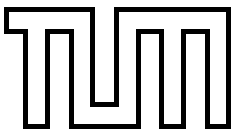

Technische Universität München
Fakultät für Informatik
Lehrstuhl für Computer Graphik und Visualisierung

GPU-Based Compression for Large-Scale Visualization

Marc Treib

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. M. G. Bader
Prüfer der Dissertation:	1. Univ.-Prof. Dr. R. Westermann
	2. Univ.-Prof. Dr. J. Krüger, Universität Duisburg-Essen

Die Dissertation wurde am 30.01.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.05.2014 angenommen.

To my family and friends

Abstract

Scientific visualization is used to aid in the analysis and exploration of numerical data sets. However, in recent years, the size of scientific data sets has increased to such an extent that typical visualization approaches become problematic. Single pictures or even videos can not contain even the most essential features of the data anymore, so interactivity becomes crucial in finding regions or features of interest. On the other hand, data sizes in the order of terabytes make it challenging to achieve interactive rates. As keeping all data in fast memory is not possible, data must be streamed from slower storage media such as hard disks. Data compression can be employed to reduce both I/O bandwidth requirements and memory usage. In interactive applications, the decompression throughput is of crucial importance. At the bare minimum, it must exceed the raw storage bandwidth in order for the compression to have any positive effect on the streaming performance. In this thesis, I present a reusable compression layer for arbitrary data given on 2D or 3D Cartesian grids. It employs algorithms which are well-known in image compression and thus achieves state-of-the-art compression rates. It is specifically tailored to exploit the highly parallel nature of contemporary GPUs. This is made possible by data-parallel formulations of all computation steps as well as a highly efficient implementation using NVIDIA's CUDA platform. To demonstrate the utility and versatility of the library, I have integrated it into three applications. The first application is a terrain rendering system allowing the interactive editing of very large and highly resolved terrain data

sets. The second is a turbulence visualization system which employs volume rendering of derived flow properties. The compression layer allows it to handle data sets where even a single time step is too large to fit into main memory. The third application enables particle tracing in extremely large flow fields. On a single desktop PC, it achieves a performance comparable to previous approaches on supercomputers. In all cases, the high compression rate and throughput allow the efficient handling of much larger data sets than would otherwise be possible.

Zusammenfassung

Wissenschaftliche Visualisierung wird bei der Analyse und Exploration numerischer Datensätze eingesetzt. Allerdings sind solche Datensätze in den letzten Jahren so groß geworden, dass viele gängige Visualisierungsansätze schwierig einzusetzen sind. Einzelne Bilder oder selbst Videos können nicht einmal mehr die wesentlichsten Merkmale der Daten darstellen, so dass Interaktivität bei der Suche nach relevanten Features oder Regionen unerlässlich wird. Andererseits machen es gerade die großen Datenmengen schwierig, Interaktivität zu erreichen. Nachdem es nicht mehr möglich ist, alle Daten in schnellem Speicher zu halten, müssen die Daten dynamisch von langsameren Medien wie Festplatten nachgeladen werden. Der Einsatz von Datenkompression kann die Anforderungen bezüglich Bandbreite wie auch Speicherbedarf reduzieren. In interaktiven Anwendungen ist der Dekompressions-Durchsatz von höchster Wichtigkeit. Damit die Datenkompression positive Auswirkungen hat, muss dieser mindestens über der Bandbreite der Speichermedien liegen. In dieser Dissertation präsentiere ich eine wiederverwendbare Softwarebibliothek zur Kompression beliebiger Daten auf kartesischen Gittern in 2D oder 3D. Sie verwendet bekannte Algorithmen aus der Bildkompression und erreicht daher Kompressionsraten, die dem Stand der Technik entsprechen. Sie ist außerdem speziell ausgerichtet auf die Parallelität heutiger GPUs. Das wird ermöglicht durch Daten-parallele Formulierungen aller verwendeten Algorithmen, sowie durch eine effiziente Implementierung basierend auf NVIDIAs CUDA-Plattform. Zur Demonstration des Nutzens

und der Vielseitigkeit der Bibliothek habe ich diese in drei Anwendungen integriert. Die erste Anwendung ist ein System zum Rendern von Terrain-Daten, welches die interaktive Bearbeitung sehr großer und hochauflösender Terrains ermöglicht. Die zweite ist ein Turbulenzvisualisierungssystem, das Techniken aus der Volumenvisualisierung auf abgeleitete Strömungsmerkmale anwendet. Durch den Einsatz von Kompression kann es so große Datensätze behandeln, dass selbst ein einzelner Zeitschritt nicht in den Hauptspeicher passt. Die dritte Anwendung ermöglicht die Berechnung von Partikeltrajektorien in extrem großen Strömungsfeldern. Diese Anwendung erreicht auf einem einzelnen PC eine Leistung, die vergleichbar ist mit früheren Verfahren unter Einsatz von Supercomputern. In allen drei Fällen können dank der hohen Kompressionsrate sowie -geschwindigkeit deutlich größere Datensätze behandelt werden als ansonsten möglich wäre.

Acknowledgments

I gratefully acknowledge the support of all of the people who made this thesis possible. First and foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Rüdiger Westermann for offering me the great possibility to pursue research in the field of scientific visualization. I am very grateful for his guidance, for his commitment to my work, and for the numerous discussions. I also want to thank the co-authors of my papers, Stefan Auer, Kai Bürger, Charles Meneveau, Florian Reichl, and Alexander Szalay, for contributing their suggestions and ideas. I have always enjoyed working with them. Furthermore, I would like to thank my current and former colleagues, Stefan Auer, Kai Bürger, Shunting Cao, Matthäus Chajdas, Ismail Demir, Christian Dick, Florian Ferstl, Roland Fraedrich, Raymund Fülöp, Stefan Hertel, Hans-Georg Menz, Mihaela Mihai, Tobias Pfaffelmoser, Marc Rautenhaus, Florian Reichl, Matthias Reitinger, Jan Sommer, Nils Thuerey, Mikael Vaaraniemi, and Jun Wu, who have always been available for discussions. I am enormously thankful to my parents and my friends for giving me all the support that I needed during this time. I want to thank the King Abdullah University of Science and Technology (KAUST) for funding my work. Finally, I wish to thank the Landesvermessungsamt Feldkirch, Austria, for providing the Vorarlberg terrain data set, as well as Charles Meneveau from Johns Hopkins University, Madhusudhanan Srinivasan from KAUST, and Markus Uhlmann from the Karlsruhe Institute of Technology for providing access to the turbulence data sets.

Contents

1 Introduction

Many scientific disciplines involve the analysis of numerical data sets, produced either by numerical simulations such as in computational fluid dynamics or by measurements, e.g. in geographical applications. The major challenge is to find the "interesting information" contained in the data. This is commonly approached with scientific visualization, which aims to produce images containing the major features. Then, the impressive capabilities of the human visual system can be employed to analyze the data and detect patterns. Interactive visualization techniques are especially useful because they facilitate data exploration. For example, a scientist may focus on a particularly promising region within the data and tune visualization parameters, all with immediate visual feedback.

However, in recent years, it has become increasingly challenging for visualization techniques to keep up with growing data sizes. In 1965, Gordon Moore predicted that the number of components in an integrated circuit would double every year [Moo65]. Looking back, the time interval has turned out to be closer to two years, but the prediction of exponential growth has stood the test of time so far. One consequence is that the computing power of processing units has increased at a similar rate in the past decades. The higher performance of supercomputers in particular has resulted in larger and higher-resolved numerical simulations. Individual pictures or even videos can not contain even the most essential features of the data anymore, so interactivity becomes crucial in finding regions or features of interest. Fortunately, the performance of visualization systems has similarly increased. However, there are

two factors which make it difficult to transform the increased computing power into increased application performance. First, the bandwidths and capacities of transmission and storage devices has not kept up with the computing power. Today, main memory, network adapters, and hard drives are all much slower and smaller than ten years ago, relative to the available computing power at the time. This makes it increasingly hard to avoid memory and bandwidth constraints. Second, in the past years, the serial performance of computers has not increased along with the number of components per circuit. While this has been possible for decades, constraints regarding power consumption and heat dissipation have made it necessary to increase the number of computation units rather than the performance of an individual unit. In consequence, parallel algorithms are now required to fully utilize the available computing power.

Interactive visualization systems employ a variety of techniques to handle the growing amounts of data. As keeping all data in fast memory is not possible, data must be streamed from slower storage media such as hard disks or even from dedicated storage systems. Additionally, region-of-interest and level-of-detail methods can often reduce the amount of data that must be streamed and kept in memory. However, these techniques may not be sufficient to achieve interactivity. They are also not universally applicable. In this situation, data compression can additionally be employed to reduce both I/O bandwidth requirements and memory usage.

In interactive applications, the speed of the decompression step is of crucial importance. At the bare minimum, the throughput must exceed the raw storage bandwidth in order for the compression to have any positive effect on the streaming performance. In applications where data may be modified, a similar argument applies to the compression speed. In previous work, several compression algorithms have been developed which offer great decompression speed, but they usually sacrifice compression speed and/or compression rate to achieve this. Additionally, many such algorithms are very application-specific and can not easily be applied in other scenarios.

In this thesis, I present a reusable compression layer for arbitrary data given on 2D or 3D Cartesian grids. It is called cudaCompress and has been released under a permissive license as a companion to this thesis. It employs algorithms which are well-known in image compression and achieves state-of-the-art compression rates for both lossless and lossy compression. However, in contrast to previous work, it is specifically

tailored to exploit the highly parallel nature of contemporary GPUs. This is achieved by data-parallel formulations of all computation steps as well as a highly efficient implementation using NVIDIA's CUDA platform. The massive computational power and memory bandwidth of current GPUs enables the high compression throughput that is required for interactive applications.

To demonstrate the utility and versatility of CUDACompress, I have integrated it into three applications from interactive visualization and computer graphics. In all cases, the high compression rate and throughput allow the efficient handling of much larger data sets than would otherwise be possible. The first application is an interactive terrain rendering and editing system. Both height fields and orthophotos are compressed using CUDACompress. The high compression ratio as well as decompression throughput allow rapid streaming from disk. In addition, the compression throughput offered by CUDACompress allows edited data to be compressed on-the-fly so that it can be stored to disk again.

The second application is a turbulence visualization system which employs volume rendering of local properties derived from the turbulent flow. The system can handle turbulence data sets which are so large that even a single time step may not fit into RAM. Again, the compression layer enables rapid streaming as well as caching large amounts of compressed data in RAM. Additionally, the system allows filtering the flow data for the purpose of multiscale analysis. With CUDACompress, the filtered data can be compressed and cached for later use.

The third and final application is a particle tracing system for extremely large flow fields on a desktop PC. The use of CUDACompress in combination with a novel smart caching scheme allows the system to achieve a performance comparable to previous approaches on supercomputers. In this application, particular attention must be given to the impact of lossy compression on the computed trajectories. A number of experiments demonstrate that the variations due to lossy compression are of similar magnitude as those induced by numerical interpolation of velocity values.

1.1 Outline

This thesis is structured as follows. Chapter 2 gives a general introduction to the topic of data compression, with a focus on image compression techniques. The fol-

lowing Chapter 3 evaluates the presented compression algorithms regarding their suitability for GPU-friendly data-parallel implementations, and presents in detail a GPU implementation of the discrete wavelet transform as well as run-length and Huffman coding. Chapters 4, 5, and 6 present three applications which are made possible only by this efficient data compression layer. Chapter 4 showcases an interactive terrain editing system capable of handling arbitrarily large, high-resolution terrain data sets. The second application, introduced in Chapter 5, is a visualization system for very large and time-dependent turbulence simulations based on volume rendering of derived quantities. Third, Chapter 6 presents a particle tracing system for extremely large flow fields. All three systems efficiently handle enormous amounts of data using only the limited capabilities of a single desktop PC. Finally, Chapter 7 briefly summarizes the thesis and presents some directions for future work.

1.2 List of Publications

Some of the research results presented in this thesis have been originally published in the following peer-reviewed conference papers and journal articles:

1. Treib M., Reichl F., Auer S., Westermann R.: Interactive editing of gigasample terrain fields. *Computer Graphics Forum 31*, 2 (2012), 383–392. doi:10.1111/j.1467-8659.2012.03017.x.
2. Treib M., Bürger K., Reichl F., Meneveau C., Szalay A., Westermann R.: Turbulence visualization at the terascale on desktop PCs. *IEEE Trans. Vis. Comput. Graphics 18*, 12 (2012), 2169–2177. doi:10.1109/TVCG.2012.274.

Additionally, the source code of cudaCompress, a GPU data compression library which was developed in the course of this thesis, has been made publicly available under a permissive license [Tre13].

2

Data Compression Fundamentals

Data compression has been studied extensively for several decades, and so only a cursory review is possible in the scope of this thesis. This chapter gives a brief summary of the most important concepts and their implementation, with a bias towards those ideas and techniques that will be used in the following chapters. For a more complete covering of the area, the reader is referred to textbooks on data compression. The book by Sayood [Say12] provides a very readable introduction to the topic of data compression, including both theoretical background and detailed descriptions of many practical data compression systems and standards. An even more comprehensive review of most data compression algorithms that were ever in use is given in the Handbook of Data Compression [SM10].

On a very high level, compression systems can be categorized as either *lossless* or *lossy*. This is a crucial difference from the perspective of the application: Many applications can not tolerate any difference between the original and the reconstructed data. Examples include text and computer programs. For these types of data, even a small deviation can radically change the meaning of the data. In other cases, some difference is acceptable as a trade-off for better compression. Here, the most prominent examples are media files. In images or audio recordings, it is usually not essential to exactly reconstruct the original data bit-by-bit. A small change in the color of a pixel will usually not make a significant difference, and may often not even be noticeable.

In practice, almost any data compression system or standard is a combination

of several algorithms taken out of a repertoire of fundamental data compression algorithms. As such, there is a lot of overlap between the design of a lossless and a lossy system. The difference can be as small as adding or removing a quantization stage. The particular application has a much larger impact on the design: A system for the compression of images will look quite different from one designed for text, even if both are lossless. Often, the first stage is an application-dependent preprocessing stage, followed by a quantization step if the system is lossy. Afterwards, the data is generally compressed further using a lossless compression technique.

This chapter first introduces the most popular lossless compression techniques, namely entropy coding in Section 2.1, run-length coding in Section 2.2, and dictionary methods in Section 2.3. A description of transform coding, a class of preprocessing techniques, follows in Section 2.4. The focus here is on image data, though similar techniques can be applied to audio and to volumetric data. Finally, in Section 2.5, the image compression standards JPEG and JPEG2000 are reviewed as practical applications of many of the presented concepts.

2.1 Entropy Coding

Entropy coding is a class of techniques which makes use of statistical properties of data to achieve compression. For an explanation of entropy coding, it is necessary to first introduce some terms from probability theory.

A (random) *experiment* is characterized by a set of possible *outcomes* $\Omega = \{\omega_i\}$, collectively called the *sample space*, each with an associated probability $P(\omega_i)$. A *random variable* $X : \Omega \to \mathbb{R}$ is a mapping from outcomes ω_i to numbers $x_i \in \mathbb{R}$. An experiment and an associated random variable can be either continuous or discrete. A continuous random variable has an uncountable number of possible outcomes; a discrete one only a countable, though possibly still infinite, number. In the following, only discrete random variables will be employed. Their output numbers x_i will usually be chosen from $\mathbb{N}$.

A *data source* S can now be modeled as a stochastic process in discrete time, defined by a sequence of random variables X_i. Each X_i corresponds to one data item generated by the source. Here, all X_i share a common set of possible outcomes $A = \{a_j\}$. A is called the *alphabet* of S. The individual a_j are called *symbols* or

letters.
In practice, a data set or *signal* can be seen as one possible realization of such a random process. The stochastic properties of the data source are generally not known, and can only be inferred approximately from the given realization.

2.1.1 Information and Entropy

For the purpose of data compression, it is useful to quantify the amount of *information* contained in a piece of data. The first rigorous definition of information was presented in an extremely influential paper by Shannon, published in two parts in 1948 [Sha48a, Sha48b].

First, it is useful to define the amount of information contained in the outcome of a random experiment. The *self-information* $I(\omega)$ of an event ω is defined as

$$I(\omega) = \log_b \frac{1}{P(\omega)} = -\log_b P(\omega) \tag{2.1}$$

In the terms of probability theory, the event ω can be either one outcome or a set of outcomes of a random variable. The definition of I seems somewhat arbitrary at first, but it at least makes intuitive sense: The occurrence of a certain event with a probability of 1 carries no information at all. The less likely an event, the larger the amount of information that is contained if it does occur. As the probability goes to 0, the information goes to infinity.
For two independent events ω_1, ω_2, the self-information is additive:

$$I(\omega_1, \omega_2) = I(\omega_1) + I(\omega_2) \tag{2.2}$$

This only holds if the two events are statistically independent. It does not hold if there is any positive or negative correlation between their occurrences.
The base of the logarithm can be chosen arbitrarily. In the context of data compression, however, it is almost always chosen as $b = 2$. In this case, the unit of information is *bits*. One bit in a computer stores exactly one bit of information.

Given the self-information of each outcome ω_i of a random variable X, we can compute its average or expected self-information. This quantity is called the *entropy*

$H(X)$ and defined as

$$H(X) = \sum_i P(\omega_i) I(\omega_i) = -\sum_i P(\omega_i) \log_b P(\omega_i) \tag{2.3}$$

The entropy is an important quantity because, as Shannon showed, it corresponds to the optimum any lossless compression scheme can achieve. That is, no lossless compression scheme can, on average, use less than $H(X)$ bits per symbol to encode the outcomes of X.

More interesting than the entropy of an individual random variable is the average information output by a data source $S = (X_1, X_2, \ldots)$. The average information per output symbol is called the *entropy rate*, though in the literature it is often simply called the entropy as well. To compute the entropy rate based on the previous definition of entropy, we would need *one* random variable describing the joint distribution of the X_i. Equivalently, we can employ the joint entropy

$$H(X_1, \ldots, X_n) = -\sum_{x_1} \ldots \sum_{x_n} P(x_1, \ldots, x_n) \log P(x_1, \ldots, x_n) \tag{2.4}$$

and define the entropy rate $H(S)$ as the limit of the normalized joint entropy

$$H(S) = \lim_{n \to \infty} \frac{1}{n} H(X_1, \ldots, X_n) \tag{2.5}$$

This is quite inconvenient to compute at best. In practice, the joint probabilities $P(x_1, \ldots, x_n)$ are usually not known, so it is not even possible in principle to compute the entropy rate. However, if the X_i are independent and identically distributed (iid), the joint entropy is equal to the sum of the individual entropies. In this case, the entropy rate of S is equal to the entropy of any X_i and can thus be calculated analogously to Eq. (2.3) as

$$H(S) = \sum_j P(a_j) I(a_j) = -\sum_j P(a_j) \log_b P(a_j) \tag{2.6}$$

It must be stressed that this computation of the entropy rate is valid only for iid random variables X_i. However, most real data sources are *not* iid—rather, their X_i are often heavily correlated. Thus, their entropy rate can *not* be computed in this

way. Eq. (2.6) can still be evaluated, but it only computes a quantity known as the *first-order entropy*. It is worth noting that in the literature, these terms are often not clearly differentiated, and so "entropy" is often written when actually only the first-order entropy is meant.

In practice, a preprocessing step is often used to decorrelate the data, so that the processed data is "approximately iid". Section 2.4 will show some examples of this. The following sections introduce some of the most common *entropy coders*, which exploit the first-order entropy of their input data to achieve compression.

2.1.2 Huffman Coding

Huffman coding is probably the earliest example of entropy coding. The algorithm was published in 1952 [Huf52] after being developed at the first ever course on information theory at MIT. It is a variable-length code (VLC) which assigns a binary *codeword* to each symbol in the input alphabet. The length of a symbol's codeword depends on the symbol's probability of occurrence. The more frequently a symbol occurs, the shorter its codeword will be. Huffman developed a method to find the optimal codewords for a given frequency distribution, in the sense that the total length of an encoded string will be minimal.

Prefix-Free Codes

An important property of VLCs is their unique decodability: Since a decoder does not know where one codeword ends and the next one begins, there might be multiple possible decodings. In a compression system, such ambiguity must of course be ruled out. A sufficient condition to ensure unique decodability of a VLC is that no codeword may be the prefix of another codeword. Such a code is called a *prefix-free code* or often, somewhat confusingly, just *prefix code*. An example of a prefix-free code is listed in Table 2.1. Any binary code can be conveniently visualized as a binary tree. Fig. 2.1 shows the binary tree corresponding to the code in Table 2.1. A codeword corresponds to a path in the tree, starting from the root. Each edge corresponds to one bit in the codeword and is labeled 0 or 1 accordingly. In a prefix-free code, codewords correspond only to the leaves of the tree. It is easy to see that such a code can be decoded uniquely: The decoder reads the input data bit by bit and

Table 2.1: Example of a prefix-free binary code.

symbol	codeword
0	000
1	1
2	011
3	0010
4	010
5	0011

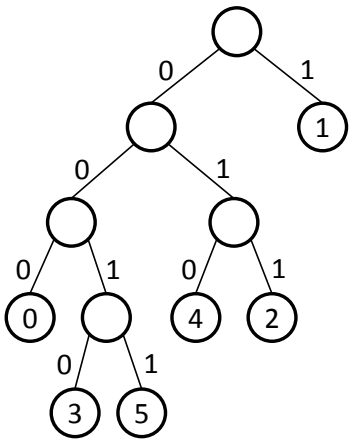

Figure 2.1: Binary tree corresponding to the code in Table 2.1. A codeword corresponds to a path from the root to a leaf node. Each edge corresponds to one bit of the codeword. The leaf nodes are labeled with the codewords they represent.

walks along the appropriate edges in the tree, starting at the root. When it reaches a leaf, a whole codeword has been read. The corresponding symbol is output, and the decoder starts again at the root of the tree.

It is worth noting that the prefix-free property is not a necessary condition for uniquely decodability. However, the decoding procedure for such a code would likely be much more complicated. More importantly, it can be shown that such a code can not provide superior compression performance. In other words, for each uniquely decodable code, there exists a prefix-free code with codewords of the same lengths. A proof for this claim can be found e.g. in Sayood's book [Say12].

Huffman Code Construction

Huffman's procedure for generating an optimal prefix code builds the binary tree (as in Fig. 2.1) in a bottom-up manner. It maintains a list of active nodes which is initialized with the tree's leaves, corresponding to the symbols in the input alphabet. While the list contains more than one element, it picks the two nodes with the lowest probabilities, removes them from the list, and creates a new node as their parent. The probability of the new node is set to the sum of the probabilities of the children, and the node is inserted into the list again. The algorithm finishes when there is only one node left in the list, corresponding to the root of the tree. The codeword for each symbol can be obtained by traversing the path from the root to that symbol. The proof that this procedure indeed results in optimal codewords is reproduced in many textbooks, e.g. Sayood's [Say12].

The Huffman code for a given input distribution is not uniquely defined. First, when choosing the two lowest-probability nodes, there may be ties. In this case, any two of the tied nodes may be chosen. Second, which of the two chosen nodes becomes the left child of the new node is arbitrary. Third, all codewords of equal length can be exchanged. In fact, the algorithm only fixes the codeword lengths. The actual codewords can be chosen freely as long as the resulting code is prefix-free. Obviously, none of these choices affect the compression rate. However, they may affect the efficiency of an implementation. In particular, the freedom in choosing codewords can be used to achieve compact storage of the Huffman table. We can use the following rules to assign codewords:

- Assign codewords in order of increasing length.
- The first codeword, i.e. the one with the smallest length, is the binary number 0 with the appropriate number of bits.
- Each subsequent codeword is generated by incrementing the previous codeword and appending 0 bits until the required length is reached.

It is easy to see that the resulting code is prefix-free. A Huffman code generated in this way is sometimes called a *canonical Huffman code*. Table 2.2 lists the canonical Huffman code for the example from Table 2.1. Fig. 2.2 shows the corresponding binary tree.

Table 2.2: Canonical Huffman code for the code from Table 2.1. Symbols are ordered by the length of the corresponding codeword. The codeword for the first symbol is the binary number zero with the appropriate number of bits. Each successive codeword is created by incrementing the previous one and appending zero bits as necessary.

symbol	codeword length	codeword
1	1	0
0	3	100
2	3	101
4	3	110
3	4	1110
5	4	1111

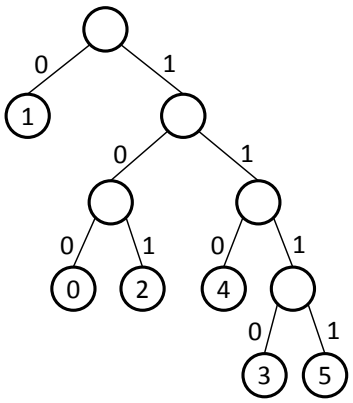

Figure 2.2: Binary tree corresponding to the Huffman code in Table 2.2.

