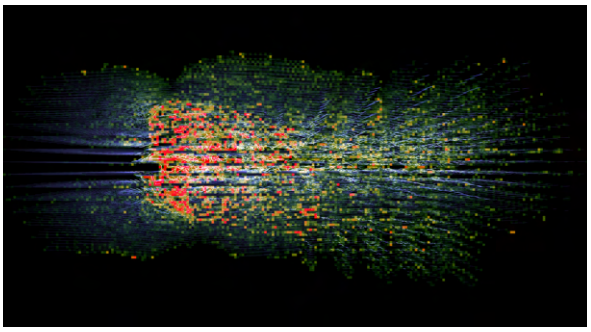
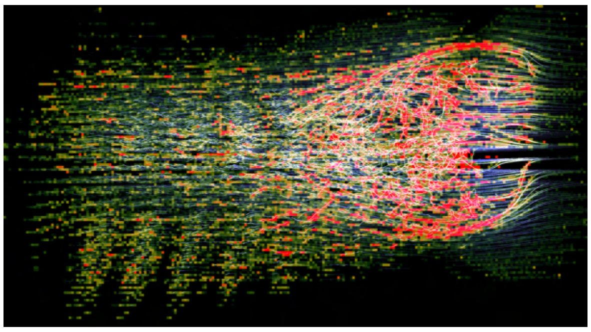

(h) Half Cylinder H
Avg. SSIM: 0.960

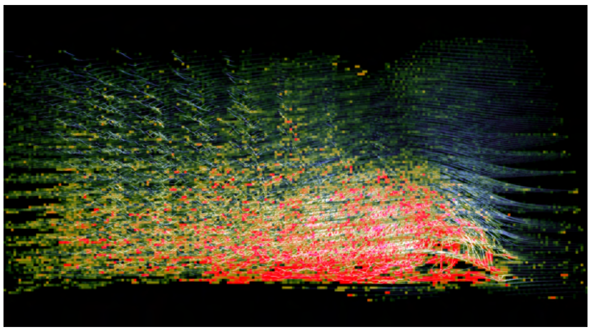
Figure A.2.: Visualized local SSIM error calculated at a square window size of 11. Red regions mean high deviations from ground truth.



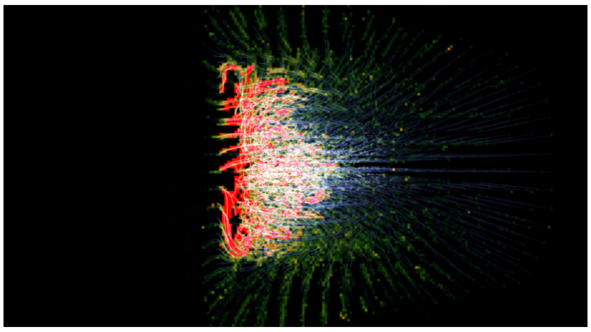
(a) Tangaroa A
Avg. SSIM: 0.918



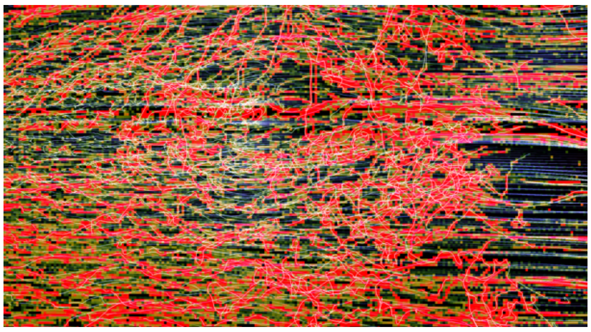
(b) Tangaroa B
Avg. SSIM: 0.790



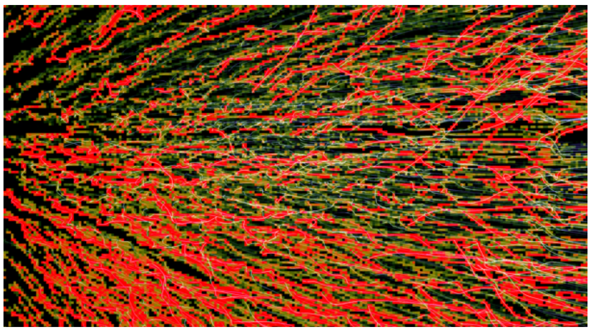
(c) Tangaroa C
Avg. SSIM: 0.806



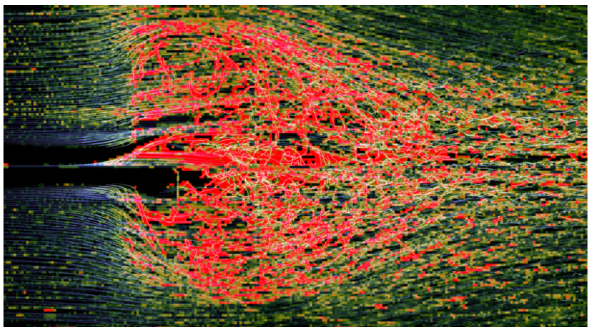
(d) Tangaroa D
Avg. SSIM: 0.924



(e) Tangaroa E
Avg. SSIM: 0.440



(f) Tangaroa F
Avg. SSIM: 0.509



(g) Tangaroa G
Avg. SSIM: 0.589

Figure A.3.: Visualized local SSIM error calculated at a square window size of 11. Red regions mean high deviations from ground truth.

A.3. Performance

Table A.5.: Timings of each datasets compression process in seconds. Measurements were taken on the high performance workstation.

Dataset	Method	Exhaustive [s]	Thorough [s]	Medium [s]	Fast [s]
Clouds	SN	357	206	105	29
	VN	352	201	118	35
Halfcylinder	SN	21, 516	6, 668		
	VN	21, 944	6, 843		
Tangaroa	SN	18, 537	5, 270		
	VN	18, 741	5, 707		

Table A.6.: Timings of the compression processes for each dataset and normalization method using different presets. Measurements at lower presets are set relative to the Exhaustive preset. Values were obtained using the 4x4 block size configuration.

Dataset	Method	Exhaustive	Thorough	Medium	Fast
Clouds	SN	357s	57.7%	29.4%	8.1%
	VN	352s	57.1%	33.5%	9.9%
Halfcylinder	SN	21, 516s	31%		
	VN	21, 944s	31.2%		
Tangaroa	SN	18, 537s	28.4%		
	VN	18, 741s	30.4%s		