All that needs to be stored and transmitted to the decoder is the sorted list of symbols and the number of codewords of each length. That is sufficient for the decoder to reconstruct the codewords in the same manner as the encoder assigned them. The actual codewords are not required, thus allowing for a more compact storage than an explicit mapping of symbols to codewords.

Encoding

Knowing how to construct the Huffman table, the actual encoding process is now straightforward. The first step is computing the symbol probabilities, i.e. counting

the number of occurrences of each symbol in the input data. With this information, the Huffman code can be constructed as described. Finally, each symbol is replaced by its assigned codeword, and the codewords are concatenated to produce the final output bit stream. Since the decoder also requires the Huffman code, it has to be stored along with the encoded data.

Decoding

The decoder receives as input the Huffman code and the bit stream of codewords. The decoding process is easiest to understand in the tree representation of the Huffman code. The decoder reads the input data bit by bit and each time moves along the corresponding edge in the tree, starting at the root. Whenever it reaches a leaf, a whole codeword has been processed. The decoder outputs the respective symbol and starts again at the root. This decoding scheme clearly works for any prefix code, not just for those created by Huffman's algorithm.

This straightforward tree-based implementation of Huffman decoding is not very efficient: Each bit of input entails following a pointer. One strategy for more efficient Huffman decoding is to build an over-complete lookup table which allows decoding each symbol in a single lookup. If the longest codeword has n bits, then the table requires 2^n entries. It is indexed by the next n bits of the input bit stream. Each entry contains the length of the first codeword within those n bits, as well as the corresponding symbol. Table 2.3 lists the corresponding lookup table for the Huffman code from Table 2.2. The decoding algorithm then becomes simple: Peek the next n bits from the bit stream, look up the symbol, and advance the position in the bit stream by the length of the codeword.

A drawback of this method is that the table can become extremely large and thus consume a lot of memory. It will also take a significant amount of time to build up such a large table, possibly negating any performance benefit during decoding. It is therefore often beneficial to take a hybrid approach: Build the lookup table only for codewords up to a certain length, e.g. 8 bits. For longer codewords, set the corresponding entry in the table to an invalid value, e.g. set the length entry to zero. When the decoder encounters such an entry, it reverts to the alternative implementation described above. Since the shortest codewords are also the most

Table 2.3: Decoding lookup table for the Huffman code in Table 2.2.

input / index	codeword length	symbol
0000	1	1
0001	1	1
0010	1	1
0011	1	1
0100	1	1
0101	1	1
0110	1	1
0111	1	1
1000	3	0
1001	3	0
1010	3	2
1011	3	2
1100	3	4
1101	3	4
1110	4	3
1111	4	5

common ones, this allows decoding most symbols in a single lookup without using excessive amounts of memory for the lookup table.

2.1.3 Golomb-Rice Coding

The Huffman coder described in the previous section explicitly determines the distribution of symbols and transmits it to the decoder. However, when the distribution is known beforehand, both these steps can be avoided, saving both computation time and storage space. Additionally, it is often possible to devise a simpler coding scheme which still produces optimal codewords, but is more computationally efficient than Huffman coding.

When the symbols follow a geometric distribution, i.e.

$$P(X = k) = (1 - p)^k p \quad \text{for some } p \in [0, 1]$$

with $k \in \mathbb{N}$, the so-called Golomb code [Gol66] produces optimal codewords. In a Golomb code, each symbol s is encoded as two numbers q ("quotient") and r ("remainder") according to

$$q = \left\lfloor \frac{s}{m} \right\rfloor \qquad \text{and} \qquad r = s - q \cdot m \qquad \text{for some } m \in \mathbb{N}^+.$$

For the code to be optimal, the parameter m must be chosen based on p according to

$$(1 - p)^m + (1 - p)^{m+1} \;\leq\; 1 \;<\; (1 - p)^{m-1} + (1 - p)^m.$$

Then, q is stored using a unary code, i.e. by q successive 1 bits followed by a single 0 bit. The remainder r is stored in truncated binary encoding: The first $2^b - m$ values are encoded in plain binary using $b-1$ bits with $b = \lceil \log_2 m \rceil$. The remaining values are encoded in binary as $r + 2^b - m$ using b bits.

If m is a power of two, this is equivalent to the standard b-bit binary encoding. Restricting the possible values of m to powers of two thus results in a simpler code called a Golomb-Rice code [RP71, Ric79]. Golomb-Rice codes are sometimes also referred to as Rice-Golomb codes or simply Rice codes, though the term Rice code also refers to a more sophisticated coder which employs a Golomb-Rice code as one component.

In practice, most data to be encoded is not geometrically distributed. Huffman coding will therefore usually outperform Golomb coding in terms of compression rate. However, transform coding (see Section 2.4) often results in data which is reasonably close to a geometric distribution. In such cases, Golomb coding can achieve similar compression rates as Huffman coding, but at a reduced computational load.

2.1.4 Arithmetic Coding

I have stated previously that Huffman coding generates optimal codewords. However, the output rate often does not equal the entropy of the data source. In fact,

Huffman coding achieves the source entropy only when the length of each codeword exactly corresponds to the self-information of the respective symbol. In other words, all symbol probabilities must be negative powers of two, so that the self-information of each symbol is a whole number of bits. In practice, Huffman coding often achieves bit rates quite close to the entropy nonetheless, so this is not always a significant limitation. However, there are some situations where Huffman coding does not perform well. For example, a large majority of the input symbols may have the same value. This is particularly common in a lossy compression context when many values are quantized to 0. In this case, the self-information of the symbol 0 is close to zero, but a Huffman coder must still allocate at least one bit to its codeword. Another example is the coding of a binary alphabet, that is, an alphabet with only two symbols. Huffman coding must allocate one bit to each codeword, and so can not achieve any compression.

To get around this fundamental limitation, a coding scheme must be used which does not use discrete codewords at all. Arithmetic coding is the most well-known example of such a scheme. The idea is due to Peter Elias. He developed it during the same course on information theory in which Huffman developed his coding method, but he never published it.

In arithmetic coding, the whole message is represented as a single, arbitrary-precision number in the unit interval $[0, 1)$. Since there are infinitely many real numbers in the unit interval, any message can be represented uniquely.

A straightforward algorithm to arithmetically encode a given input string is the following: Partition the unit interval $[0, 1)$ into sub-intervals and assign one sub-interval to each symbol in the input alphabet. The sizes of the sub-intervals are chosen to be proportional to the symbol frequencies. A string of symbols can be encoded by applying this process recursively: The sub-interval from the previous step is subdivided again using the same proportions. Fig. 2.3 shows an example of arithmetic coding using the symbol frequencies given in Table 2.4.
Given the final sub-interval and the symbol frequencies, the decoder can uniquely reconstruct the message by applying the same subdivision procedure. If the number of symbols in the message is known to the decoder, it is also sufficient to transmit any number within the final interval instead of the interval boundaries.

The algorithm as described so far is not very practical because it requires the

Table 2.4: Symbol frequencies for arithmetic coding example.

symbol	frequency
0	5
1	4
2	1

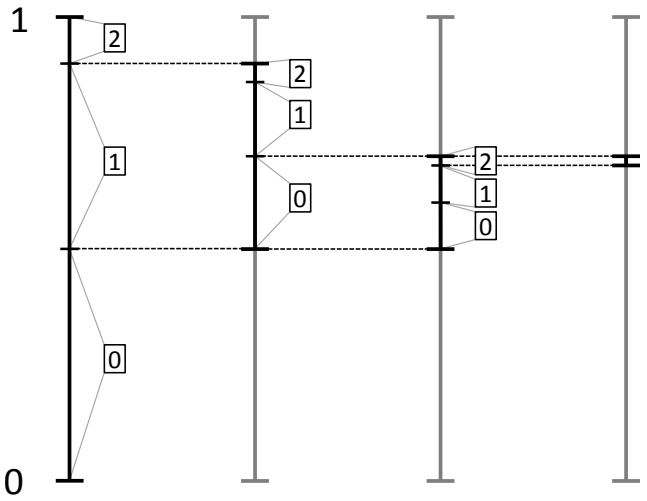

Figure 2.3: Example: Arithmetic coding of the string "102" using the frequencies given in Table 2.4. Any number within the final interval may be used to represent the encoded string.

use of arbitrary-precision arithmetic. Practical variations of the algorithm based on fixed-precision arithmetic were later developed independently by Pasco [Pas76] and Rissanen [Ris76, RL79]. The basic observation is that the current sub-interval is only ever restricted, never expanded. This means that as soon as the leading bit in the binary representation of the upper and of the lower bound is equal, it can never change. That happens when the current interval is fully contained in either the upper or lower half of the unit interval. For example, after encoding the first symbol in the example in Fig. 2.3, the current interval is contained in the upper half. Any number contained in the interval must start with this bit. Thus, it must be the first bit of the encoded message. This bit can be transmitted and is not required in the encoder anymore. At this point, the interval bounds can be *rescaled* to cover

only the remaining half of the unit interval, thus reducing the required precision. This is equivalent to shifting out the most significant bit of both interval bounds, and shifting in a 1 bit for the upper bound and a 0 bit for the lower bound. As a 1 bit was shifted out, this corresponds to scaling the upper half of the unit interval to cover the whole unit interval. This operation is called S_U. After encoding the second symbol in Fig. 2.3, the current interval is contained in the lower half of the remaining interval, and thus another rescaling is possible. Now a 0 bit was shifted out, so the lower half is scaled to cover the whole interval, called an S_L operation.
So far, if the active interval straddles the middle point, then no rescaling is possible. This means that the active interval can still become arbitrarily small. To handle this case as well, it is necessary to introduce another rescaling operation, S_M, which scales the middle half of the interval to cover the whole interval. This finally allows an implementation with fixed precision, but it creates a problem with encoding: So far, an S_U operation corresponds to a 1 bit and an S_L to a 0 bit. There seems to be no room left to encode an S_M. The solution is based on the following observation: An S_M operation followed by an S_U is equivalent to an S_U followed by an S_L. Similarly, $S_M \rightarrow S_L$ is equivalent to $S_L \rightarrow S_U$. Multiple successive occurrences of S_M can be resolved in a similar way: ${S_M}^n \rightarrow S_U$ is equivalent to ${S_U}^n \rightarrow S_L$. In practice, whenever an S_M occurs in the encoder, this is recorded, but no bit is transmitted. When an S_L or S_U finally occurs, the appropriate sequence of bits is transmitted based on the number of recorded S_M operations. Fig. 2.4 repeats the example from Fig. 2.3, but includes the rescaling operations. It also shows the interval bounds as binary numbers, demonstrating that the encoder only needs to keep track of a small window into the full binary numbers.

The number of bits required in the binary upper and lower bound depends on the frequency distribution of the input data. If the least likely symbol has a probability of $p_{\min}$, then it must be possible to uniquely identify $1/p_{\min}$ sub-intervals within the active interval. This requires $\lceil \log_2 1/p_{\min} \rceil$ bits. However, the active interval can be as small as one quarter of the distance between upper and lower bound. Therefore the bounds must be represented with at least $\lceil \log_2 1/p_{\min} \rceil + 2$ bits.

The decoding process is analogous to encoding. The decoder keeps track of the current lower and upper bounds. It mimics the rescaling operations of the encoder based on the bits of the encoded binary number.

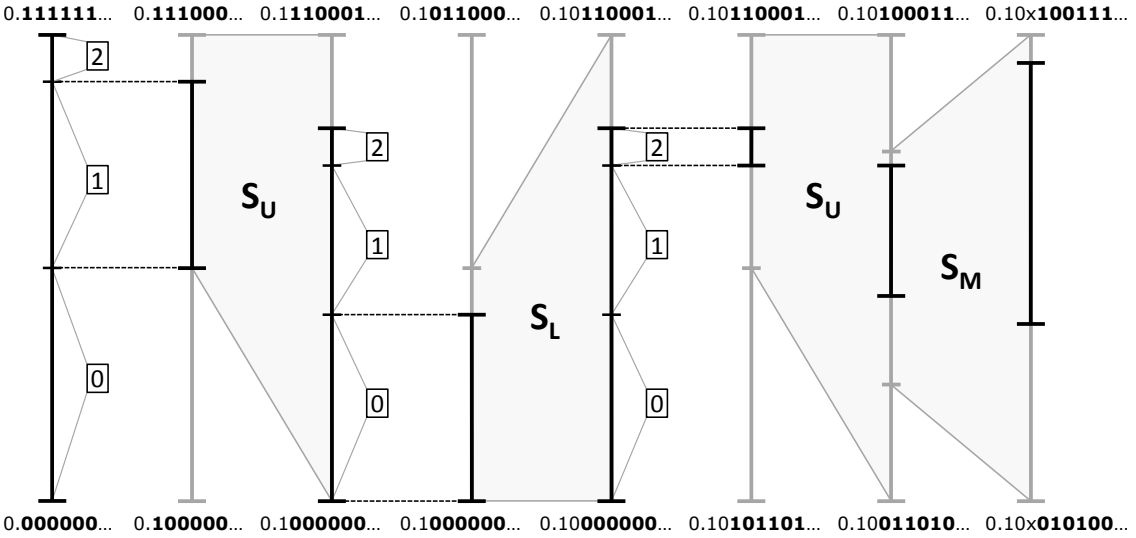

Figure 2.4: Example: Arithmetic coding of the string "102" with rescaling operations, using the frequencies given in Table 2.4. The current sub-interval is indicated in black, the currently represented interval in gray. The upper and lower interval bounds in binary are listed above and below; the encoder only needs to track the bits printed in bold. The rescaling operations ensure that the current sub-interval covers at least one quarter of the represented interval. Note that after the S_M operation, the next output bit is not known yet, indicated by an x in the binary numbers.

2.1.5 Adaptive Huffman and Arithmetic Coding

Both Huffman and arithmetic encoding as described so far require an extra analysis pass to compute the symbol frequencies. However, in some applications this may be impractical or at least inconvenient. Additionally, there is some storage overhead since the frequency information must be transmitted to the decoder. An alternative approach is to compute the frequency information during the coding process from previously seen data. This is called *adaptive* coding, as the coder dynamically adapts to the statistics of the data source. This approach has two main benefits: First, it gets rid of the frequency analysis pass in the encoder. Second, it removes the need for storing the frequency information, since the decoder can reconstruct it from the codeword stream itself. Additionally, it often achieves slightly better compression rates in practice. This is due to the fact that many practical data sources do not have constant statistical properties; in other words, the individual X_i making up a

data source S (see Section 2.1) are not identically distributed. If their distribution changes smoothly with i, then an adaptive coder will automatically adapt to the local statistical properties and thus better model the source.

For arithmetic coding, an adaptive scheme is easy to realize. The frequency table is initialized with a constant probability for each symbol. After a symbol has been encoded or decoded, its entry in the frequency table is incremented. This results in adjusted interval sizes for the following symbols.
For Huffman coding, the process is slightly more complicated. Changing a single frequency may change the shape of the Huffman tree, so the Huffman code would need to be re-built after each symbol. That would introduce a large performance penalty, so typically the code is re-built only periodically after some fixed number of symbols has been processed.

2.1.6 Comparison Between Huffman and Arithmetic Coding

Compared to Huffman coding, arithmetic coding has several benefits. The output rate of arithmetic coding asymptotically approaches the actual data entropy. Thus it achieves better compression than Huffman coding, where each codeword must be a whole number of bits. Whether the difference in compression rate is significant depends on the situation. In many cases, the output rates of Huffman and arithmetic coding are quite close. However, in some situations, the difference is crucial, e.g. when coding a binary alphabet.
Another benefit is that the probability table required for arithmetic coding is easy to update, facilitating adaptive coding as presented in the previous section. It is also easy to use and maintain multiple probability tables, particularly for binary alphabets.

The main drawback of arithmetic coding is its comparatively high computational complexity. The throughput of an arithmetic coder will generally be significantly lower than that of a Huffman or Golomb-Rice coder. Additionally, in the non-adaptive case, the storage overhead for the frequency information is slightly higher than with Huffman coding since Huffman coding does not require the actual symbol frequencies.

2.2 Run-Length Encoding

The coding techniques introduced so far have assumed that the individual symbols are all independent, and therefore have not attempted to exploit any correlations between symbols. The following sections cover techniques which do try to make use of correlations between symbols, and as such are orthogonal to the entropy coders discussed so far.

Run-length encoding (RLE) is likely the simplest compression algorithm which makes use of any correlations between symbols, or "patterns" in the data. It replaces repeated occurrences of the same symbol (a *run*) by a pair (s, n), where s specifies the symbol and n the number of occurrences. An individual occurrence of a symbol s is thus replaced by $(s, 1)$. This means that data which has few or no runs will be expanded rather than compressed. It is easy to see that for most types of data, RLE on its own is not very useful. However, it can sometimes be combined with other techniques in a productive way. An example is the JPEG image compression format described in Section 2.5.1.

2.3 Dictionary Techniques

Dictionary techniques try to find repeated sequences of symbols in the input data and replace them by a "back reference" to the previously encountered instance. While a great variety of different variants of this idea exist, almost all of them are based on one of two fundamental techniques introduced by Ziv and Lempel in 1977 [ZL77] and 1978 [ZL78], called LZ77 and LZ78. The initials are swapped in the acronyms; reportedly this originally happened by accident and stuck.

2.3.1 LZ77

LZ77 uses a sliding window around the current position in the input data. In this window, it searches for repetitions. The dictionary is thus implicitly defined by a fixed range of previously processed data. Fig. 2.5 shows an example of the process. The part of the window to the left of the current position corresponds to the dictionary and is called the search buffer. The right part is called the look-ahead buffer. The

encoder searches the longest prefix of the look-ahead data in the search buffer. It then outputs a triple (i, n, s), where i is the index of the longest match in the search buffer counting left from the current position, n is the length of the match, and s is the next symbol in the look-ahead buffer after the match. If no match is found, the output is simply $(0, 0, s)$. Explicitly specifying the next symbol s ensures progress even if a symbol is encountered which does not occur in the search buffer at all. After outputting a triple, the current position is incremented by $n+1$ and the sliding window moved accordingly.

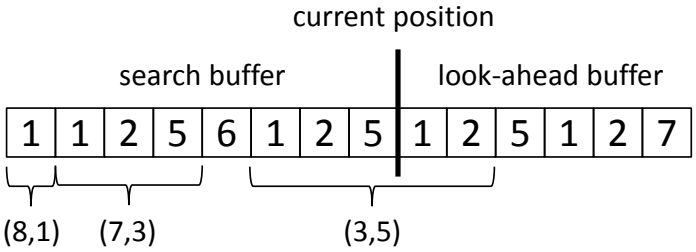

Figure 2.5: Example of LZ77 compression. The algorithm searches for prefixes of the look-ahead buffer within the search buffer. Three matches are found, labeled with their offset relative to the current position and their length. Note that the last match actually extends into the look-ahead buffer. In this case, the output would be $(3, 5, 7)$, consisting of the longest match and the following symbol. In practice, the buffers are much larger, typically at least on the order of kilobytes.

The third value of the triple is really only required when no match could be found. A common enhancement of LZ77 called LZSS [SS82] makes use of this. Instead of always outputting triples, it first outputs a single bit which indicates whether a match was found. If there was a match, the output is a pair (i, n), otherwise it is only the single symbol s.

2.3.2 LZ78

An implicit assumption in LZ77 is that repeating patterns occur close together. If a pattern is repeated, but the distance between repetitions is larger than the search buffer, then LZ77 will "miss" this repetition. In contrast, the LZ78 approach builds up an explicit dictionary of previously seen patterns. The encoder outputs pairs (i, s), where i is the index of the longest match in the dictionary, and s is the symbol

in the input data following the match. The concatenation of the dictionary entry at index i and the symbol s then becomes a new entry in the dictionary. Given the sequence of (i, s) pairs, the decoder can construct exactly the same dictionary as the encoder and thus correctly decode the data.

The idea behind LZ78 is similar to adaptive arithmetic coding in the sense that information about the input data is learned dynamically from previous data. No auxiliary information needs to be stored.

LZ78 can in principle capture repeating patterns of arbitrary length and arbitrary distance to each other. However, there are two problems: First, the memory required to store the dictionary grows continuously as new entries are added. In practice, the size of the dictionary has to be limited in some way. Second, the number of bits required to encode an index i into the dictionary also grows with the size of the dictionary. This means that there is a trade-off in choosing the size of the dictionary. A larger dictionary will potentially capture more and longer patterns, but will also produce a larger storage overhead for the indices.

One option to handle these problems in practice is to limit the dictionary to some fixed size. When that size has been reached, no new entries in the dictionary are created. The encoder then watches the compression rate of the output. If the compression rate worsens significantly, the dictionary is flushed so that new entries can be created.

Similar to the LZSS modification for LZ77, it is possible to avoid the explicit transmission of s in LZ78. The enhanced variant is due to Welch and called LZW [Wel84]. Instead of starting with an empty dictionary, the dictionary is initially filled with all symbols from the input alphabet. In this way, it is not possible that no match is found in the dictionary, and consequently it is sufficient to transmit only the indices i instead of the (i, s) pairs of LZ78.

2.3.3 Use of LZ algorithms in practice

The LZ algorithms and their variations have proven to be exceedingly useful and versatile in practice. In fact, almost all general-purpose compression tools in practical use are based on some variant of LZ compression. For example, the well-known DEFLATE algorithm [Deu96] is a combination of LZSS compression with subsequent

Huffman coding. It is the basis of the widely used `.zip` and `.gz` compression formats, and used as the final compression stage in the `.png` image format.

LZW is used e.g. in the `.gif` image format. Algorithms based on LZ78 are much less common in practice than LZ77-based ones. This is not due to any inherent disadvantage of the algorithm, but rather because of several patents on LZ78 and LZW. Though the relevant patents have now expired, LZ77 remains much more popular.

The LZ algorithms are also surprisingly versatile in the compression rate vs. speed trade-off. On the one hand, the Lempel-Ziv-Markov chain algorithm (LZMA) introduced by the 7-Zip compression tool is among the best known general-purpose compression algorithms. On the other hand, Lempel-Ziv-Oberhumer (LZO) [Obe11] and LZ4 [Col13] are LZ variants tuned for speed, and LZ4 in particular can achieve decompression speeds of multiple gigabytes per second on a single CPU core.

2.4 Transform Coding

The data compression techniques introduced so far, in particular entropy coding and dictionary techniques, make very few assumptions about the structure of the data. Entropy coding techniques only assume that some values occur more often than others. They treat each element individually without making use of any possible correlations. Dictionary techniques exploit repeating sequences of values and thus imply a 1D structure in the data. Both make use only of values being exactly equal, with no concept of similarity between values. In the terms of Stevens' theory of scales of measurement [Ste46], they are based on a *nominal* scale, allowing only to distinguish elements, but not to order them or to compute a degree of difference between them. However, when looking at a photographic image, neighboring pixels will often have *similar*, but rarely *equal* values. The same is true for the samples in many other kinds of numeric data such as volumetric CT or MRI data, flow fields, or audio data. It therefore makes sense to consider an *interval* scale, allowing the computation of differences between elements. When neighboring elements often have similar values, storing differences instead of values can result in better compression.

This section introduces a class of techniques collectively called *transform coding*. In contrast to the dictionary techniques which consider the input data a linear 1D

array, transform coding can easily be extended to exploit correlations in two or more dimensions. Here, the focus is on 2D image data, but similar techniques can be applied to audio data, volumetric data, etc.

Transform coding techniques, as the name implies, transform the input data to some other representation. This has two main objectives. The first is to decorrelate the data and thus compact most of the "content" into a few data elements. The remaining data elements can then be compressed to a small size. More formally, the transform should reduce the first-order entropy of the data, and therefore make entropy coding techniques more effective. The second objective is to decompose the data into components which have different interpretations. A lossy compression system can then selectively dismiss some parts which are considered less important, or store them at a lower fidelity.

The transforms themselves are usually invertible in the mathematical sense, so the original values can be fully reconstructed from the transform coefficients. However, when implemented in practice using floating-point arithmetic, rounding errors can occur and lead to information loss. This means that the transforms are not invertible anymore. In a lossy context, that is usually not a problem, as the loss of information in the following quantization step will eclipse any rounding errors during the transform. For lossless compression, however, care must be taken that the transform remains truly invertible. Typically this means that only integer arithmetic may be used. Luckily, reversible integer variations of many transforms exist.

2.4.1 Discrete Cosine Transform

The well-known Fourier transform, discovered in the 1820s by Joseph Fourier, allows describing any periodic function as a sum of sine waves of different frequencies. This provides a different way of looking at a function: Instead of a function value at each point in time or space, there now is an amplitude for each frequency of sine wave. The Fourier transform thus allows analyzing the frequency content of a given function. Accordingly, the representation as a sum of sine waves is often called the *frequency domain*, while the "standard" representation is called the *time domain* or the *spatial domain* depending on the application.

The discrete cosine transform (DCT) [ANR74] is a variant of the discrete Fourier

transform (DFT). In fact, it is equivalent to a DFT of appropriately padded and replicated data. While the DFT operates on complex numbers, the DCT operates on real numbers only. Another difference is that the DFT assumes a periodic signal. For most practical data, this results in an artificial "edge" between the last and the first sample. The DCT, on the other hand, is based on a mirrored periodic extension and so avoids this spurious edge. Overall, the DCT is therefore much better suited for many practical purposes including data compression.

It is worth noting for completeness' sake that there are actually eight different versions of the DCT, along with eight different discrete sine transform (DST) versions. The difference between them lies in different types of symmetric extension: The symmetry can be either even or odd at both the left and right boundary of the signal. Additionally, the points of symmetry can either lie exactly at the last sample, i.e. at 0 and $n-1$, or outside the signal by half a sample distance, i.e. at $-\frac{1}{2}$ and $n-\frac{1}{2}$. In the latter case, the last sample of the signal will be replicated by the symmetric extension. The remainder of this section only discusses the so-called type-II DCT, which implies even symmetry at both boundaries and points of symmetry between samples. The type-III DCT is the inverse of the type-II DCT.

The DCT represents a signal as a sum of cosine functions of different frequencies. It can be interpreted as a basis transform from the "standard basis" consisting of the unit vectors $(1, 0, 0\ldots)$, $(0, 1, 0, \ldots)$, ... to a basis which consists of discretely sampled cosine functions. Fig. 2.6 shows the DCT basis vectors $\mathbf{c}_i$, $0 \leq i < n$ for $n = 8$ samples, along with the underlying continuous cosine functions. The full transform can be written as a matrix $\mathbf{C} = [\mathbf{c}_{i,j}]$ consisting of the basis vectors as rows. In detail, the transform matrix is defined as

$$\mathbf{c}_{i,j} = s_i \cdot \cos \frac{(2j+1)i\pi}{2n} \quad \text{with } s_i = \begin{cases} \sqrt{\frac{1}{n}} & , i = 0 \\ \sqrt{\frac{2}{n}} & , \text{else} \end{cases} \tag{2.7}$$

The scaling factors s_i are chosen so that the L_2 norm of each basis vector is 1 and so the transform is orthonormal. The inverse transform can therefore be found by simply transposing the transform matrix.

The DCT can be extended to multiple dimensions by transforming along each dimension separately. For 2D data such as images, this means first transforming

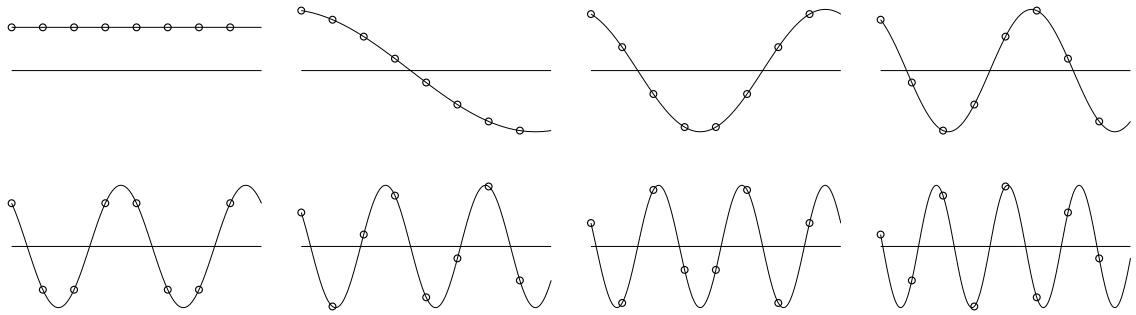

Figure 2.6: Discrete cosine transform basis vectors for $n = 8$.

each row of the image, then each column. The DCT is separable, so transforming the columns first will give the same result.

The matrix form of the DCT immediately suggests a simple $O(n^2)$ implementation based on a matrix-vector product. As the DCT is based on the DFT, there also exists an $O(n \log n)$ algorithm analogous to the fast Fourier transform [CT65] for computing the DCT. However, this still means that the required computation time grows more than linearly with the data size. This is very inconvenient in many applications. In practice, the DCT is therefore applied separately to fixed-size blocks of the input data. In image compression, a block size of 8×8 pixels is most common. Some video compression formats also allow other block sizes such as 4×4 or 16×16. Fig. 2.7 depicts the basis functions for a 2D 8×8 DCT. Fig. 2.8 shows an example of an image and its DCT coefficients. It is clear to see that typically only the coefficients in the upper left of each block have large values. These coefficients correspond to the low-pass information in the image; the top left coefficient in each block is simply a scaled average of all values in the block. This satisfies the first goal of transform coding: Most of the "content" of the image has been contracted into few coefficients. The second benefit stems from the fact that the human visual system is less sensitive to high-frequency variations. This means that the high-pass coefficients can be stored at a lower fidelity than the low-pass coefficients without significantly affecting the perceived image quality. We will see an application of this idea in Section 2.5.1, where the JPEG image compression standard is discussed.

The DCT has a few shortcomings with respect to data compression. One stems from the blockwise application of the transform. If the transform coefficients are

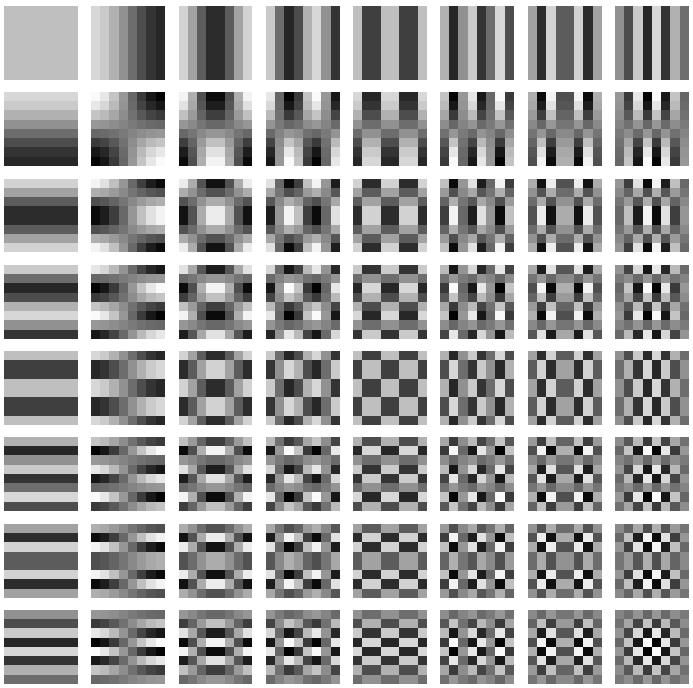

Figure 2.7: Basis functions of the 8×8 2D DCT, computed as the outer product of the 1D basis vectors in Fig. 2.6. Black corresponds to a value of -0.25, white to 0.25.

compressed in a lossy way, then the block structure can become visible in the reconstructed data. However, this is usually only a problem when a fairly low bit rate is used. Additionally, there are transforms derived from the DCT which avoid this effect: The lapped orthogonal transform (LOT) [MS89] and the lapped biorthogonal transform (LBT) [Mal98] employ overlapping blocks. This makes it possible to use basis functions which go to zero on the block boundaries. In turn, this avoids most visible blocking artifacts; even at low bit rates, a smooth image is reconstructed. The JPEG-XR image compression format [ITU09] uses an LBT.
Another potential shortcoming of the DCT is the use of floating-point arithmetic. Because of the inherent rounding errors, a truly lossless reconstruction is generally not possible in practice. However, it is possible to construct "DCT-like" transforms which use only integer arithmetic and which are invertible. Such transforms are used e.g. in JPEG-XR [ITU09] as well as in the H.264 video compression standard [ITU03].

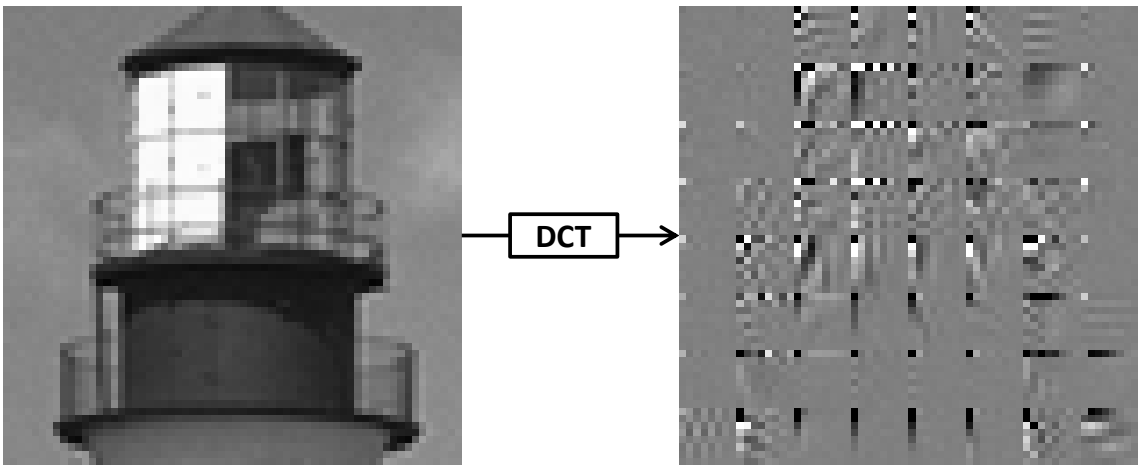

Figure 2.8: Blockwise 8×8 DCT applied to a grayscale cutout of image "kodim21" taken from the Kodak test image suite [FE99]. The block structure is clearly visible in the transformed image, with the largest coefficients typically appearing to the upper left of each block. The magnitude of the transform coefficients is exaggerated for clarity.

2.4.2 Discrete Wavelet Transform

Methods based on Fourier analysis, such as the DCT introduced in the previous section, give excellent localization in frequency space: They tell us exactly which frequencies occur in the data, which is very useful for data compression. However, they give no spatial localization: They do not tell us *where* in the signal these frequencies occur. Every DCT basis function has an impact on the whole domain (compare Figs. 2.6, 2.7). This has implications for data compression. For example, a single sharp edge in an image will require a large coefficient for some high-frequency basis function. However, this basis function will introduce high-frequency content in the whole image. Large values for some lower-frequency coefficients are required to cancel out these unwanted effects. This in turn impairs the compression rate.
This effect is one of the reasons why the DCT is applied in small blocks in practice. However, there is a trade-off involved, as smaller blocks make the compression of large homogeneous regions less effective. For this reason, some compression standards such as H.264 video compression [ITU03] allow varying block sizes, but this significantly complicates the compression process.

A different approach is to use a set of basis functions which have local support. This

circumvents the mentioned shortcoming of Fourier-based methods: A local feature will only influence a fairly small number of basis functions.
One option for such a set of local basis functions, and certainly the most popular one, is the multi-resolution analysis based on wavelets. The term "multi-resolution analysis" in the context of wavelets was introduced in the late 1980s by Stéphane Mallat [Mal89], though research on wavelets had been ongoing for several years before that. There is a large amount of mathematical literature regarding wavelets, their properties, and their construction. This thesis will only cover the basic idea behind the wavelet multi-resolution analysis and how the discrete wavelet transform (DWT) can be implemented in practice. For more mathematical background, the interested reader is referred to the book by Jensen and Cour-Harbo [JCH01] for an entry-level introduction, or to Mallat's book [Mal08] for a more extensive treatise.

Multi-Resolution Analysis

The idea behind the wavelet multi-resolution analysis is to build a basis out of translated and scaled versions of one underlying function called the *mother wavelet* ψ. The mother wavelet is non-zero only in a small region, leading to the locality properties. It is translated to cover the whole domain. It also covers only a small frequency band, and is scaled to cover higher or lower frequencies. The family of translated and scaled functions $\psi_{l,i}$ is generated from ψ according to

$$\psi_{l,i}(t) = \sqrt{2^l}\ \psi\left(2^l t - i\right), \qquad l, i \in \mathbb{Z}. \tag{2.8}$$

Incrementing l halves the width of the resulting function, which thus corresponds to a higher frequency band. Changing i moves the function along the x axis. The size of each step scales with the width of the function, defined by l. The normalization factor $\sqrt{2^l}$ is chosen so that the L_2 norm stays constant.

The mother wavelet can be chosen so that the $\psi_{l,i}$ are pairwise orthogonal and thus form a basis of some function space. Two functions f, g are called orthogonal if their inner product $\langle f, g\rangle$ is zero, i.e. $\langle f, g\rangle = \int f(x)g(x)dx = 0$. However, representing a function in this basis will generally require an infinite number of basis functions $\psi_{l,i}$: To represent a constant component, i.e. content of frequency zero, the wavelet must be infinitely scaled. To address this, it is necessary to introduce an additional

scaling function ϕ which complements the wavelet. It is scaled and translated in the same way as the mother wavelet. While the wavelet corresponds to some kind of difference between function values, the scaling functions represents an average. The scaling function thus covers all frequencies below some limit where the mother wavelet "takes over". This avoids the need to scale the mother wavelet infinitely to cover the lowest frequency content.

The interplay between wavelet and scaling function is easiest to understand with a concrete example of a wavelet and the corresponding scaling function. For the simplest and oldest wavelet, the Haar wavelet, the scaling function ϕ and wavelet ψ are defined as follows:

$$\phi(t) = \begin{cases} 1 & , 0 \leq t < 1 \\ 0 & , \text{else} \end{cases} \tag{2.9}$$

$$\psi(t) = \begin{cases} -1 & , 0 \leq t < \frac{1}{2} \\ 1 & , \frac{1}{2} \leq t < 1 \\ 0 & , \text{else} \end{cases} \tag{2.10}$$

Clearly, all the wavelet functions $\psi_{l,i}$ are orthogonal. Additionally, the scaling functions $\phi_{l,i}$ at a fixed level l are orthogonal. The $\phi_{l,i}$ are also orthogonal to the wavelet functions $\psi_{k,i}$, $k \geq l$ at the same and all finer levels.

Fig. 2.9 (left) schematically shows the decomposition of a signal into *low-pass* components corresponding to the scaling function and *high-pass* components corresponding to the wavelet. The signal s_0 can be represented by the translated scaling functions at some fine scale l_0. It can then be split into the low-pass or *approximation* part s_1 corresponding to the scaling functions at $l_1 = l_0 + 1$ and the high-pass or *detail* part d_1 corresponding to the wavelet at l_1. The coarser signal s_1 can again be decomposed into the even-coarser s_2 and the corresponding differences d_2, etc. The original signal can be reconstructed from the low-pass signal at some chosen scale and the sequence of high-pass parts up to the finest level, e.g. from $s_1 + d_1$ or from $s_3 + d_3 + d_2 + d_1$. Fig. 2.9 (right) shows one of the basis functions corresponding to each signal component. The remaining basis functions are obtained by translation.

This successive decomposition is the core of the wavelet multi-resolution analysis. The approximation sequence s_l, $l \geq 0$ represents the signal at ever-coarser resolutions. This property makes the wavelet transform interact very well with level-of-

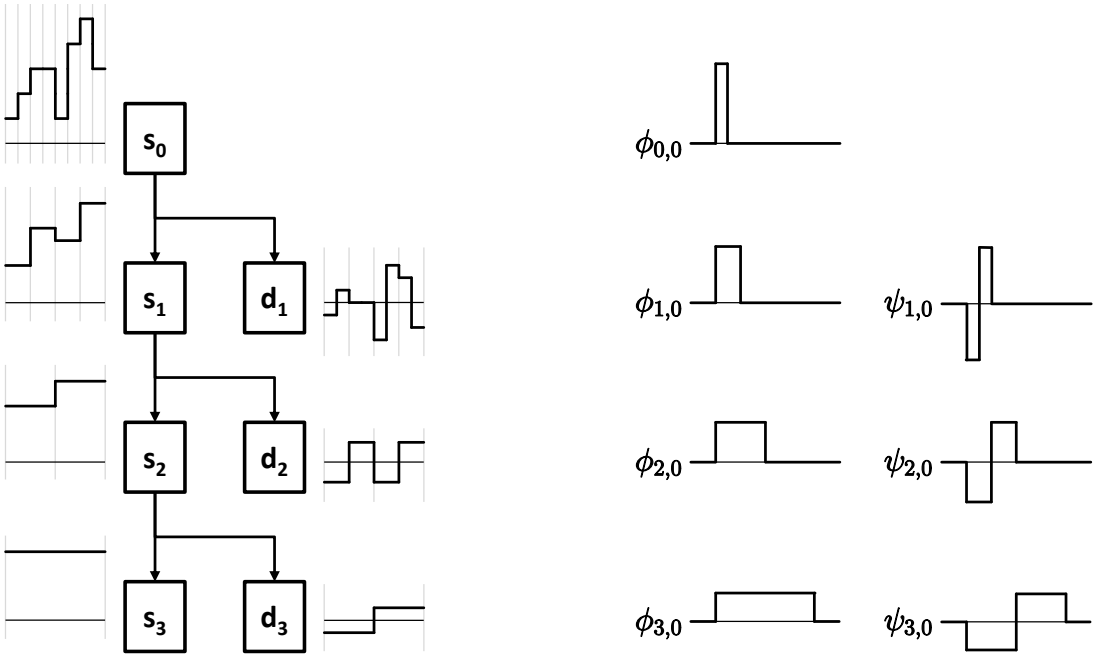

Figure 2.9: Multi-resolution wavelet decomposition. Left: Schematic representation of successively decomposing a signal into low-pass and high-pass components. A concrete example signal is also shown. The signal s_0 is split into the low-pass part s_1 and the high-pass part d_1. The coarser signal s_1 is further split into s_2 and d_2, and so on. Light gray lines in the example signals delineate the support of individual basis functions. Right: Some of the employed basis functions from the Haar wavelet.

detail methods, as the approximations created by the wavelet transform can be used directly as the coarser levels of detail. The detail sequence d_l, $l \geq 1$ provides the complementary high-frequency information within different frequency bands.

Filter Banks

The iterative level-by-level structure of the DWT sketched in Fig. 2.9 also leads to practical implementations. What is still missing is a way to split a given discrete signal into its approximation and detail parts. This can be achieved by applying a specially designed linear filter bank consisting of a low-pass filter corresponding to the scaling function and a high-pass filter corresponding to the wavelet. At first, this seems to double the amount of data. However, for the filter banks corresponding to

wavelet bases, it is sufficient to keep only every second element of the filter outputs. Only the even-indexed low-pass coefficients and the odd-indexed high-pass coefficients are required to reconstruct the input signal.
The inverse transform uses another set of filters. Zeros are inserted in place of the missing coefficients. Then the appropriate filters are applied, and their outputs are added. The forward transform is often called *analysis*, the inverse transform *synthesis*.

The coefficients for the filters corresponding to the Haar wavelet (Eqs. (2.9) and (2.10)) are listed in Table 2.5. It can be seen that the low-pass filter is simply an average of two neighboring values, while the high-pass filter gives half their difference.

Table 2.5: Filter coefficients for the Haar wavelet.

	−1	0	1
analysis low-pass		0.5	0.5
analysis high-pass	−0.5	0.5	
synthesis low-pass	1.0	1.0	
synthesis high-pass		1.0	−1.0

Table 2.6 shows a concrete example of the forward and inverse transform using the Haar wavelet. The input data is low-pass and high-pass filtered using the analysis filters. Only every second filter output is retained in the transformed data. In the inverse transform, the low-pass and high-pass output are separated again. Both are "upsampled" by inserting zeros and then filtered using the synthesis filters. The two arrays of filter outputs are added to once again obtain the original values.
In this example, we performed many redundant computations: In the forward transform, half of the computed filter outputs were discarded. Similarly, in the inverse transform, half of the filter inputs were zero and thus had no impact on the result. In practice, these computations are skipped for greater efficiency.

While the Haar wavelet is easy to understand, it does not achieve very good separation of low-frequency and high-frequency components. One way to look at this is in frequency space: The Haar scaling function amounts to a box filter, whose counterpart in frequency space is a sinc function. The sinc is a poor low-pass filter, so the

Table 2.6: Example of computing the filter-based DWT using the Haar wavelet.

input data		1.0	2.0	3.0	3.0	1.0	4.0	5.0	3.0	
low-pass filtered		1.5	2.5	3.0	2.0	2.5	4.5	4.0		
high-pass filtered			0.5	0.5	0.0	−1.0	1.5	0.5	−1.0	
transformed		1.5	3.0	2.5	4.0	0.5	0.0	1.5	−1.0	
upsampled low-pass	0.0	1.5	0.0	3.0	0.0	2.5	0.0	4.0	0.0	
filtered		1.5	1.5	3.0	3.0	2.5	2.5	4.0	4.0	
upsampled high-pass		0.0	0.5	0.0	0.0	0.0	1.5	0.0	−1.0	0.0
filtered		−0.5	0.5	0.0	0.0	−1.5	1.5	1.0	−1.0	
reconstructed		1.0	2.0	3.0	3.0	1.0	4.0	5.0	3.0	

Haar low-pass coefficients will actually contain a lot of high-frequency information and vice versa. This results in poor compression performance. Fortunately, many wavelet filter banks have been developed which provide better performance for various purposes. In the following, we will look at some other wavelets with more favorable properties. One popular class of wavelets are the Daubechies wavelets, named after their creator Ingrid Daubechies [Dau88]. The filter coefficients for the Daubechies4 wavelet are listed in Table 2.7. In contrast to the Haar wavelet, there exists no closed-form expression for the Daubechies4 scaling function and mother wavelet; they are defined only implicitly by recursive application of the filters. However, it can be shown that the same orthogonality relations between the basis functions still hold.

Table 2.7: Approximate filter coefficients for the Daubechies4 wavelet.

	−2	−1	0	1	2
analysis low-pass		0.3415064	0.5915064	0.1584937	−0.091506
analysis high-pass	−0.091506	−0.1584937	0.5915064	−0.3415064	
synthesis low-pass	−0.183012	0.3169873	1.183013	0.6830127	
synthesis high-pass		−0.6830127	1.183013	−0.3169873	−0.183012

It is worth noting that the Haar and Daubechies wavelets, as well as any other orthogonal wavelets, are fully defined by any one of the four filters. The analysis high-pass filter can be built from the analysis low-pass filter by reversing the order of the coefficients and negating all coefficients with an odd index. Similarly, the synthesis high-pass filter can be built from the analysis low-pass filter, up to a scaling factor, by negating all odd coefficients without changing their order. The synthesis low-pass filter can be constructed from the analysis high-pass filter in the same way.

Boundary Handling

In the DWT example in Table 2.6, some elements of the filtered sequences are missing. They could not be computed because the corresponding filter operation referenced elements outside of the input sequence. Conveniently, the missing values would have been discarded in the following subsampling step anyway. However, this only holds for the Haar wavelet. All other wavelets use filters of larger support, and thus some of the "uncomputable" elements will be required. Thus, it becomes necessary to somehow substitute the missing filter inputs. The simplest way is to assume all out-of-bounds values to be equal to zero, or to the last existing element. However, this will result in additional non-zero filter outputs which must be retained to ensure correct reconstruction. This is obviously very undesirable during data compression. A different option to supply the missing values is periodic extension. This will result in a periodic filter output and thus avoids the need to retain any extra values. However, it will result in an artificial edge between the last and the first signal value where instantiations of the signal are joined. This is also unfavorable for compression. It would be better if we could use symmetric extension like it is done in the DCT. For this to work, the basis functions and thus the filters need to be symmetric. Unfortunately, it can be shown that orthogonal wavelets always have an even number of non-zero filter coefficients and so can never be symmetric. To create symmetric wavelets, it is necessary to relax the orthogonality constraint, i.e. the simple relation between the low-pass and the high-pass filter. The resulting wavelets are called *biorthogonal*, as only a mutual orthogonality between the analysis and the synthesis filters remains. The most popular biorthogonal wavelets are taken from a family of wavelets called CDF after its creators Cohen, Daubechies, and Feauveau [CDF92].

In particular, the CDF 5/3 and CDF 9/7 wavelets are commonly used, e.g. in the JPEG2000 standard (see Section 2.5.2). The numbers refer to the number of filter coefficients in the analysis low-pass and high-pass filters. The coefficients for the CDF 5/3 and CDF 9/7 filters are listed in Tables 2.8 and 2.9, respectively. Both are symmetric and thus allow for boundary handling via symmetric extension. This is one of the reasons why these wavelets are among the most popular for data compression.

Table 2.8: Filter coefficients for the CDF 5/3 wavelet.

	0	±1	±2
analysis low-pass	0.75	0.25	−0.125
analysis high-pass	0.5	−0.25	
synthesis low-pass	1.0	0.5	
synthesis high-pass	1.5	−0.5	−0.25

Table 2.9: Approximate filter coefficients for the CDF 9/7 wavelet.

	0	±1	±2	±3	±4
analysis low-pass	0.6029490	0.2668641	−0.0782233	−0.0168641	0.0267488
analysis high-pass	0.5575435	−0.2956359	−0.0287718	0.0456359	
synthesis low-pass	1.1150871	0.5912718	−0.0575435	−0.0912718	
synthesis high-pass	1.2058980	−0.5337281	−0.1564465	0.0337282	0.0534975

Multiple Dimensions

So far, we have only looked at the DWT as a one-dimensional transform. In multiple dimensions, the DWT can be applied to each dimension individually, similar to the DCT. The DWT is separable in the sense that, apart from possible rounding errors, it does not matter in which order the individual 1D DWTs are applied. An example of a 2D DWT applied to an image is shown in Fig. 2.10. The image quadrant labeled “LL” is the low-pass part in both dimensions, and therefore is just a lower-resolution version of the input image. The “LH” and “HL” quadrants contain high-pass information

in one dimension, i.e. mostly horizontal and vertical edges, respectively. The "HH" band contains high-frequency content in both dimensions.

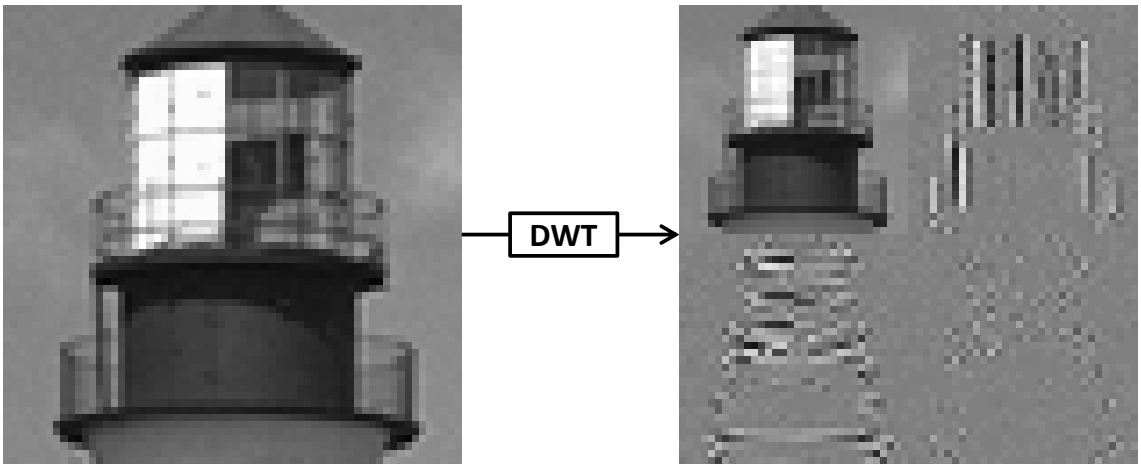

Figure 2.10: One-level DWT using the CDF 9/7 wavelet applied to a grayscale cutout of image "kodim21" taken from the Kodak test image suite [FE99]. The image is decomposed into four subbands named after their low-/high-pass content: "LL" (upper left), "HL" (upper right), "LH" (lower left), and "HH" (lower right). The magnitude of the coefficients in the three high-pass subbands is exaggerated for clarity.

Normalization

It is worth noting that the scaling of the analysis filter coefficients can in practice be chosen arbitrarily: If all the analysis low-pass coefficients are multiplied by some constant $a > 0$, then the synthesis low-pass coefficients must be divided by a so that synthesis will remain the inverse of analysis. The same is valid for the high-pass coefficients. There are multiple conventions for scaling or "normalizing" the wavelet filter coefficients.

In math textbooks, the wavelet filters are most commonly normalized to be orthonormal, i.e. to preserve energy (compare the normalization factor in Eq. (2.8)). This means that the sum of the squares of the signal values will not be changed by the transform. To achieve this normalization, all analysis filter values listed in Tables 2.5, 2.7, 2.8, and 2.9 must be multiplied by $\sqrt{2}$, the synthesis values accordingly divided by $\sqrt{2}$. A side-effect of this normalization is that no additional scaling is

necessary when constructing the synthesis filter coefficients from the analysis filter coefficients.

For multi-resolution applications, however, it is typically most useful to normalize the analysis filters for unit *nominal gain*. The *(nominal) DC gain* g_{dc} and *(nominal) Nyquist gain* g_{nyq} of a filter kernel $\mathbf{x} = (x_{-n}, \ldots, x_0, \ldots, x_n)$ are defined as

$$g_{\mathrm{dc}}(\mathbf{x}) = \sum_{i=-n}^{n} x_i \quad , \qquad g_{\mathrm{nyq}}(\mathbf{x}) = \sum_{i=-n}^{n} (-1)^i x_i \tag{2.11}$$

The low-pass analysis filter is normalized for unit DC gain, the high-pass filter for unit Nyquist gain. This is the convention used here; the same convention is used in the JPEG2000 image compression format [TM01]. Note that the DC gain of the high-pass filter and the Nyquist gain of the low-pass filter are always zero.
For a constant signal, the low-pass coefficients will then be equal to the signal value, while the high-pass coefficients will be zero. Conversely, for a signal alternating between 1 and -1, all high-pass coefficients will be equal to 1, while the low-pass coefficients will be zero. This is a useful property for multi-resolution image applications, as the low-pass coefficients can be interpreted directly as a lower-resolution version of the original image. Additionally, the range of values is expected to be preserved, which is often convenient. However, this is not guaranteed; some combinations of input values may still result in a range expansion.

Lifting

The filter bank implementation of the DWT is based on floating-point arithmetic. As such, it is subject to rounding errors and is thus not admissible for lossless compression. Fortunately, there is a second, completely different way to implement wavelets called the *lifting scheme* [Swe98]. Lifting allows for an integer implementation of the DWT, and has several other benefits: First, a lifting implementation needs only about half as many arithmetic operations compared to an implementation based on filter banks. Second, lifting makes the design of new wavelets straightforward, even on domains which are not Cartesian grids. Due to this property, the lifting-based DWT is also called the *second-generation wavelet transform*. For data compression, however, the most important factor is that it allows an integer-based implementa-

tion. This enables a truly reversible transform in practice, which is a prerequisite for lossless compression.

In the lifting scheme, the input data $[x_i]$ is first split into its even and odd halves:

$$s'_i := x_{2i} \quad , \qquad d'_i := x_{2i+1} \tag{2.12}$$

Here, the even part is called s for "signal", the odd part d for "detail" because their values are updated by the transformation process to correspond to the low-pass and high-pass part, respectively. The transform is performed in a number of so-called *lifting steps*. Each step either changes the odd elements based on their even neighbors, or changes the even elements based on their odd neighbors. The former is commonly called a *prediction* step, the latter an *update* step. Changing a data element is restricted to adding or subtracting a weighted sum of the neighboring coefficients. Finally, an optional *scaling* step may be applied. Fig. 2.11 shows the schematic structure of the lifting scheme including one prediction (P) and one update (U) operation.

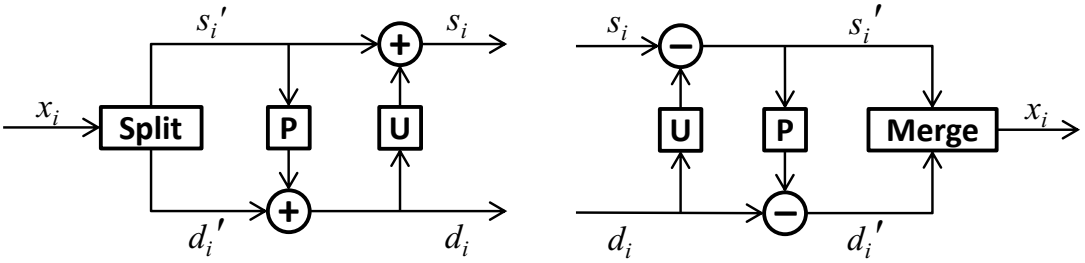

Figure 2.11: Schematic structure of the lifting scheme with one prediction (P) and one update (U) step. Left: Forward transform. Right: Inverse transform obtained by reversing the order of the prediction and update steps and exchanging addition with subtraction.

Every transform based on the lifting scheme is trivially invertible. The inverse transform can be found by simply reversing the order of the lifting steps and swapping pluses and minuses, as shown in Fig. 2.11 right. It is easy to see that this exactly reverts all changes from the forward transform.

For example, the CDF 5/3 wavelet (filter coefficients in Table 2.8) can be repre-

sented by one prediction and one update step:

$$\begin{aligned} d_i &= d'_i - \frac{s'_i + s'_{i+1}}{2} \\ s_i &= s'_i + \frac{d_{i-1} + d_i}{4} \end{aligned} \tag{2.13}$$

It can be easily verified that, up to a scaling factor, this corresponds to the CDF 5/3 filters listed in Table 2.8. Table 2.10 shows an example of applying the CDF 5/3 wavelet using lifting steps. The example demonstrates that the lifting steps can be performed in-place, in contrast to the filter-based DWT implementation. However, this results in interleaved low-pass and high-pass coefficients. If a subband order as in Fig. 2.10 is desired, then an additional reordering step is necessary.

Table 2.10: Example of computing the lifting-based DWT in-place using the CDF 5/3 wavelet. Gray values are created by mirrored extension. The signal values are updated in-place; updated values are indicated in bold. The prediction and update steps produce the low-pass and high-pass values in interleaved order.

input data		1.0	2.0	3.0	3.0	1.0	4.0	5.0	3.0	5.0
predict	0.0	1.0	**0.0**	3.0	**1.0**	1.0	**1.0**	5.0	**−2.0**	
update	0.0	**1.0**	0.0	**3.25**	1.0	**1.5**	1.0	**4.75**	−2.0	
undo update		**1.0**	0.0	**3.0**	1.0	**1.0**	1.0	**5.0**	−2.0	5.0
undo predict		1.0	**2.0**	3.0	**3.0**	1.0	**4.0**	5.0	**3.0**	

Every wavelet filter bank can be decomposed into a sequence of lifting steps [DS98]. For example, the CDF 9/7 filters can be expressed by the four lifting steps

$$\begin{aligned} d''_i &= d'_i + \alpha(s'_i + s'_{i+1}) \\ s''_i &= s'_i + \beta(d''_{i-1} + d''_i) \\ d_i &= d''_i + \gamma(s''_i + s''_{i+1}) \\ s_i &= s''_i + \delta(d_{i-1} + d_i) \end{aligned} \tag{2.14}$$

where $\alpha \approx -1.58613432$, $\beta \approx -0.05298012$, $\gamma \approx 0.88291108$, $\delta \approx 0.44350685$. Again,

it is straightforward to verify that, up to a scaling factor, this is equivalent to the CDF 9/7 filters listed in Table 2.9.

Integer Implementation

So far, the lifting steps still make use of floating-point numbers: Even if the inputs are integers, the outputs will generally not be whole numbers. By introducing appropriate rounding in the prediction and update operations, the output can be forced to integers. Somewhat surprisingly, this can be done while still preserving the invertibility of the transform. For the CDF 5/3 wavelet, the lifting steps (Eq. (2.13)) can be adapted as follows:

$$\begin{aligned} d_i &= d'_i - \left\lfloor \frac{s'_i + s'_{i+1}}{2} \right\rfloor \\ s_i &= s'_i + \left\lfloor \frac{d_{i-1} + d_i}{4} \right\rfloor \end{aligned} \tag{2.15}$$

This transform is often used for lossless compression, e.g. in JPEG2000 (see Section 2.5.2). Similarly, an integer version of the CDF 9/7 filters can be constructed. However, this is not commonly used for the following reason: The rounding operations effectively change the basis functions. Thus, the carefully constructed properties of the wavelet basis are slightly deteriorated. Since the lifting implementation of the CDF 9/7 wavelet requires four lifting steps, there are more rounding operations and thus a larger deterioration. This negates any advantage of the larger filters.

2.4.3 Color Space Transforms

Apart from the two spatial dimensions which the DCT and DWT exploit, images also have a color dimension. Digital images typically use the color channels red, green, and blue (RGB), representing different wavelengths of light, to encode color. So there are only three samples along this third dimension. This is sufficient because the human eye can only differentiate between three ranges of wavelengths, which roughly correspond to the RGB channels.

There is typically a significant amount of correlation between the RGB channels. Fig. 2.12 top and middle shows an image and its red, green, and blue channels

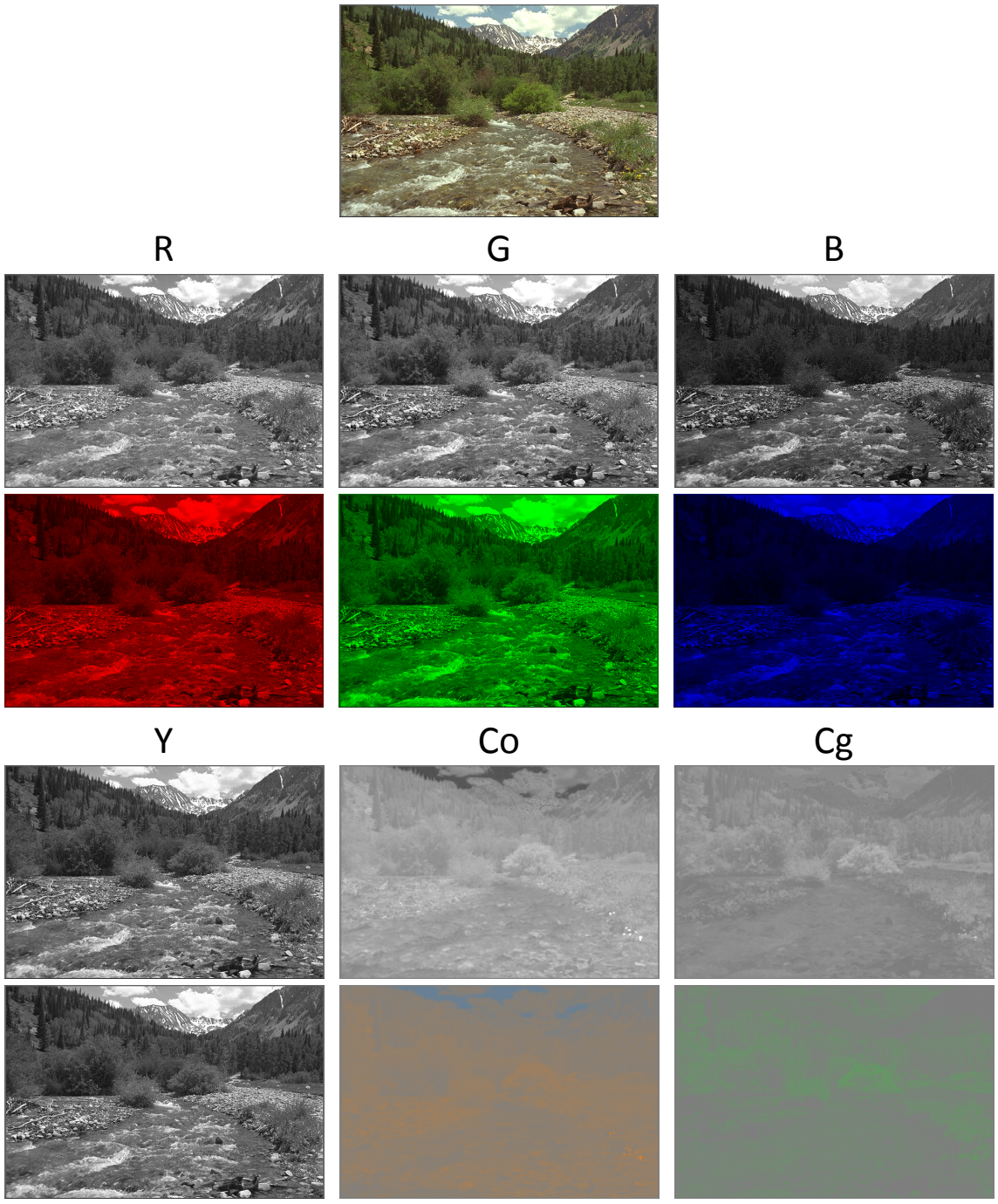

Figure 2.12: Top: Image “kodim13” taken from the Kodak test image suite [FE99]. Middle: RGB channels shown separately. Bottom: YCoCg channels shown separately.

displayed separately. Clearly, the individual color channels look much alike. This correlation can be exploited by converting the RGB values to another color space. Many different color spaces exist for various purposes. Most useful for image compression are color spaces which separate the image into *luma* (brightness) and *chroma* (color) components. Some of the most common examples are YCbCr ("luma, chroma blue, chroma red") and YCoCg ("luma, chroma orange, chroma green") [MS03a]. The transformation between RGB and YCoCg is defined as follows:

$$\begin{pmatrix} Y \\ Co \\ Cg \end{pmatrix} = \begin{pmatrix} 1/4 & 1/2 & 1/4 \\ 1/2 & 0 & -1/2 \\ -1/4 & 1/2 & -1/4 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \quad , \quad \begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 1 & -1 \\ 1 & 0 & 1 \\ 1 & -1 & -1 \end{pmatrix} \begin{pmatrix} Y \\ Co \\ Cg \end{pmatrix} \tag{2.16}$$

YCoCg has been shown to provide very good decorrelation for many images. It is also computationally very simple, requiring only integer additions and shifts. A downside is that the transformation defined above is not reversible when using integer values. This means that it can not be used for lossless compression. However, the transform can be made reversible with a simple change which basically amounts to doubling the values of Co and Cg in Eq. (2.16) before rounding [MS03b]. The modified transform is known as YCoCg-R. Note that this increases the dynamic range of the Co and Cg channels by one bit. That is, for a 3×8 bit RGB image, $8 + 9 + 9$ bits per pixel are required to store its YCoCg-R representation.

Fig. 2.12 bottom shows the Y, Co, and Cg channels of the image in Fig. 2.12 top. While some correlation between the channels is still visible, the amount is clearly much smaller than with the RGB channels. Most of the energy has been concentrated into the luma channel; the chroma channels have much lower contrast and thus are more amenable to compression. An additional benefit of this color space transform stems from the fact that the human eye has a much lower resolution for color than for brightness. This means that the color information can be stored at a lower spatial resolution with only a small effect on the visual quality.

2.5 Image Compression in Practice

The previous sections have introduced many basic data compression techniques. To see how they can be combined in practice to form effective data compression systems, we will now look at two popular standards for image compression, JPEG and JPEG2000. This section will focus on the high-level picture and the justification for the choices made in JPEG and JPEG2000. It will not cover all the details necessary for an actual implementation. The books by Pennebaker and Mitchell [PM93] and by Taubman and Marcellin [TM01] cover the JPEG and JPEG2000 standards in greater detail.

2.5.1 JPEG

The JPEG standard [ITU92] was first published in 1992 as the first ever standard for the compression of photographic images. Today, it is still the most popular format for this application, even though many alternatives exist. This section explains the most common way in which JPEG compression is applied. It does not attempt to cover all aspects of the JPEG standard.
JPEG processes an input image in several stages, most of which have been introduced in similar form in the previous sections. The stages of JPEG encoding are the following:

1. **Color space transform.** The first step in JPEG encoding is transforming the input RGB image into the YCbCr color space. This is a "luma + 2× chroma" color space similar to the YCoCg color space introduced in Section 2.4.3. Strictly speaking, the color space transform is not part of the JPEG standard; JPEG just assumes that its input is given in the YCbCr color space.
2. **Chroma subsampling.** As an optional second step, the resolution of the two chroma channels can be reduced. The rationale for this is that the human visual system is less sensitive to color than to brightness. Most commonly, the resolution of the chroma channels is halved in each dimension. This is called 4:2:0 sampling in JPEG. The first number refers to the number of reference luma samples in one row of pixels. The other two numbers indicate the number of chroma samples in two rows of pixels—in this case, 2 in the first row and 0 in

the second, resulting in a factor 2 reduction in each dimension. Other possible chroma samplings are 4:4:4 (no subsampling), 4:2:2 (factor 2 subsampling in x direction, no subsampling in y direction), or 4:1:0 (factor 4 in x, factor 2 in y). In the following steps, each channel is processed separately.

3. **DCT.** The data is now split into blocks of 8×8 samples. If chroma subsampling was applied, then luma and chroma blocks cover different-sized spatial regions. Each block is transformed using the DCT, described in Section 2.4.1. This results in a structure similar to the one shown in Fig. 2.8 where most of the information is contracted into the top left coefficient in each block. As this coefficient represents the average value in the block, it is called the DC coefficient, after the term "direct current". All other coefficients are called AC for "alternating current".

4. **Quantization.** The DCT coefficients are quantized using a uniform scalar quantizer. This is the stage where most of the information loss occurs. Different JPEG quality levels are realized by choosing different quantization step sizes. A floating-point coefficient x_i is quantized to an integer c_i according to $c_i = \text{round}(x_i/\Delta)$, where the quantization step Δ is an integer between 1 and 255. The quantization step can be chosen differently for each of the 8×8 coefficients in a block. Typically, the quantization steps get larger toward the lower right of the block, so that higher frequencies are represented more coarsely than lower frequencies.

5. **Coding.** Finally, the quantized coefficients are encoded using a somewhat peculiar scheme. The DC and AC coefficients are handled separately. Due to the quantization, many of the AC coefficients are expected to be zero. A run-length coding step makes use of this. The AC coefficients are arranged along a zig-zag order as shown in Fig. 2.13. This roughly orders the coefficients by frequency. The 63 coefficients are replaced by a variable-length list of pairs (z_k, v_k) with one entry for each non-zero coefficient. The z_k value is the number of zeros that preceded the non-zero coefficient in the input stream, v_k is the value of the coefficient. An obvious final step would be entropy coding of the remaining values. However, the number of distinct values can be quite large. To reduce the number of input symbols to the entropy coder and thus achieve a

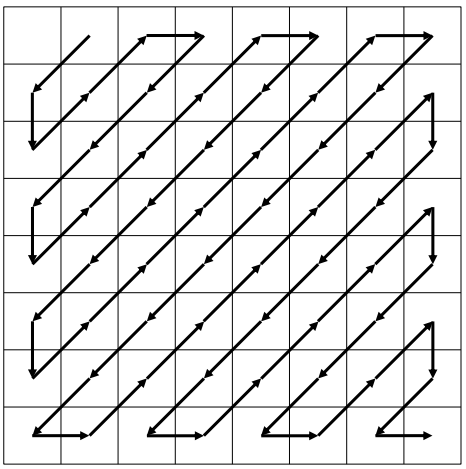

Figure 2.13: Zig-zag order used in JPEG for the encoding of AC coefficients.

simpler decoder, JPEG represents the coefficient values v_k in a so-called category code. Each v_k is replaced by a pair (c_k, u_k), where c_k is the length of the binary representation of v_k and u_k is this c_k-bit binary representation. This results in a list of triples (z_k, c_k, u_k). The z_k and c_k are appended and used as the input symbols to a Huffman coder. Both are limited to 4 bits each, so this effectively limits the number of distinct symbols the Huffman coder has to handle. The u_k are not further encoded.

The DC coefficients are averages of the 8×8 blocks, and so all DC coefficients together can be seen as a low-resolution version of the original image. For many images, there will still be correlation between neighboring DC values. To exploit this correlation, JPEG applies another prediction step before the final encoding. All DC values are arranged in scan-line order, and each is predicted by its predecessor. In this way, except for the very first value, only differences need to be stored, which tend to be much smaller. The differences are converted to a category code similar to the AC values. Again, the length values c_k are Huffman encoded, while the amplitudes u_k are stored directly.

The JPEG standard has been extremely successful. More than two decades after its publication, it remains by far the most widely used format for the compression of photographic images. However, it does have several shortcomings. One stems from the blockwise application of the DCT. At very low bit rates, typically under 1 bit per

pixel, the block structure becomes visible. This results in very noticeable artifacts and correspondingly poor visual quality. The second major shortcoming of JPEG is that it is fairly inflexible. That is, it is generally not possible to partially decode a compressed JPEG image, e.g. to extract only a low-resolution or low-fidelity version of the image, without decoding the full image first. This lack of flexibility becomes more apparent when comparing JPEG to JPEG2000, which is discussed in the next section.

2.5.2 JPEG2000

JPEG2000 [ITU02] was developed with the intention of replacing JPEG as the standard way to compress and store digital images. It is based on a DWT instead of JPEG's blockwise DCT and thus avoids the blocking artifacts at low bit rates. It also offers many improvements in scalability. A JPEG2000-compressed image can be decoded incrementally in resolution or in quality. That is, it is possible to extract a lower-resolution or lower-quality image from a JPEG2000 bit stream which also contains the high-resolution and high-quality data. It is also possible to decode only some spatial regions of an image, or only individual image components, i.e. color channels. The scalability modes can also be mixed. The resolution scalability is achieved naturally through the multiscale nature of the DWT. The quality and spatial scalability are due to JPEG2000's sophisticated entropy coding stage which will be described later in this section.
JPEG2000 also allows lossless compression. This feature can be combined with the scalability options, so that a single JPEG2000 image can offer a progression from lossy to lossless reconstruction.
In addition to the scalability features, JPEG2000 was supposed to achieve improved image quality at the same compression rate compared to JPEG. However, whether this goal was fully reached is questionable. When image quality is measured in terms of root-mean-square (RMS) error, JPEG2000 consistently outperforms JPEG by a large margin. However, the RMS error is a fairly poor indicator of visual quality. With more sophisticated metrics which try to model visual quality as experienced by human observers, the benefit of JPEG2000 over JPEG is less clear. In fact, the consensus seems to be that JPEG2000 consistently achieves superior quality compared

to JPEG only at low to medium bit rates; at higher bit rates, often JPEG actually performs slightly better [ECW04]. This is one of the reasons why JPEG2000 has not replaced JPEG in practice, along with the higher implementation complexity and lower compression/decompression throughput. So while the JPEG2000 standard can be considered a failure in the sense that is has not been widely adopted, many of the employed concepts and methods are interesting and useful. The following paragraphs describe the major components of a JPEG2000 compressor.

Color Space Transform

Similar to JPEG, JPEG2000 encoding starts with a color space transform. JPEG2000 defines two color transforms. One is reversible, i.e. based on integer arithmetic, and applicable for the lossless compression mode. The other is based on floating-point arithmetic and thus irreversible because of rounding errors, but results in slightly better compression rates.

DWT

The second step is applying a multi-level DWT; typically, about five DWT levels are used. Again, two different transforms are defined: One is based on an integer version of the CDF 5/3 wavelet, and suitable for lossless compression. The second uses the CDF 9/7 wavelet and is not reversible, but gives better compression rates.

Quantization

The DWT coefficients are quantized using a uniform scalar dead-zone quantizer where the zero bin is twice as large as any other bin. A floating-point coefficient x_i is quantized to an integer c_i according to $c_i = \text{sign}(x_i) \lfloor |x_i|/\Delta \rfloor$. Due to the way the DWT coefficients are normalized in JPEG2000 (see Table 2.9), the quantization step needs to be adjusted for different DWT levels. Compared to an orthonormal transform, the analysis coefficients are scaled by a factor of $1/\sqrt{2}$, so the quantization steps needs to be scaled by the same factor per 1D DWT. For each successive 2D DWT level, the quantization step should thus be halved.
For lossless compression, the quantization step is set to 1 for all levels. Since the

DWT coefficients are already integers in this case, quantization is thereby effectively skipped.

Entropy Coding

JPEG2000 uses a sophisticated entropy coding scheme called *embedded block coding with optimal truncation* (EBCOT). EBCOT individually encodes blocks of DWT coefficients, typically of size 64×64 or 32×32, called *code blocks*. This enables the spatial scalability feature of JPEG2000, as the code blocks can be decoded individually. Each code block is compressed as a sequence of bit planes. This essentially means that the most significant bit of each coefficient is encoded first, then the second most significant bit and so on. In this way, any prefix of the compressed bit stream can be decoded to get a low fidelity reconstruction, equivalent to using a coarser quantization step. This gives rise to JPEG2000's quality scalability.
The encoded bit planes or *quality layers* of all code blocks can be interleaved arbitrarily. Different orderings result in different progressions of resolution, quality, or spatial location. In the final combined bit stream, each block is preceded by a marker which identifies its contribution.

The actual entropy coding is performed by an adaptive binary arithmetic coder. However, straightforward arithmetic coding of the individual bits would be very inefficient. There exists a significant amount of correlation between the bits in one coefficient, and also between coefficients in a local neighborhood. To make use of these correlations, the arithmetic coder uses many different *contexts*. In the terms of Section 2.1.4, each context corresponds to a separate symbol frequency table. For each bit to be encoded, the context is chosen based on previously encoded bits of the same coefficient as well as the values of neighboring coefficients. The goal is that each context will closely model the probability distribution for one particular situation, so that arithmetic coding will be maximally effective.

JPEG2000 groups the bits of each coefficient into *sign*, *significance*, and *magnitude refinement*, and employs a number of contexts for each group. The significance and magnitude refinement bits taken together make up the magnitude of the coefficient. The significance part comprises the leading zero bits up to and including the first non-zero bit. The magnitude refinement part then contains the remaining bits. During

encoding, a coefficient is called *significant* if all its significance bits have already been visited, i.e. the first non-zero bit has been encountered. Fig. 2.14 demonstrates the decomposition into groups, and the order in which the individual bits are processed.

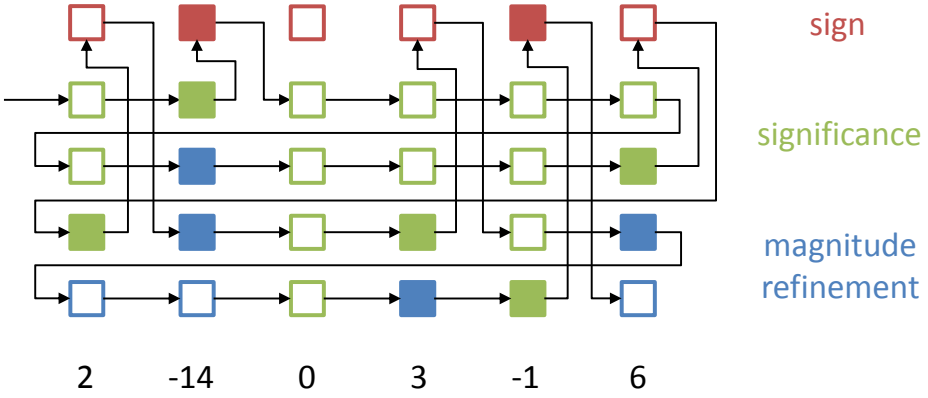

Figure 2.14: Bit plane scan order used in JPEG2000. Each box corresponds to one bit; each column represents a binary number in sign-magnitude representation. The bits of each number are grouped into three categories: the *sign* bit, the *significance* bits which are the leading zero bits up to and including the first non-zero bit, and the remaining *magnitude refinement* bits. The black arrows indicate the order in which the bits are encoded. The magnitude bits are processed in decreasing order of significance. The sign bit of each number is inserted when the first non-zero magnitude bit is encountered.

For significance bits, the context is chosen based on the significance of the eight immediate neighbors of the current coefficient. The assumption is that a coefficient is more likely to become significant if many of its neighbors are already significant. The context selection also takes into account to which DWT subband the current coefficient belongs (compare Fig. 2.10). In an HL subband, i.e. high-pass in x direction, large coefficients usually result from vertical edges. The context selection therefore gives more weight to the vertical neighbors. Conversely, in an LH subband, the horizontal neighbors are assigned greater weight. In HH subbands, the diagonal neighbors are most strongly weighted. In total, JPEG2000 uses nine contexts for significance coding, representing different "predictions" for the value of the bit to be encoded.

Sign coding is based on similar principles. One out of five contexts is chosen based on the signs of the current coefficient's four immediate neighbors. Neighbors which are not significant yet, i.e. whose sign is not known yet, are not useful for the purpose

of context selection and are ignored.

Finally, there are three contexts for coding magnitude refinement bits. Two are used only for the first magnitude refinement bit of each coefficient. The first is used if any of the coefficient's eight neighbors is already significant; the purpose is to make use of any correlations between the magnitudes of neighboring coefficients. If none of the neighbors are significant, the second context is used. For all remaining magnitude refinement bits, there exist little or no useful correlations, so they are all coded in the third and final context.

3

GPU Data Compression

There are two main criteria to quantify the performance of a compression system. The first is the compression ratio, or "quality per bit" for lossy compressors. The second is the throughput, i.e. how much data can be compressed or decompressed in a given amount of time. Depending on the application at hand, one can be more important than the other. For example, in a long-term storage scenario where the stored data is accessed infrequently, a high throughput may not be critical. On the other hand, in interactive applications, particularly the decompression throughput is often more important than the compression ratio. Here, the time required for "compressed transfer + decompression" must be smaller than the time for "uncompressed transfer" for the compression to be beneficial. If the data may be modified and needs to be stored again, an analogous argument applies to the compression throughput as well. Of course, to handle very large data sets interactively, a high compression ratio is critical as well.

The goal of this chapter is to develop a compression system which is suitable for interactive applications. The input data will be given on 2D or 3D Cartesian grids, such as images and height fields (see Chapter 4) or fluid simulation results (see Chapters 5 and 6). Both integer-valued and floating-point data should be supported. The compression ratio should be high enough so that very large data sets can be handled with limited memory. On the other hand, both compression and decompression should be fast enough to outperform a hard drive in terms of throughput.

Some existing compression standards such as JPEG2000 are versatile enough to

handle integer and floating-point data in 2D and 3D, and they achieve state-of-the-art compression ratios. However, even highly optimized CPU implementations of common image compression standards do not attain the necessary throughput. Achieving significantly higher throughput on the same hardware without sacrificing compression ratio seems impossible. Therefore, GPUs become an attractive option because of their massive computational power and high memory bandwidth. However, this power can only be realized effectively with *data-parallel* algorithms. This means that many small, similar, independent tasks are required.

Unfortunately, many data compression algorithms are inherently sequential, or at least typically formulated in a sequential way (see Chapter 2). So at first glance, data compression seems to be a poor fit for GPUs. In the course of this chapter, I will demonstrate that GPUs can be used effectively for data compression. I have implemented the presented algorithms in the CUDACOMPRESS library using NVIDIA's CUDA API. The source code of CUDACOMPRESS is publicly available under a permissive license [Tre13].

For efficient GPU compression, data-parallel formulations of a set of data compression algorithms are required. A straightforward technique to achieve parallelism is *blocking*: Partition the data into blocks, and compress each block separately. This allows all blocks to be handled independently and thus in parallel. However, it will also impact the compression ratio. First, any correlation or redundancy between blocks will be missed with this approach. Second, the offset of each block inside the compressed data stream needs to be stored, so that each decoder thread can jump directly to its segment of data. Thus, there is a trade-off in choosing the block size. Smaller block sizes will lead to more parallelism and therefore potentially better performance on a GPU. On the other hand, they will generally also result in a worse compression ratio. So blocking is used only when there is no other way to expose parallelism.

There are several libraries which provide "parallel primitives", i.e. operations which can be used as building blocks for data-parallel algorithms. Examples of such libraries include CUDPP [HOS*13] and thrust [HB13]. A surprisingly versatile operation offered by such libraries is the parallel prefix sum or scan. It computes for each element in an array the sum of all preceding elements. This can often be used to parallelize seemingly sequential algorithms, and is employed multiple times in

cudaCompress. Another common operation is parallel reduction, i.e. finding the sum or the maximum of the elements of an array.

In addition to the use of data-parallel algorithms, great care must be taken in the implementation to achieve good performance on GPUs. For example, the peak memory bandwidth can only be achieved if neighboring threads always access neighboring memory addresses simultaneously. The CUDA C Programming Guide [NVI13b] and CUDA C Best Practices Guide [NVI13a] by NVIDIA contain detailed descriptions of current GPU architectures along with many performance-related programming guidelines.

The remainder of this chapter is dedicated to the development and implementation of a GPU compression library which is efficient enough for interactive applications while still achieving the compression ratios necessary for handling very large data sets. The next section analyzes previous work on throughput-oriented applications of data compression. The goal of the following Section 3.2 is to find a combination of compression algorithms that fits well to the data-parallel GPU architecture while still achieving a competitive compression ratio. Afterwards, I describe efficient GPU implementations of the individual compression stages in detail in Sections 3.3, 3.4, and 3.5. Finally, in Section 3.6, I introduce the API of the cudaCompress library and demonstrate its use with some examples.

3.1 Related Work

In the following, I review the most popular compression options in applications where throughput is critical. Some of these works make use of the GPU, others use only the CPU.

First of all, there are some throughput-oriented CPU implementations of general-purpose dictionary methods which achieve a very high throughput [Obe11, Col13]. Consequently, they are suitable for interactive applications and have been applied e.g. in the context of volume rendering [FSK13]. However, being lossless, they typically achieve only modest compression ratios. Floating-point data in particular tends to compress very poorly with dictionary-based methods. For better compression, there are some lossless compression schemes specialized to floating-point data [BR09, LI06]. A GPU implementation of these methods achieves very high throughput [OB11].

However, the achieved compression ratio is still quite moderate.

Several previous works have implemented general-purpose compression algorithms on GPUs with the goal of achieving higher throughput than is possible on CPUs. Ozsoy et al. [OS11, OSC12] have implemented the LZSS algorithm on the GPU using CUDA. They achieve speedup factors of 2-3 compared to a CPU implementation for both encoding and decoding. However, this still does not result in sufficient throughput for interactive applications. There is also a significant impact on the compression ratio compared to the CPU version due to the use of blocking. Patel et al. [PZM*12] have developed a bzip2-like compression algorithm for GPUs using CUDA, including a Huffman encoder and decoder. However, their implementation achieves lower performance than the reference CPU version. Particularly the Huffman tree design performs poorly on the GPU due to a lack of parallelism in the algorithm. Overall, it can be said that the advantages of using GPUs for general-purpose data compression remain questionable.

However, GPUs have also been employed for efficient entropy coding. For a given Huffman table, Balevic [Bal09] has shown the efficient GPU realization of a Huffman encoder, but has not addressed fast decoding. Fraedrich et al. [FBS07] have presented a GPU Huffman decoder. Olano et al. [OBGB11] have introduced a texture compression scheme which includes arithmetic decoding on the GPU. In all cases, very significant speedups could be achieved over comparable CPU implementations.

There are many application areas for which specialized compression schemes have been developed. Volume rendering [EHK*06] is an application where routinely very large amounts of data must be handled interactively. Consequently, using data compression is a natural choice. The recent survey by Balsa Rodriguez et al. [BGI*13] provides a comprehensive treatment of compression techniques used in the context of volume visualization. In particular, transform coding has been applied to volume rendering using both the DCT [LMC02, YL95] and the DWT [GS01, GWGS02, NS01, Wes94].

Another popular compression technique for volume rendering is vector quantization. In vector quantization, a data set is represented by a small codebook of representative values and, for each data point, an index into this codebook. This allows for very fast decoding on the GPU via a single indirection [FMA05, SW03]. However, the construction of a good codebook is extremely time-consuming, and it is difficult to

satisfy the very high quality requirements of some applications (see e.g. Chapter 6).

Throughput-oriented compression techniques are also common for texturing in real-time graphics. Some special fixed-rate compression formats such as S3TC [INH03] are implemented directly in graphics hardware. These formats are popular since they enable hardware-supported random access on the GPU. This means that rendering, including hardware-supported interpolation, is possible directly from the compressed form, thus reducing GPU memory requirements. By adding a second, CPU-based compression stage, the compression ratio can be improved [NIH06, NIH08]. However, the fixed-rate first stage allows little or no control over the quality vs. compression ratio trade-off, and these formats lack support for floating-point data. For RGB data, S3TC achieves a moderate compression ratio of 6:1 in the DXT1 format. By converting the initial color samples into the YCoCg color space and using the DXT5 format, improved reconstruction quality can be achieved at the cost of doubling the compressed size [vWC07]. Additionally, while decompression is extremely fast, the compression step is usually quite involved. While the efficient compression of color fields into the S3TC format on the GPU is also possible, this comes at the cost of quality [vWC07].
Olano et al. [OBGB11] have presented a variable-rate texture compression format based on arithmetic coding of a multi-resolution difference pyramid. The format supports fast decoding on the GPU, but does not address fast encoding.

Terrain rendering [PG07] is another area where large data sets must be handled in real-time. There are two data modalities involved: color-valued orthophotos and scalar-valued height fields. The orthophotos are most commonly compressed with a GPU-supported format such as S3TC. For the height fields, on the other hand, a large variety of compression schemes exists [BGMP07, BGP09, DSW09, GMC*06, LC10, PSM*07, WZY08]. However, these approaches are typically tied to specific height field representations and are thus not applicable for other tasks. Also, all of these approaches address only fast decompression, while compression time is considered less important.

3.2 Choice of Algorithms

In Chapter 2, we have seen two classes of techniques to decorrelate data for the purpose of compression: dictionary techniques and transform coding. For data given on Cartesian grids, transform coding is a natural choice since it is known to provide excellent compression performance for image data. Many image compression algorithms incorporate transform coding, e.g. JPEG, JPEG2000 and JPEG-XR. Dictionary techniques have been used only in some lossless image compression techniques, e.g. in PNG and GIF. Additionally, even in optimized GPU implementations, dictionary techniques do not achieve the necessary throughput [OSC12]. On the other hand, both the DCT and the DWT are easily parallelized and thus fit the GPU well [OK08, WLHW07, TSP*08, vdLJR11]. The DWT achieves significantly better quality than the DCT at the same compression ratio in the sense of RMS error. Additionally, the multi-resolution properties of the DWT interact very well with level-of-detail approaches which are common in many graphics and visualization applications. Therefore, the DWT was chosen as the transform to use in cudaCompress.

The choice of the coder to use for the following entropy coding stage is more involved. The contestants must be evaluated both regarding their compression ratio as well as their suitability for an efficient GPU implementation. In the remainder of this section, I investigate the following options:

- An arithmetic coder,
- a Golomb-Rice coder,
- a Huffman coder,
- a run-length coder which only handles runs of zeros, similar to the one used in JPEG (see Section 2.5.1),
- a combined run-length and Golomb-Rice coder, where the Golomb-Rice coder compresses the remaining symbols as well as the run lengths, and
- a combined run-length and Huffman coder, where the Huffman coder compresses the remaining symbols as well as the run lengths.

All of these coders are usually formulated in a sequential way (see Sections 2.1 and 2.2). I now address what changes must be made to each coder to allow a ba-

sic data-parallel implementation. The low-level details necessary to actually achieve efficient GPU implementations are discussed in Sections 3.4 and 3.5.

3.2.1 Arithmetic Coder

Arithmetic coding is an attractive option because it achieves almost optimal compression within the framework of entropy coding. However, both encoder and decoder contain sequential data dependencies due to the incremental updating of the current interval bounds. The only option for a parallel implementation thus is encoding small blocks of data individually. Compared to a sequential arithmetic coder, this results in two kinds of storage overhead. First, the encoder must be "flushed" at the end of each block. Second, either the compressed sizes of the individual blocks or their offsets in the combined bit stream must be stored so that each decoder thread knows where to begin.

A secondary concern is whether to use adaptive coding (see Section 2.1.5). An adaptive coder does not require a separate frequency analysis pass, and often achieves a slightly better compression ratio. However, there are two reasons which make adaptive coding inappropriate in this case:

1. There is a penalty associated with starting the adaptive coding process, as the coder needs to learn the statistics of the input data before effective compression is possible. If many small blocks are encoded separately, this penalty becomes very significant.
2. An adaptive coder must maintain its probability table. Storing such a probability table for each thread easily exceeds the GPU's fast memory.

I conclude that a non-adaptive coder is the more appropriate choice for a parallel implementation. This means that a frequency analysis pass is required before the actual encoding. This pass basically amounts to the computation of a histogram, which can be done efficiently on GPUs [Pod07a].

3.2.2 Golomb-Rice Coder

A Golomb-Rice encoder replaces each symbol by a predefined codeword and concatenates all codewords to get the final compressed representation. The first step,

assigning codewords, is fully parallel: each codeword can be assigned independently. The second step, concatenating the codewords, seems to be sequential at first. However, a prefix sum on the codeword lengths produces the position of each codeword in the compressed data stream. With this information, the concatenation can be performed in parallel as well.

The decoder, however, can not be parallelized immediately. To allow multiple decoder threads, it becomes necessary to store an entry point into the compressed stream for each thread. This results in some storage overhead compared to the sequential coder.

3.2.3 Huffman Coder

Huffman coding, like arithmetic coding, first requires a frequency analysis step which amounts to a histogram of the input data. Based on the histogram, a Huffman table must be constructed (see Section 2.1.2). The table construction algorithm is strongly sequential, and there is no known practical way of parallelizing it. Fortunately, the runtime cost of this step is very minor, since typically no more than a few hundred items need to be handled. It is therefore acceptable to perform this step sequentially on the CPU.

The actual Huffman encoding and decoding then proceed very similarly to the Golomb-Rice coder. The only difference is that the codewords are read from the Huffman table rather than being statically predefined. Thus it is also required to store entry points into the compressed stream for the decoder threads.

3.2.4 Run-Length Coder

Run-length encoding basically amounts to a *stream compaction* operation, which in turn requires little more than a prefix sum. Stream compaction is offered by several GPU libraries such as thrust [HB13] and CUDPP [HOS*13, HSO07]. Consequently, no further adjustments are necessary to allow a parallel implementation of run-length encoding.

The decoder boils down to a scattered write operation. The output indices for the scattered write can be computed by a prefix sum on the run lengths.

3.2.5 Compression Ratio Comparison

I used two standard image suites to test the encoders: the Kodak test images [FE99] (called *kodim* in the following) and the "New Test Images" (RGB 8-bit) [Raw11] (*newtest*). I applied the following test procedure:

- Convert the RGB image data to the YCoCg color space.
- Perform a three-level DWT using the CDF 9/7 wavelet to each image channel.
- Quantize all high-pass coefficients.
- Encode the quantizer output for each subband separately.
- Record the final output bit rate as well as the entropy of the quantizer output.

The tests were repeated for a variety of quantization steps between 0.1 and 20. Additionally, each test was performed once with the regular sequential coders and once with the adapted parallel versions. Figs. 3.1 and 3.2 plot the results of the sequential coders for kodim and newtest, respectively. The graphs show the coding overhead, i.e. output bit rate minus entropy, plotted vs. the entropy of the coder input. For reference, "good visual quality" is achieved at an entropy of about 0.5 to 1 bit. The results mostly concur with the expectations. Arithmetic coding performs very close to the entropy, with some overhead for storing the frequency tables. Huffman coding works well at a high entropy, but introduces significant overhead below an entropy of about 2 bits. Golomb-Rice coding has only a small disadvantage compared to Huffman coding. Run-length coding on its own is only useful at extremely low entropy. However, when combined with Golomb-Rice or particularly Huffman coding, it works extremely well. Below one bit of entropy, the run-length + Huffman coder even slightly beats the entropy.

The results for the parallel case are more interesting. They are plotted in Figs. 3.3 and 3.4 for kodim and newtest, respectively. Where blocking is necessary, the block size was chosen to be 128 elements as a compromise between exposing parallelism and limiting storage overhead. The block sizes were stored as 16-bit integers. Accordingly, the entropy coders, i.e. Golomb-Rice, Huffman, and arithmetic, each suffer a bit rate penalty of roughly $16/128 = 0.125$ bits per element. Run-length coding requires no blocking and thus suffers no penalty. This is particularly beneficial in combination

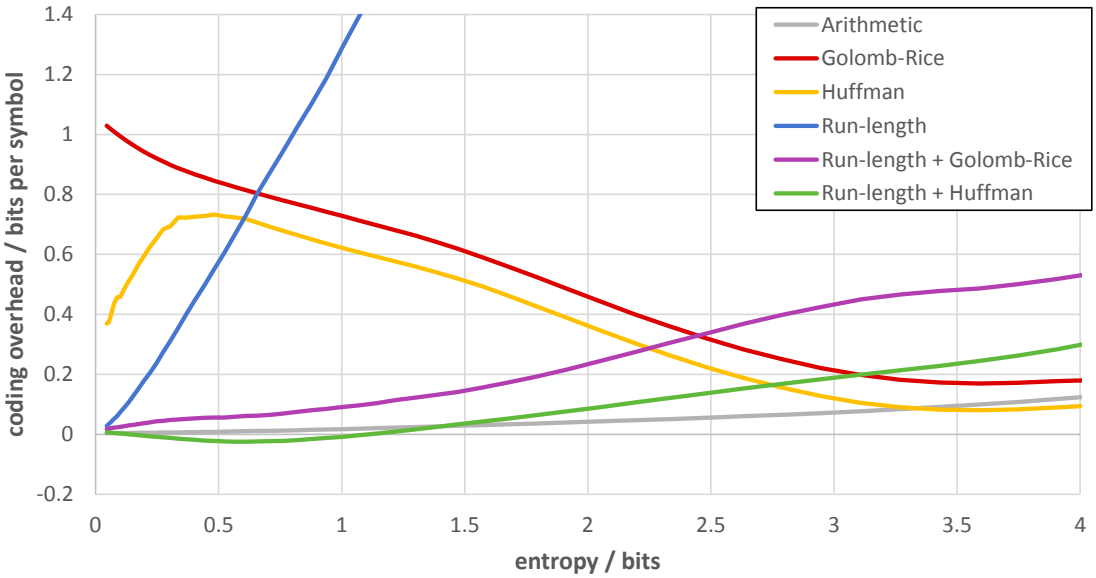

Figure 3.1: Coding overhead (i.e. bit rate minus entropy) vs. entropy for various sequential coders applied to the kodim suite.

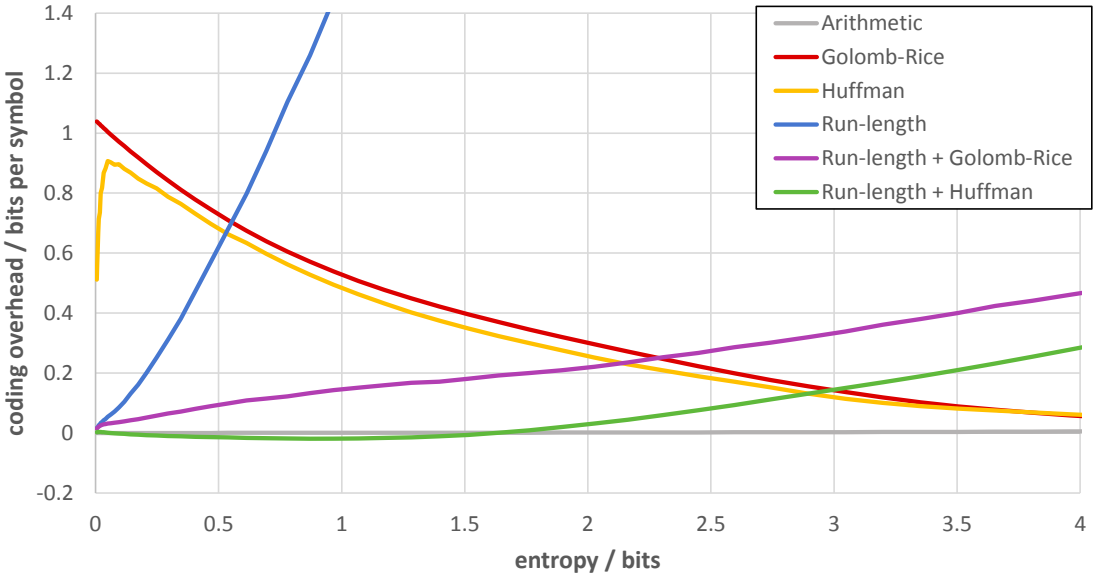

Figure 3.2: Coding overhead (i.e. bit rate minus entropy) vs. entropy for various sequential coders applied to the newtest suite.

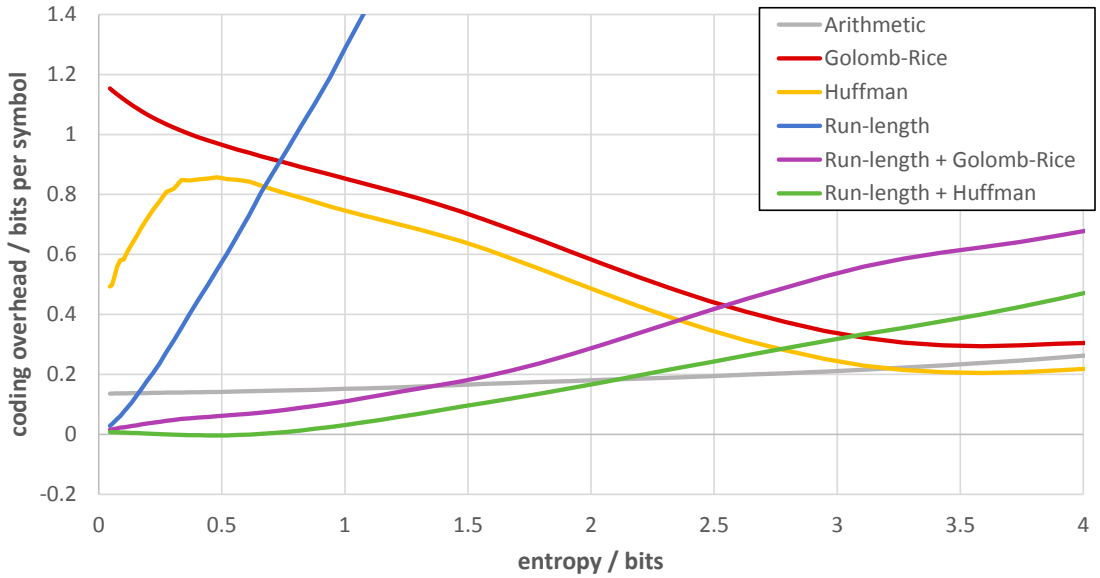

Figure 3.3: Coding overhead (i.e. bit rate minus entropy) vs. entropy for various parallel coders with a block size of 128 elements applied to the kodim suite.

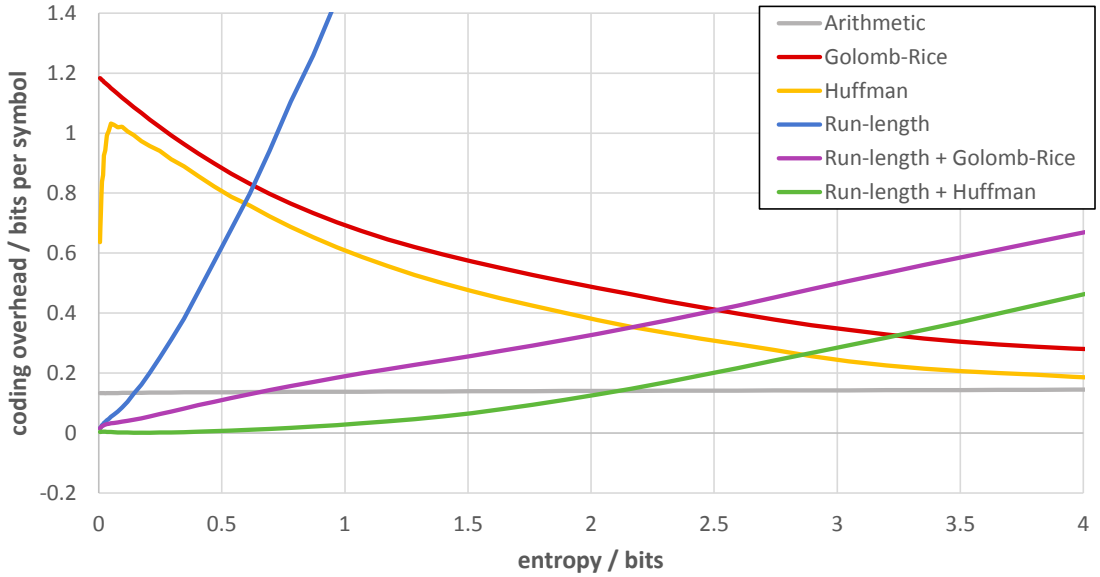

Figure 3.4: Coding overhead (i.e. bit rate minus entropy) vs. entropy for various parallel coders with a block size of 128 elements applied to the newtest suite.

with Golomb-Rice or Huffman coding. At low entropy, the run-length coder strongly reduces the number of elements that have to be handled by the entropy coder. As a side-effect, this reduces the storage overhead due to blocking. Consequently, the combined run-length + entropy coders perform extremely well at low entropy. The run-length + Huffman coder beats the arithmetic coder up to an entropy of about 2 bits.

This makes the run-length + Huffman coder the first choice in terms of compression ratio. Using Golomb-Rice instead of Huffman coding results in a moderate compression ratio penalty. If the throughput were significantly higher, then Golomb-Rice coding could be a valid choice. Both the encoding and the decoding procedures are very similar, and can be expected to achieve similar performance. The main difference is the lack of a frequency analysis step in the Golomb-Rice encoder. However, it is also possible to use static precomputed Huffman tables. This allows skipping the frequency analysis step during Huffman encoding at the cost of compression ratio. Thus Huffman coding is the more general choice and has no significant disadvantage compared to Golomb-Rice coding. Therefore, CUDACompress implements the run-length + Huffman coder as its entropy coding stage.

Having made a choice regarding the compression algorithms to use, I now proceed to describe in detail their efficient implementation on current GPUs.

3.3 Discrete Wavelet Transform

In Chapter 2, two implementation strategies for the DWT were introduced: convolution with linear filters and lifting. The DWT has been implemented on the GPU using both convolution [WLHW07, TSP*08] and lifting [vdLJR11]. Separable image convolution can be implemented very efficiently using CUDA [Pod07b]. The basic strategy is to load a block of pixels into shared memory, perform the convolution on each pixel, and directly write the result back to global memory. Some overlap between the blocks is necessary to cover the support of the filter kernel—e.g. with the CDF 9/7 wavelet, a "halo" region of 4 pixels is required. For blocks at the boundary of the image, the halo pixels are created by mirrored extension.
Only few changes to the basic image convolution implementation are necessary to support the DWT. In the forward transform, different filters are used for even and

odd pixels: the low-pass filter for even pixels, the high-pass filter for odd pixels. The low-pass and high-pass results are written to separate memory regions to achieve the subband order which is convenient for further processing (compare Fig. 2.10). Similar adaptations are necessary in the inverse transform. This implementation is extremely efficient on current GPUs, achieving a throughput close to the maximum memory bandwidth of the device (see Section 3.6.2).

A drawback of the convolution-based implementation is that it does not support a reversible integer DWT. Therefore, CUDACompress also contains a lifting-based implementation of the integer CDF 5/3 wavelet. The basic strategy remains the same as in the convolution case: load a block into shared memory, perform the computations there, and write back the result. In CPU implementations, lifting usually achieves superior performance because it requires fewer arithmetic operations. On the GPU, however, the ratio of compute power to memory bandwidth is so large that the arithmetic operations in the convolution-based implementation do not matter. On the other hand, the lifting implementation requires synchronization points after each lifting step as well as multiple round trips to shared memory, resulting in reduced performance. Therefore, CUDACompress uses the convolution-based implementation to compute floating-point DWTs, and reverts to the lifting-based implementation only for integer DWTs.

3.4 Run-Length Coding

In general, run-length encoding replaces multiple sequential occurrences of the same symbol in a data stream (a *run*) by one single instance of the symbol plus a number indicating how often the symbol occurs. During transform coding, many runs of symbols equal to zero are expected, while runs of other symbols are rather unlikely. Therefore, CUDACompress implements a variant of run-length encoding which only handles runs of zeros: Each non-zero symbol in the input stream is replaced by a pair of values containing the original symbol and the number of zeros that precede it in the stream, called the *zero count*.

3.4.1 Encoder

Run-length encoding can easily be expressed in terms of data-parallel operations. A graphical illustration of the run-length encoding implementation in cudaCompress is shown in Fig. 3.5. First, each symbol is flagged as either zero (marked by 0) or non-zero (marked by 1). An exclusive prefix sum over these flags generates the output indices for all non-zero symbols. Next, the zero count for each symbol is computed by subtracting from the symbol's index the index of its predecessor plus one. Trailing zeros in the data do not need to be stored, since the total number of symbols is known to the decoder.

An additional complication arises because of the subsequent Huffman coding: Very large input values to the Huffman coder can result in long codewords and thus reduce the compression ratio. Therefore, the zero counts should be limited to some maximum value, which is chosen as 255 in cudaCompress. To achieve this, additional zeros are inserted into the compacted symbol stream at locations where the zero count exceeds the given limit. This is realized by first computing the number of additional zeros to insert at each index, and then performing an inclusive prefix sum over these values to obtain an offset for each entry. Finally, each symbol and associated zero count is re-positioned according to this offset in the symbol stream.

3.4.2 Decoder

Similar to the encoder, the run-length decoder can be written in terms of parallel operations. Fig. 3.6 illustrates the process. The first step is an inclusive prefix sum over the zero counts, following by adding to the resulting values the index of the respective element. This determines the original indices of the non-zero symbols. For efficiency reasons, both computations are performed in a single kernel. Finally, the output array is cleared with zeros, and each symbol is written to its target position in a scattered write operation.

3.5 Huffman Coding

To further compress the run-length encoded symbol stream, cudaCompress employs Huffman coding. Since the symbols and zero counts typically occur with very

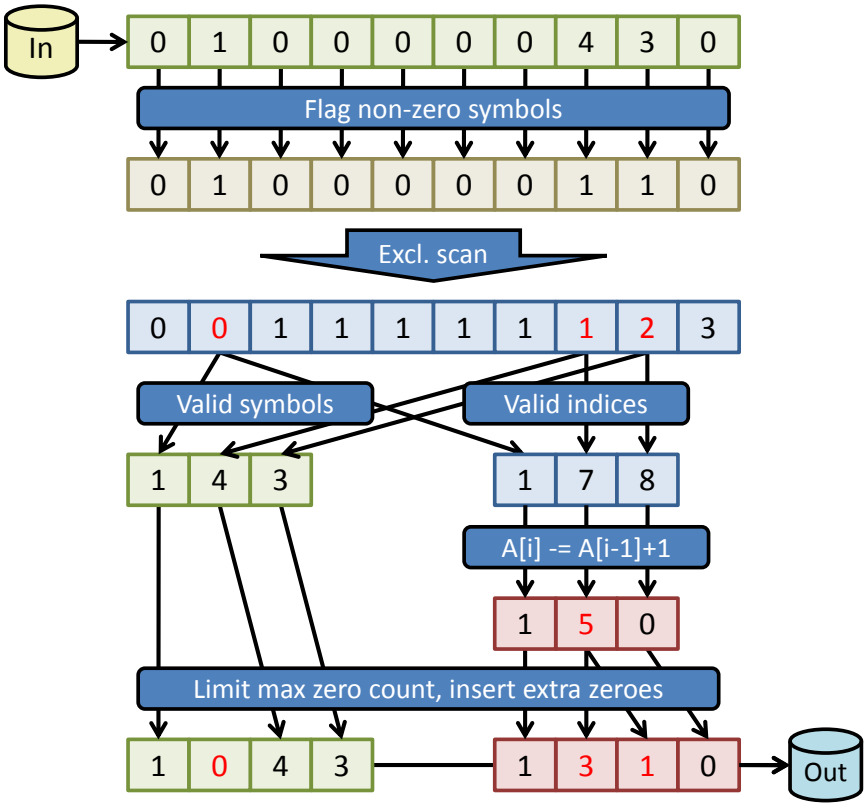

Figure 3.5: Parallel implementation of run-length encoding. Input symbols (green arrays) are flagged as zero or non-zero. A scan operation generates output indices (blue arrays). The number of removed zeros is stored (red arrays), and zeros may be inserted into the symbol stream to limit the maximum zero count (to 3 in the example).

different distributions, they are encoded separately.

3.5.1 Encoder

The Huffman encoder performs two basic operations: It computes the Huffman table for the input data and then performs the encoding of the data stream using this table.

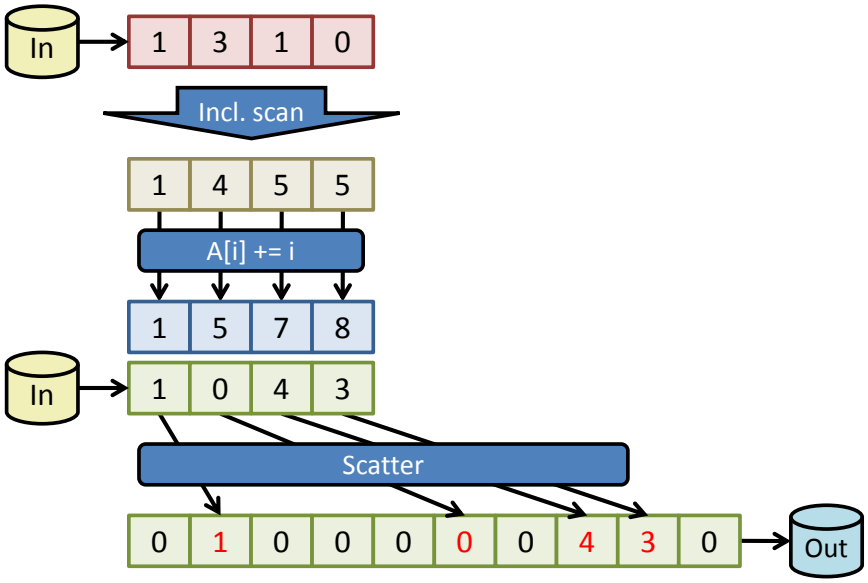

Figure 3.6: Parallel implementation of run-length decoding. A scan over the zero counts (red array) yields preceding zeros of each symbol. Adding each element's index gives the original indices (blue array) of stored symbols (green array). A scattered write finally reconstructs the original data.

Table Design

The computation of the Huffman table can be further subdivided into (a) the computation of the relative probabilities of all occurring symbols and (b) the assignment of codewords of appropriate lengths to each symbol. The first step is equivalent to building a histogram of the input data, which can be realized on the GPU using atomic operations. However, atomic operations become very inefficient if write conflicts occur and thus concurrent accesses are serialized. As the distribution of wavelet coefficients is very likely to be heavily skewed towards small values, many such conflicts are expected in the first few histogram bins.

To avoid these conflicts, CUDACompress stores one histogram *per thread* in shared memory and combines these histograms in a second pass using a parallel reduce operation per bin [Pod07a]. However, the number of histogram bins that can be processed per thread is strongly limited by the available shared memory. This means that many passes over the data may be required if the number of bins is large. Thus,

cudaCompress uses this technique for the first few bins, where the largest number of conflicts is expected. The execution then switches to a kernel that computes one histogram per warp using shared memory atomics [Pod07a, SK07]. Even though this can still lead to memory conflicts, experiments have shown superior performance since the number of conflicts is small and a much larger number of bins per pass can be used.

The algorithm for computing the Huffman table exposes little parallelism, and a GPU implementation has been shown to perform poorly [PZM*12]. Therefore, cudaCompress realizes this step on the CPU. This requires a round-trip to the CPU, but since the Huffman table has to be stored to disk anyway, there is little additional overhead. The table construction including the GPU-CPU data transfer consumes less than 10% of the overall encoding time.

Encoding

After the Huffman table has been built, the encoder replaces each symbol by its codeword and performs a bit stream compaction. A naive implementation writes for each symbol the bit-length of its codeword into an auxiliary buffer and uses an exclusive prefix sum over these numbers to compute the position of each symbol in the bit stream. Then, each symbol can be written to the respective position. However, as the codewords are not aligned to word boundaries, this step requires atomic operations, and since most codewords are much shorter than a memory word (32 bits) the performance is slowed down considerably by a high number of write conflicts.

The data-parallel implementation used cudaCompress is illustrated in Fig. 3.7. To reduce the number of conflicts, each thread writes k consecutive codewords. This also means that only every k-th element of the bit index array is needed, so cudaCompress first sums every k adjacent codeword lengths and then performs the prefix sum operation only on the smaller set of elements. For optimal memory bandwidth use, the compaction step writes to shared memory. The compacted array can then be written to global memory using coalesced memory transactions.

The decoder needs one of the codeword bit indices to start each decoder thread. Thus, cudaCompress stores every m-th element of the index array, i.e. the bit index

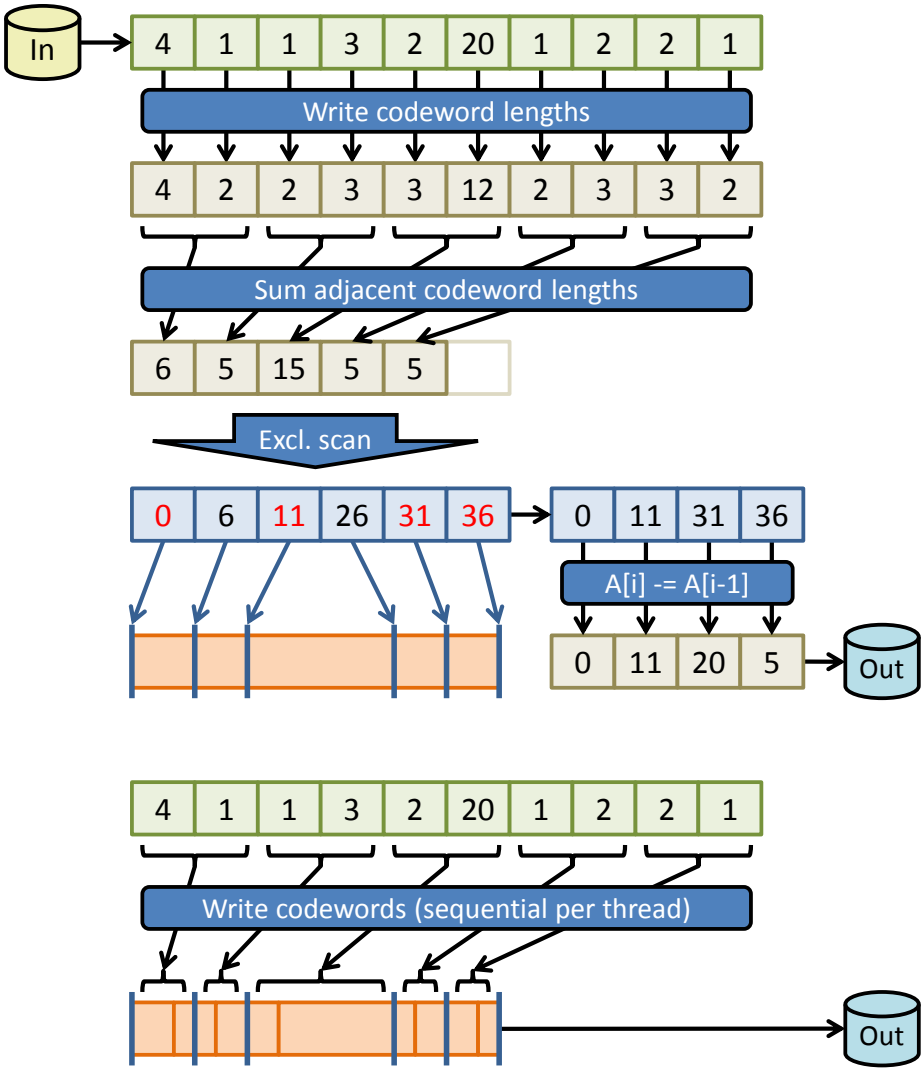

Figure 3.7: Parallel implementation of Huffman encoding. Lengths of codewords corresponding to input symbols (green array) are written into an auxiliary buffer. Every k adjacent values ($k = 2$ in the example) are summed up. A scan operation computes the output bit index (blue array) for every k-th codeword. Every m-th bit index ($m = 2$ in the example) is stored as side information for the decoder. The last element of the index array carries the length of the compacted stream and is also stored. In the final step, codewords are written to an output buffer (orange array) in groups of k per thread.

of every $k \cdot m =: n$-th codeword, as side information. In the current implementation, $k = 8$ and $m = 16$, so every 128-th index is stored. To save memory, increments between indices are stored instead of absolute values. This allows storing the increments as 16 bit integers, which is sufficient to accommodate 128 codewords of 511 bits each—far more than the maximum possible codeword length for reasonable numbers of input symbols [AMM00].

3.5.2 Decoder

Fig. 3.8 illustrates the parallel implementation of the Huffman decoder. The decoder receives the codeword stream and the incremental bit indices that were stored by the encoder as side information. An inclusive prefix sum computes the bit index of every n-th codeword. For each such index, one thread decodes n symbols sequentially. The implementation of this step is similar to the one proposed by van Waveren [vW06], but cudaCompress builds the lookup table for short symbols on the GPU.

Every $t = 32$ consecutive threads (one warp) write their symbols simultaneously in an interleaved order during decoding. In this way, all threads in one warp write to consecutive memory addresses, resulting in coalesced accesses. In a second pass, each block of $t \times n$ interleaved values is read into shared memory, reordered, and written back to global memory. Taking advantage of coalescing in this way results in an improvement of the decoder throughput by about 40%.

Making the decoder threads each write a symbol simultaneously implies that they have to read from their input bit streams at different speeds, as their input codewords generally have different lengths. Therefore, the reads can not be coalesced. To avoid frequent accesses to the input data in global memory, each thread caches 2×32 bits of the input bit stream in registers. In this way, the threads only have to access the global memory after a codeword has been decoded completely.

3.6 The cudaCompress Library

I have implemented the described parallel data compression algorithms and released the resulting library, called cudaCompress, under a permissive open-source license [Tre13]. In particular, cudaCompress contains efficient implementations of

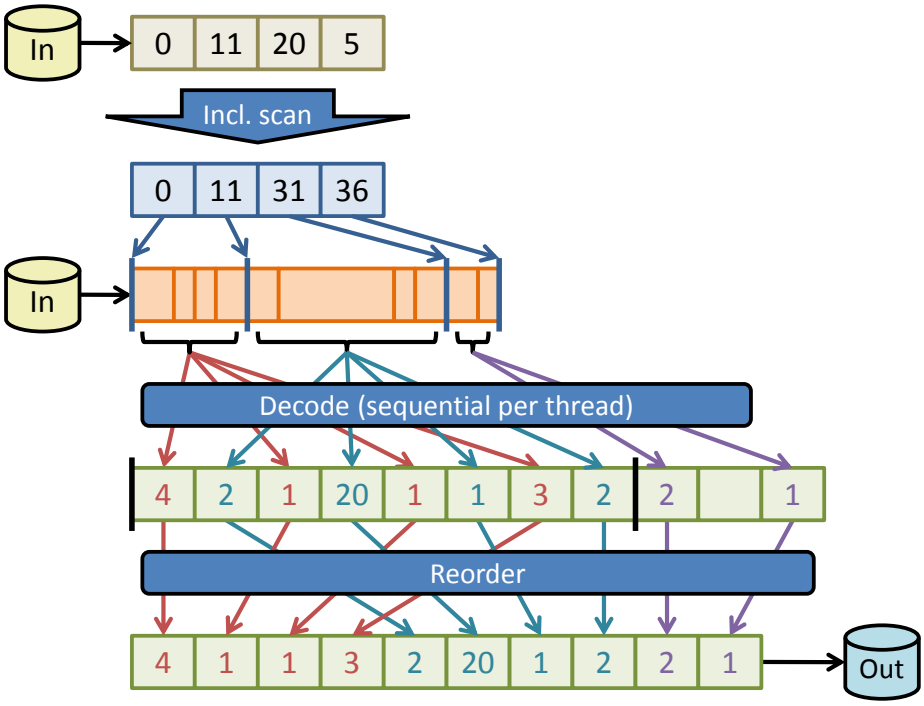

Figure 3.8: Parallel implementation of Huffman decoding. A scan operation over relative bit indices gives the bit indices (blue array) of every n-th codeword ($n = 4$ in the example). n symbols per thread are sequentially decoded (green array) from their codewords (orange array). To achieve coalesced memory accesses, t consecutive threads ($t = 2$ in the example) write their output in an interleaved order. A final blockwise reordering operation restores the correct order of elements.

the DWT using the CDF 9/7 and integer CDF 5/3 wavelets as well as run-length and Huffman coding. Additionally, CUDACOMPRESS provides utility functions such as quantization and color space transforms between RGB and YCoCg. This functionality can be combined to create efficient lossy compression systems for integer and floating-point data given on Cartesian grids. Lossless compression of integer data is also possible. However, effective lossless compression of floating-point data requires different techniques and is presently not supported.

The core of CUDACOMPRESS is composed of a run-length and a Huffman coder. For maximum efficiency during encoding and decoding, all temporary resources such as GPU buffers are preallocated by creating a `cudaCompress::Instance`. This `Instance` is then passed to the actual encoding functions such as `encodeRLHuff`

which performs run-length and Huffman encoding. Other functionality such as the DWT and quantization, which do not depend on preallocated resources, are provided in the `cudaCompress::util` namespace.

3.6.1 Usage Example

The operations offered by cudaCompress can be combined with ease to create a full data compression algorithm. Fig. 3.9 lists the complete source code of a program implementing a simple image compression algorithm using cudaCompress. The two functions implementing compression and decompression, `compressImage` and `decompressImage`, each consist of only four function calls. While this is a very simple example, it demonstrates the convenience of implementing data compression on top of cudaCompress. As more advanced examples, the cudaCompress distribution [Tre13] contains the full source code of the compression algorithms used in the systems described in Chapters 4, 5, and 6.

3.6.2 Performance

After demonstrating how cudaCompress can be used, I now present some benchmark results to show the achieved performance. All benchmarks were performed on a PC with an Intel Xeon E5520 CPU (quad core, 2266 MHz), 12 GB of DDR3-1066 RAM, and an NVIDIA GeForce GTX 580 graphics card. Here, I analyze the individual operations offered by cudaCompress. Performance results for complete compression systems built on top of cudaCompress are given in the following chapters, in particular in Sections 4.5.3, 5.5, and 6.5.3.

Discrete Wavelet Transform

The DWT does not perform any data-dependent operations, so its performance is independent of the particular input values. Computing the forward floating-point CDF 9/7 DWT on a 1024×1024 grayscale image requires 0.11 ms. This corresponds to a throughput of 9500 MPix/s, or an effective memory bandwidth of 142 GB/s. The inverse transform achieves 9200 MPix/s or 137 GB/s. For comparison, a device-to-device `cudaMemcpy` achieves 162 GB/s on the same hardware.

```
#include <fstream>

#include <cuda_runtime.h>

#include <cudaCompress/Instance.h>
#include <cudaCompress/Encode.h>
#include <cudaCompress/util/DWT.h>
#include <cudaCompress/util/Quantize.h>

// Global resources shared by compressImage and decompressImage.
cudaCompress::Instance* pInstance = nullptr; // the cudaCompress Instance.
float* dpScratch = nullptr;                  // scratch buffer for DWT.
float* dpBuffer = nullptr;                   // output buffer for DWT.
cudaCompress::Symbol16* dpSymbols = nullptr; // input/output for entropy coder.

cudaCompress::BitStream compressImage(
    const unsigned char* dpImage,  // input image in GPU memory
    int sizeX, int sizeY,          // image size
    float quantStep)               // quantization step
{
    // Expand image values to float and do first-level DWT.
    cudaCompress::util::dwtFloat2DForwardFromByte(
        dpBuffer, dpScratch, dpImage, sizeX, sizeY);
    // Do second-level DWT in the same buffers. Need to specify pitch now!
    cudaCompress::util::dwtFloat2DForward(
        dpBuffer, dpScratch, dpBuffer,  sizeX/2, sizeY/2, 1, sizeX, sizeY);
    // dpBuffer now contains the multi-level DWT decomposition.

    // Quantize the coefficients and convert them to unsigned values (symbols).
    // For better compression, quantStep should be adapted to the transform level!
    cudaCompress::util::quantizeToSymbols2D(dpSymbols, dpBuffer, sizeX, sizeY, quantStep);

    // Run-length + Huffman encode the quantized coefficients.
    cudaCompress::BitStream bitStream;
    cudaCompress::encodeRLHuff(pInstance, bitStream, &dpSymbols, 1, sizeX * sizeY);
    return bitStream;
}

void decompressImage(
    cudaCompress::BitStream& bitStream, // compressed image data
    unsigned char* dpImage, int sizeX, int sizeY, float quantStep)
{
    cudaCompress::decodeRLHuff(pInstance, bitStream, &dpSymbols, 1, sizeX * sizeY);

    cudaCompress::util::unquantizeFromSymbols2D(dpBuffer, dpSymbols, sizeX, sizeY, quantStep);

    cudaCompress::util::dwtFloat2DInverse(
        dpBuffer, dpScratch, dpBuffer, sizeX/2, sizeY/2, 1, sizeX, sizeY);
    cudaCompress::util::dwtFloat2DInverseToByte(
        dpImage, dpScratch, dpBuffer, sizeX, sizeY);
}
```

Figure 3.9: Example code for simple image compression using cudaCompress.

```
void main()
{
    int sizeX = 1024, sizeY = 1024;
    float quantStep = 4.0f;
    unsigned char* dpImage = nullptr;

    // Read image data from file.
    std::vector<unsigned char> image(sizeX * sizeY);
    std::ifstream file("image.raw", std::ifstream::binary);
    if(!file.good()) return;
    file.read((char*)image.data(), sizeX * sizeY);
    file.close();

    // Initialize cudaCompress, allocate GPU resources and upload data.
    pInstance = cudaCompress::createInstance(-1, 1, sizeX * sizeY);

    cudaMalloc(&dpImage, sizeX * sizeY);
    cudaMemcpy(dpImage, image.data(), sizeX * sizeY, cudaMemcpyHostToDevice);

    cudaMalloc(&dpScratch, sizeX * sizeY * sizeof(float));
    cudaMalloc(&dpBuffer, sizeX * sizeY * sizeof(float));
    cudaMalloc(&dpSymbols, sizeX * sizeY * sizeof(cudaCompress::Symbol16));

    // Compress the image.
    cudaCompress::BitStream bitStream = compressImage(dpImage, sizeX, sizeY, quantStep);

    // Write compression rate to stdout.
    int compressedSize = bitStream.getBitSize();
    float ratio = float(sizeX * sizeY * 8) / float(compressedSize);
    printf("Compressed size: %i b  (%.2f : 1)\n", compressedSize, ratio);

    // Rewind bitstream and decompress.
    bitStream.setBitPosition(0);
    decompressImage(bitStream, dpImage, sizeX, sizeY, quantStep);

    // Download reconstructed image and write to file.
    std::vector<unsigned char> imageReconst(sizeX * sizeY);
    cudaMemcpy(imageReconst.data(), dpImage, sizeX * sizeY, cudaMemcpyDeviceToHost);
    std::ofstream out("image_reconst.raw", std::ofstream::binary);
    out.write((char*)imageReconst.data(), sizeX * sizeY);
    out.close();

    // Cleanup.
    cudaFree(dpSymbols);
    cudaFree(dpBuffer);
    cudaFree(dpScratch);

    cudaFree(dpImage);

    cudaCompress::destroyInstance(pInstance);
}
```

Figure 3.9: Example code for simple image compression using cudaCompress (cont'd).

The integer CDF 5/3 DWT applied to a 1024×1024 image of 2-byte integers achieves 11000 MPix/s in the forward and 10400 MPix/s in the inverse transform. Despite the simpler filters and smaller data elements, this is only slightly more than in the floating-point case. This is due to the use of integer arithmetic, which on a GPU is much more expensive than floating-point arithmetic, as well as additional synchronization points after each lifting step.

In transform coding, computing a forward DWT is often followed by quantization. Similarly, the inverse DWT is often preceded by dequantization. To avoid another pass over the data to perform the quantization step, cudaCompress also contains combined DWT and quantization/dequantization kernels. Since the quantization step represents very little overhead, these run at essentially the same speed as the regular DWT kernels.

Coding

In contrast to the DWT, the performance of the run-length and Huffman coders does depend on the input data. Their performance is therefore more difficult to analyze comprehensively. Data which compresses better typically also results in fewer computations and memory accesses and thus higher throughput. Another factor is how "uniform" the data is, i.e. how well the individual GPU threads are balanced.

I have benchmarked the run-length and Huffman coders using the following procedure on a grayscale image of size 2048×2048:

- Apply two levels of a DWT using the CDF 9/7 wavelet.
- Split the resulting coefficients into 4×4 tiles of size 512×512 each.
- Discard the tile corresponding to the low-pass subband, because its coefficients have very different characteristics compared to the other tiles.
- Quantize the remaining 15 tiles.
- Finally, encode the quantizer labels using the Huffman coder as well as the combined run-length + Huffman coder.

I have repeated the procedure with a range of quantization steps. The graphs in Figs. 3.10 and 3.11 plot the run-length + Huffman coding time vs. the entropy of

the quantized coefficients. All reported times were averaged over 100 runs. The timings include all required data transfers between CPU and GPU. As expected, input data of a lower entropy generally results in lower computation times in both encoding and decoding. The encoding times for the run-length + Huffman coder range from about 4 ms at 0.1 bits of entropy to 7.5 ms at 5 bits. This corresponds to a throughput of 520 to 980 million elements per second. The decoding times are between about 0.5 ms and 4.2 ms for a throughput between 930 and 7860 million elements per second. When the Huffman coder is applied directly to the data without the run-length coding step (Fig. 3.12), it achieves a throughput between 710 and 980 million elements per second. The Huffman decoder achieves between 3000 and 6550 elements per second.

3.6.3 Compression Quality

To assess the compression ratio and reconstruction quality that are achieved with cudaCompress, I have performed a number of tests using the Kodak test image suite [FE99] (*kodim*) and the "New Test Images" suite (RGB 8 bit) [Raw11] (*newtest*). On each image a three-level DWT was performed, and the resulting coefficients were compressed as described.

I have compressed the same images using JPEG, JPEG2000, and the S3TC DXT1 format. For JPEG and JPEG2000 compression, I used the ImageMagick library v. 6.8.7 [Ima13]. The S3TC compression was performed using the Squish library v. 1.11 [Bro08] at the highest quality setting, iterative cluster fit.

Figs. 3.13 and 3.14 show the compression quality in dB of RGB PSNR depending on the bit rate in bits per pixel (bpp). It can be seen that JPEG2000 gives the best results in terms of compression rate. The compression using cudaCompress usually outperforms JPEG, often significantly, except for bit rates below 1.3 bpp for kodim and below 0.5 bpp for newtest. However, at such low bit rates, neither algorithm can produce visually acceptable results. The fixed-rate DXT1 compression is clearly outperformed by all other approaches.

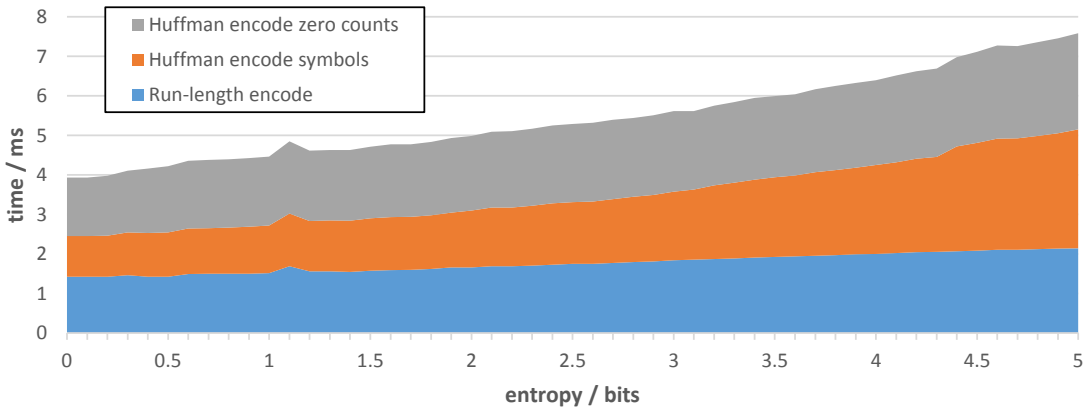

Figure 3.10: Encoding times for the run-length + Huffman coder vs. input entropy.

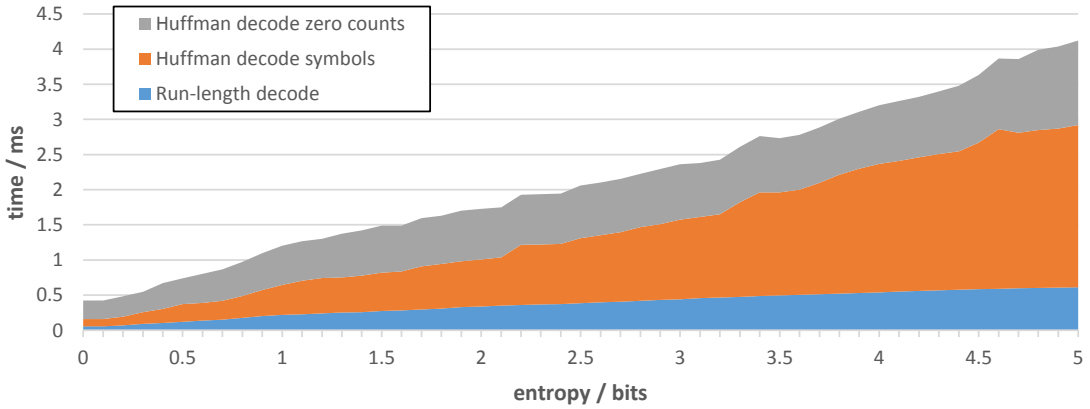

Figure 3.11: Decoding times for the run-length + Huffman coder vs. input entropy.

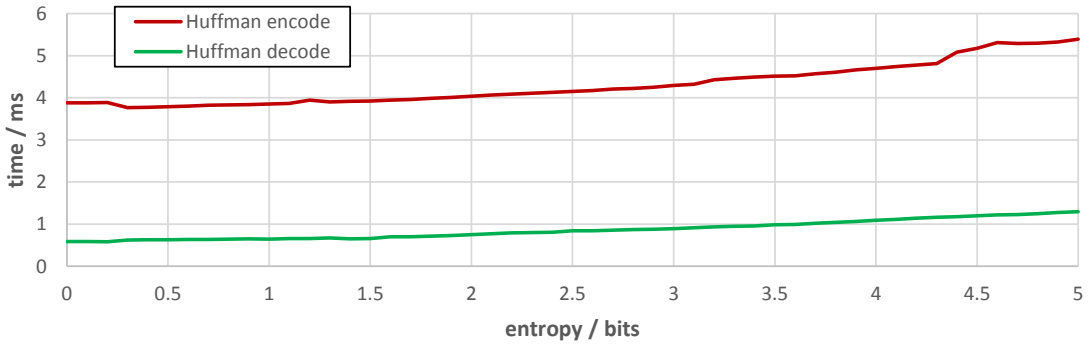

Figure 3.12: Encoding and decoding times for the Huffman coder vs. input entropy.

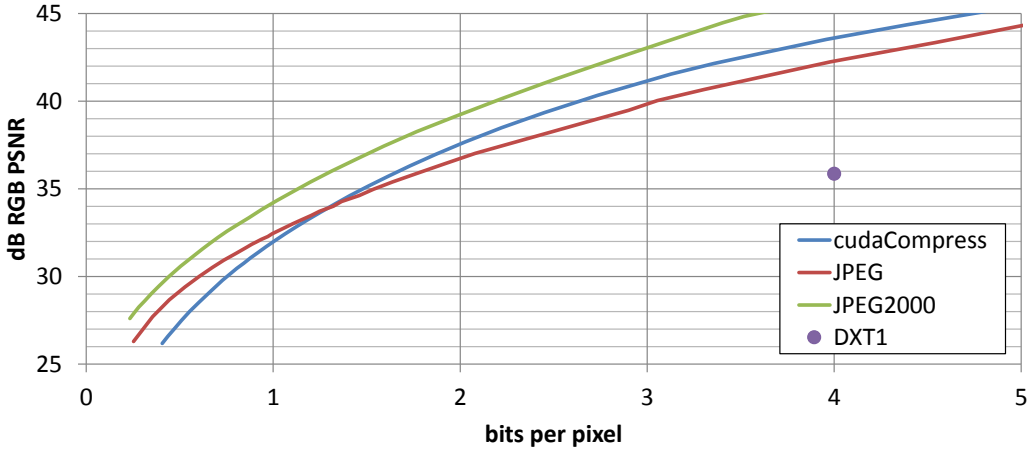

Figure 3.13: Graph of PSNR vs. bpp for the test image suite kodim.

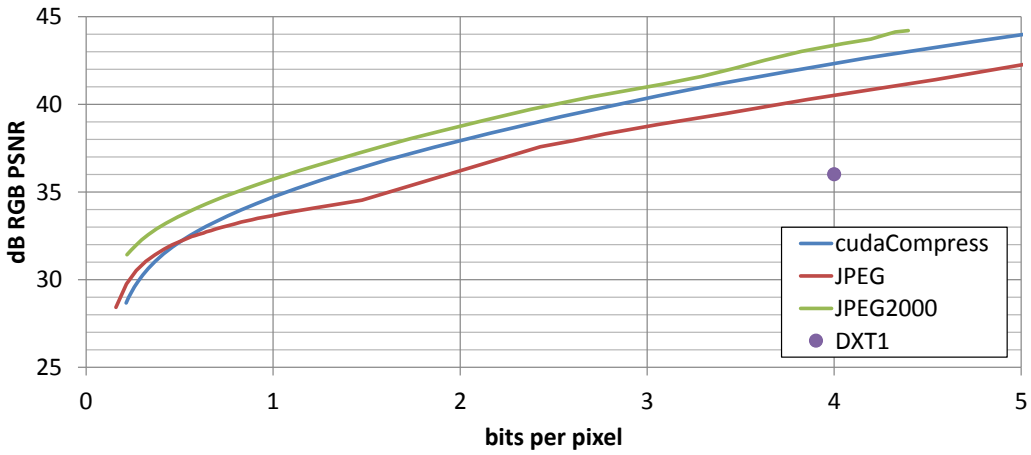

Figure 3.14: Graph of PSNR vs. bpp for the test image suite newtest.

4 Interactive Terrain Editing

As a first application of the presented compression methods, I present an interactive terrain rendering and editing system [TRAW12]. Previous terrain rendering approaches have addressed the aspect of data compression and fast decoding for rendering, but applications where the terrain is repeatedly modified and needs to be buffered on disk have not been considered so far. Such applications require both decoding and encoding to be faster than disk transfer, which can be achieved with the cudaCompress library. I present a novel approach for editing gigasample terrain fields at interactive rates and high quality. The construction and rendering of a height field triangulation is avoided by using GPU ray-casting directly on the regular grid underlying the compression scheme. I demonstrate the efficiency of the method for interactive editing and continuous level-of-detail rendering of terrain fields comprised of several hundreds of gigasamples.

4.1 Introduction

Today, high-resolution terrain fields consisting of many billions of color and height samples are available, and a number of techniques exist to render such fields efficiently. For an overview of the different approaches underlying these techniques let me refer to the survey by Pajarola and Gobbetti [PG07]. To avoid bandwidth limitations due to disk transfer and to reduce the number of rendered primitives, height field compression such as adaptive triangulation or differential vertex encoding has

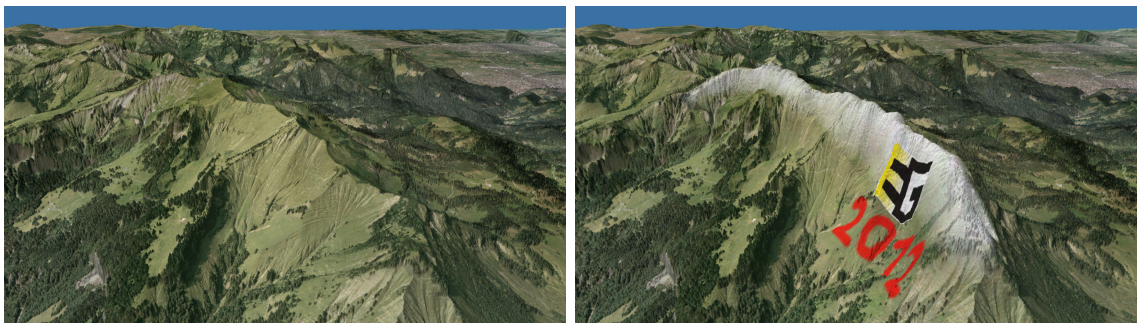

Figure 4.1: A terrain field of over 300 gigasamples (left). Direct editing using a paint and displacement brush (right) and simultaneous rendering of the resulting changes is performed at 60 fps on a 1920×1080 viewport.

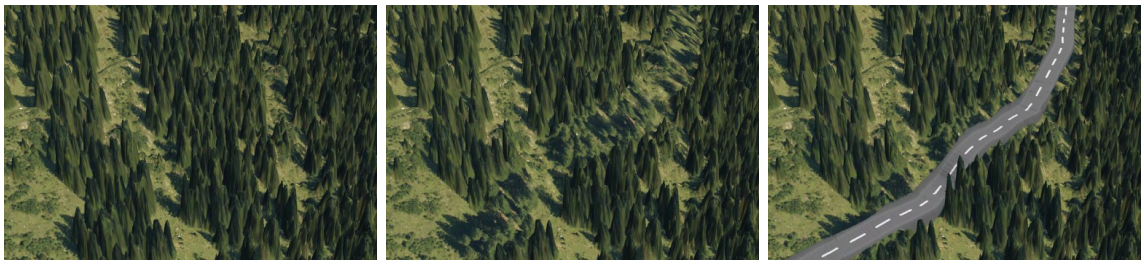

Figure 4.2: Creating a street: A forest (left) is cleared along a path (middle) to build a street (right).

been incorporated into terrain rendering approaches. The orthophoto used to texture the height field is typically compressed using the fixed-rate compression format S3TC. The requirement to decode the compressed data at very high rates has played a major role in the selection of compression schemes for terrain fields.

There is also an increasing interest in techniques that allow editing terrain fields interactively, including applications ranging from game level design and virtual world modeling to geographic planning and geological simulation systems. Since the modified data needs to be buffered in disk memory so that it can be displayed and modified again at a later time, both decoding *and encoding* have to be faster than disk transfer. Current compression schemes for terrain fields are problematic in such applications because the construction of the compressed data representation requires extensive preprocessing. Even though medium-quality S3TC compressors come at the required throughput [vWC07], no such encoder has been reported for height maps

or high-quality compression. Thus, existing terrain editing approaches have focused on alterations of uncompressed terrain, not taking into account disk I/O bandwidth limitations. To the best of my knowledge, interactive visually-guided editing of terrain fields so large that they require compression has not been achieved before.

Contribution: I present a novel approach to interactively edit terrain fields which are so large that I/O bandwidth becomes the major bottleneck (see Fig. 4.1 for an example). To handle such fields efficiently, I employ the cudaCompress library introduced in Section 3.6 for the compression of both color and height fields.. Special emphasis has been put on efficiently combining level-of-detail editing and rendering. To accomplish this, the internal data representation is based on pixel and height field raster data, and rendering is performed directly on these rasters using ray-casting. The particular contributions are

- a high-throughput GPU coder which can encode and decode up to 290 and 1070 MPix/s, respectively, at compression rates similar to JPEG2000,
- a push-pull error compensation scheme to avoid the propagation of quantization errors between resolution levels,
- a progressive and view-dependent update scheme for editing operations to avoid latencies due to bandwidth limitations, and
- a prototype system that demonstrates interactive editing and rendering of large terrain fields comprised of more than 300 gigasamples.

The remainder of this chapter is structured as follows: In the next section I review previous work on terrain editing. Section 4.3 gives an overview of the different parts my system is comprised of and outlines their interplay. Following in Section 4.4 is a brief discussion of the GPU compression scheme, making use of the cudaCompress library. Next, in Section 4.5, I analyze the compression rates, the reconstruction quality, and the coding performance that cudaCompress achieves in this application, and I demonstrate the efficient interplay between data compression and rendering in a prototype terrain editing system. The chapter is concluded in Section 4.6 with some ideas for future enhancements.

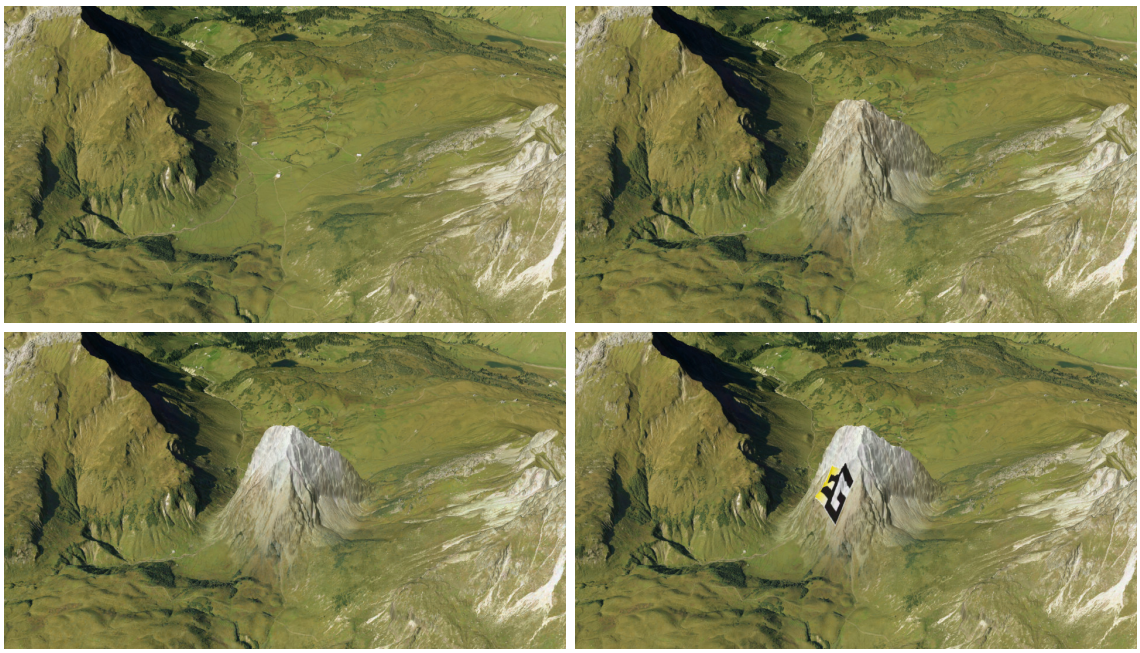

Figure 4.3: Placing a mountain using a height stamp, painting snow on top, and using a color stamp to add the EG logo.

4.2 Related Work

Terrain rendering approaches usually incorporate some form of height field compression to reduce limitations in disk and CPU-GPU bandwidth as well as the number of rendered polygons. Pajarola and Gobbetti [PG07] discuss the basic principles underlying many of these techniques, and many others [BGMP07, GMC*06, BGP09, DSW09, LC10] provide specific details on customized compression schemes.

On the other hand, only few approaches have been reported for interactive terrain editing, where the internal data representation is continually modified. He et al. [HCP02] perform the editing operations on a regular height map and create an adaptive triangulation on-the-fly. The efficient construction of an error-controlled mesh hierarchy from a regular height map on the GPU has been demonstrated by Lambers and Kolb [LK10]. Ammann et al. [AGD10] also edit a regular height map, but avoid constructing a height field triangulation and perform ray-casting directly on this map. Brandstetter et al. [BIMW*10] perform edits at coarser resolution levels

and discard finer details in the edited regions. None of these approaches, however, has considered the propagation of changes between different resolution levels.
Atlan and Garland [AG06] edit the coefficients of a Haar wavelet transform on the CPU. Thus, for some simple editing operations the propagation of changes is not required. Bhattacharjee et al. [BPN08] apply the editing operations directly on the GPU, but then also perform the propagation of changes on the CPU. In both cases, the finest resolution level has to be available in CPU memory.
Bruneton and Neyret [BN08] propose a method for efficiently embedding vector features into the height field by adapting a uniform height field triangulation. Terrain orthophotos are generated procedurally by an appearance shader and, thus, streaming of high-resolution color data to the GPU is not required. Furthermore, updated appearance and elevation maps at finer levels are always created on-the-fly once they become visible. Thus, once these maps get paged out of GPU memory, they have to be re-created when the user comes back to the respective terrain region. This is significantly different to my approach.

For multi-resolution editing, I use concepts similar to those proposed by Perlin and Velho [PV95] for multi-resolution pixel image editing using wavelet transforms.

4.3 Gigasample Terrain Editing

I now give an overview of the different components of the prototype editing system as well as their interplay. In particular, I describe the internal data structure and the embedding of the compression scheme into the editing system.

The terrain editing system is intertwined with a visually continuous terrain renderer based on a tiled quadtree terrain representation, where 2×2 adjacent tiles on each level are exactly covered by one tile on the next-coarser level. Each tile represents the data on a uniform grid of size 1024^2, with the leaf nodes corresponding to the original data. My terrain representation is similar to the one proposed by Dick et al. [DSW09]. In my case, however, instead of storing the data at the according resolution, a tile at a particular quadtree level stores the compressed differences between this data and a low-pass filtered copy of it. To compute these differences, a discrete wavelet transform (DWT) is performed. At runtime, tiles within a spherical prefetching region around the camera are loaded from disk into CPU memory. The world-space radius of the

prefetching region is doubled with every coarser level.

4.3.1 Tile Tree Creation and Reconstruction

After the terrain field has been partitioned into a set of tiles, for each tile a node is created and the multi-resolution tile tree is constructed in a bottom-up procedure: One level of a DWT is computed on the data in each tile, which splits the data into a lower-resolution approximation and so-called detail coefficients. These coefficients encode the difference between the approximation and the original data. Only the detail information is stored at the nodes, and the lower-resolution approximations of 2×2 adjacent tiles are merged to form the data at a new parent node. This procedure is then repeated recursively until the tree has a user-defined depth. Finally, the detail coefficients at each node are encoded as described in Section 4.4, and the compressed data stream is stored to disk.

To reconstruct the data at a particular node, the tile tree is traversed along the path from the root to this node. At each node except the root, an inverse DWT using the detail coefficients at this node and the coarser approximation stored at the parent node is performed. The resulting data is the coarse approximation that is then used to reconstruct the data at the next child node. This process is repeated along the path until the selected node is reached.

4.3.2 Rendering

In each frame, the set of tiles required to render the current view is determined by traversing the tile tree in depth-first order. The traversal is stopped when the tile that is represented by the current node is completely outside of the view frustum, or if the maximum screen-space error when rendering the data of this tile falls below a user-defined threshold. A visited node is marked if the tile it represents is visible.

The tree is then traversed again as before, and the data at the marked nodes is reconstructed as described before. For nodes whose data is not already resident in GPU memory, the compressed data that is required to perform the reconstruction is streamed from disk to the GPU. On the GPU, decoding as well as the inverse DWT are performed and the reconstructed data is stored in a 2D buffer. Once the data for

a tile has been reconstructed on the GPU, it is always tried to keep this data on the GPU for as long as possible.

The reconstructed 2D raster data is rendered using a GPU ray-caster [DKW09] which performs a discrete traversal of the raster until a hit with the height field is determined. At this position, the tile's orthophoto is evaluated using anisotropic interpolation and the resulting color is used as the pixel color.

4.3.3 Editing

Editing is performed on the currently rendered tiles. Since all rendered data is stored in 2D buffers, the editing operations can be realized in a very efficient way. After each editing operation, the tile tree has to be updated to enforce the applied changes at all resolution levels. Here, I distinguish between the propagation of the applied changes upwards (to coarser resolution levels) or downwards (to finer resolution levels) in the tile tree (see Fig. 4.4). In addition, when the height field was modified, a 2D maximum mipmap which is used to accelerate the ray-casting process [DKW09] has to be recomputed for each affected tile.

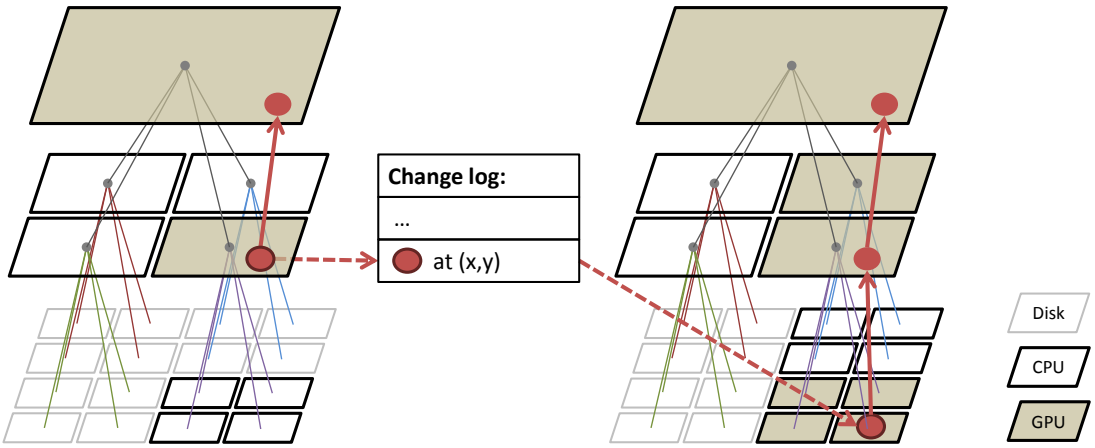

Figure 4.4: Propagation of changes. Left: The effect of an editing operation in a visible tile is immediately propagated to the tile's ancestors. The operation is stored in a change log. Right: When a finer tile is required, operations in the change log are applied to this tile, and the modifications are propagated to the ancestors again to ensure consistency.

The propagation to coarser resolution levels is performed instantly whenever an

editing operation is performed. It is simply realized by constructing the tile tree again as described before, but now starting the construction at the nodes storing those tiles that were affected. To avoid paging during the update operation, the system keeps the ancestors of all rendered tiles in GPU memory.

The propagation to finer levels is realized differently, since in general the finer tiles which are affected by an editing operation are not available on the GPU. Therefore, during editing all applied operations are recorded, i.e. the brush positions and action parameters. Once a finer tile is requested—either for upload to the GPU or for rendering—to which the changes have not yet been applied, the data is reconstructed and the editing operations are first applied before the data is rendered. By means of this delayed, on-demand propagation, the number of update operations that have to be applied at once is proportional to the number of requested tiles in the current frame, regardless of how many tiles in the tree are affected. Whenever such a delayed update is performed, the resulting changes have to be committed to the coarser levels again. This is necessary since applying an editing operation to a finer tile and downsampling the changes to a coarser tile is in general not identical to applying the operation directly to the coarser tile.

The update of modified tiles on disk is triggered via a backup interval, which is set to 250 ms in the current implementation. If no further editing operations occurred within one such interval, the tiles which have been altered are compressed and stored in CPU memory again, so that they can be removed from GPU memory when they are not needed for rendering anymore. When no CPU memory is available any more or a tile falls out of the prefetching region, the modified tiles are written to disk.

To edit the terrain height field and orthophoto my system provides several tools such as a paint brush and a stamp to draw color and/or height offsets (see Fig. 4.1 and 4.3), a flatten tool to smooth high-frequency details (see Fig. 4.5 for an application), and a special tool allowing a street to be drawn into the terrain along a user-defined spline curve as shown in Fig. 4.2. This variety shows that my approach is flexible enough to integrate more sophisticated terrain editing tools, for instance as proposed by de Carpentier and Bidarra [dCB09].

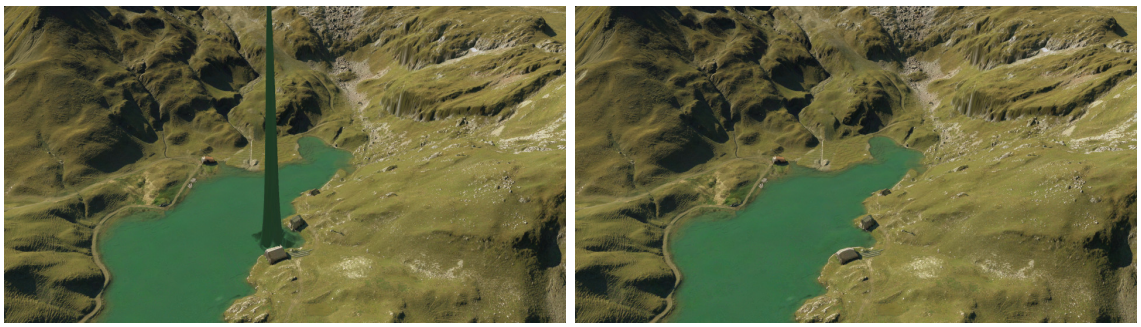

Figure 4.5: Interactive flattening to remove scanning artifacts.

4.4 Data Compression

In the following, I describe the four different stages the GPU coder is comprised of. I first discuss the compression of RGB pixel data, and then outline the particular changes to accommodate the processing of scalar-valued height fields. All these operations are implemented on top of the cudaCompress library.

Color space transform: The RGB color values of each tile's orthophoto are first transformed into the YCoCg color space [MS03b] to exploit correlations between the color channels. The YCoCg values are transformed back into RGB values only for display.

DWT: After color space conversion, a DWT is performed on the channels of the YCoCg pixel data separately using the CDF 9/7 wavelet [CDF92]. A multi-resolution pyramid is constructed in a bottom-up manner by repeatedly applying the DWT to the approximation coefficients at each level.

Quantization and push-pull: In a top-down manner, the floating-point detail coefficients C_i at the nodes of the tile tree are quantized into integer values c_i via standard scalar dead-zone quantization as $c_i = \text{sign}(C_i) \left\lfloor \frac{|C_i|}{\Delta} \right\rfloor$, where Δ is a user-defined quantization step.

To avoid propagating quantization errors from the coarser to the finer levels during reconstruction, I perform a push-pull error compensation (see Fig. 4.6). At every

node the difference is computed between the reconstructed signal from the parent node and the signal resulting from the DWT on the original data. The difference values are quantized, and they are encoded and stored in addition to the quantized detail coefficients. During reconstruction, these values are added to the low-pass coefficients at the parent node before the inverse DWT is performed. This ensures that the low-pass coefficients are of the same fidelity as the high-pass coefficients. Due to the recursive nature of the push-pull procedure, any remaining errors will be compensated at the next finer level.

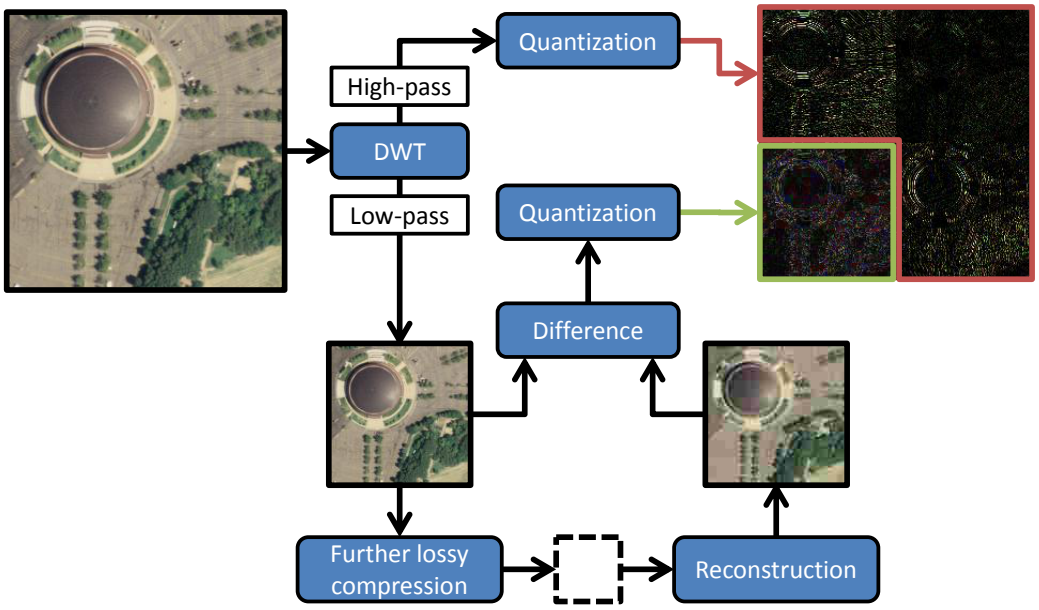

Figure 4.6: Push-pull error compensation: To avoid propagating errors from coarser to finer levels, the difference between the original low-pass coefficients and their reconstruction is stored in addition to the detail coefficients.

A different strategy to circumvent this problem is employed in JPEG2000, where coefficients at coarser levels are quantized using ever smaller quantization steps. This results in slightly better compression rates, but has the undesirable effect that the effective bit rate is increased at every coarser level. Due to the embedding properties of JPEG2000's EBCOT coder, this can be compensated by appropriately reordering the compressed bit stream, at the cost of some storage overhead and a much more complex coding scheme. My approach, on the other hand, allows the bit rate to

stay approximately constant over all levels without adding undue complexity in the decoder.

Coding: For encoding, the quantized wavelet coefficients are concatenated into a sequential stream in scan-line order. The coefficients are compressed using a run-length + Huffman coder.

Height field compression: For height field compression, besides not requiring a color space conversion, a maximum compression error should be guaranteed so that the screen-space error during rendering can be predicted. To achieve this, I first quantize the scalar height values such that the vertical resolution matches the resolution of the underlying sampling grid at the current level of detail, e.g. if height samples are taken at a 1 m spacing, then these samples are quantized such that the quantization intervals are 1 m as well. The quantized values are then transformed via a reversible integer DWT using the CDF 5/3 wavelet [CDF92] on the GPU. Difference encoding to avoid the propagation of quantization errors is performed in the same way as for color data.

4.5 Results

To demonstrate the efficiency of my terrain editing approach I have used a textured terrain height field of Vorarlberg, Austria. The orthophoto has a size of 447000×677000 pixels or about 300 gigapixels at a spatial resolution of 12.5 cm. The height field is given on a 2D grid with a spatial resolution of 1 m. All timings were performed on a PC with an Intel Xeon E5520 CPU (quad core, 2266 MHz), 12 GB of DDR3-1066 RAM, and an NVIDIA GeForce GTX 580 graphics card, except where explicitly noted otherwise.

4.5.1 Rendering and Editing

Rendering the terrain at a screen-space pixel error of 0.7 using GPU ray-casting takes between 15 and 20 ms per frame on a 1920×1080 viewport. Compared to rendering, the cost of applying an editing operation to the uniform height and color maps at a

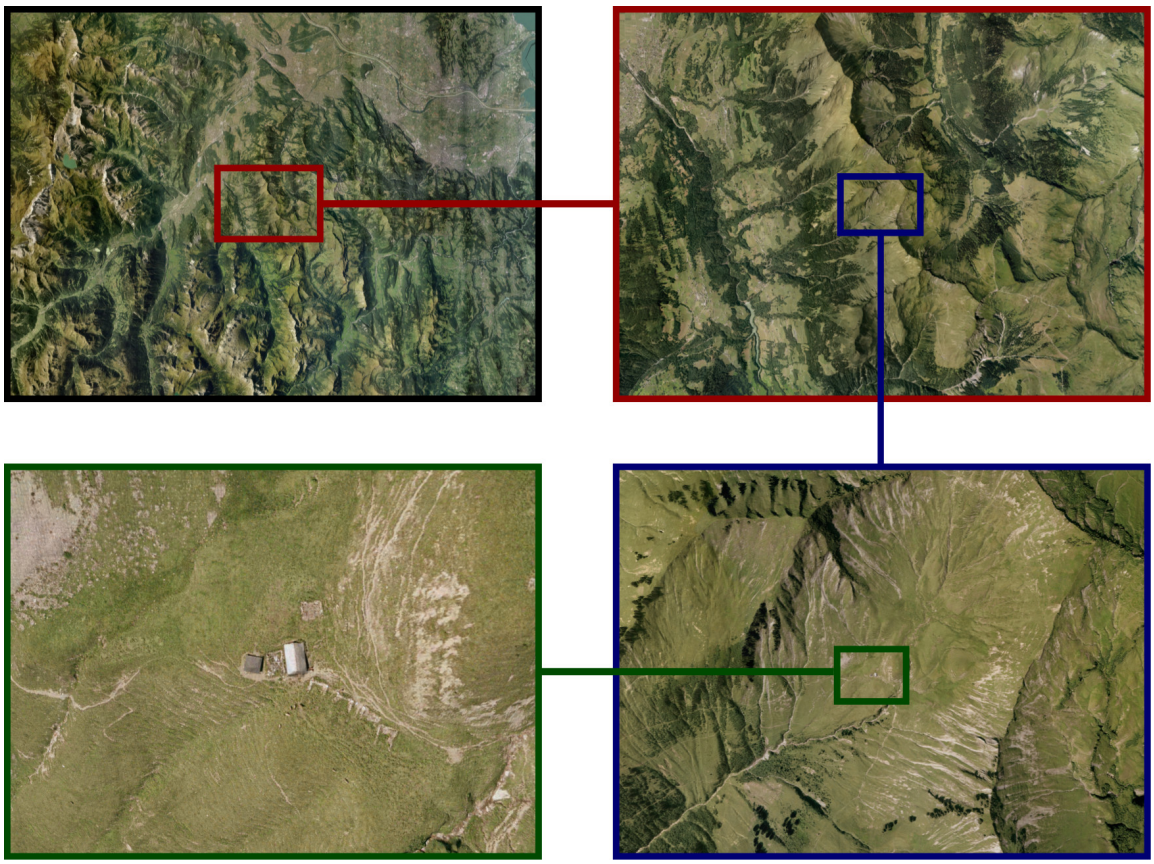

Figure 4.7: Zoom into the Vorarlberg data set.

particular level is negligible in general. Only when a very large part of the terrain is modified at once, or when many individual editing operations have been logged and have to be applied at once to update the data, does altering the respective maps become more costly than rendering. Since for editing purposes the height maps and orthophotos of all visible tiles including their ancestors need to be available on the GPU in uncompressed form, for higher-resolution viewports and a thereby increased number of tiles, the limited GPU memory can become a bottleneck. On the other hand, even in the current scenario where a very large terrain field is processed, all required data could always be stored in GPU memory and CPU-GPU bandwidth limitations were not observed.

After the editing operations have been applied at a particular level, the resulting

changes have to be committed into the tile tree. This requires computing a number of DWTs, encoding the resulting detail coefficients, and finally writing the updated tiles to disk. For instance, if one tile (height and color) on the finest level of a tile tree of depth 12 is modified, the DWTs take about 15 ms, encoding takes about 120 ms, and writing the data to disk takes about 60 ms.

4.5.2 Compression Rate and Quality

To assess the compression rate and reconstruction quality of cudaCompress applied to the terrain data, I have performed a number of tests using a set of 100 sub-images of the Vorarlberg orthophoto, each of 2048×2048 pixels. On each image a three-level DWT was performed, and the resulting coefficients were compressed as described.

I have compressed the same images using JPEG, JPEG2000, and the S3TC DXT1 format. For JPEG and JPEG2000 compression, I used the ImageMagick library v. 6.7.0 [Ima13]. The S3TC compression was performed using the Squish library v. 1.11 [Bro08] at the highest quality setting, iterative cluster fit. It is worth noting that JPEG and DXT1 do not support resolution-incremental decoding. In an application where this is required, the effective bit rates would thus be about 1.3 times higher than the given ones.

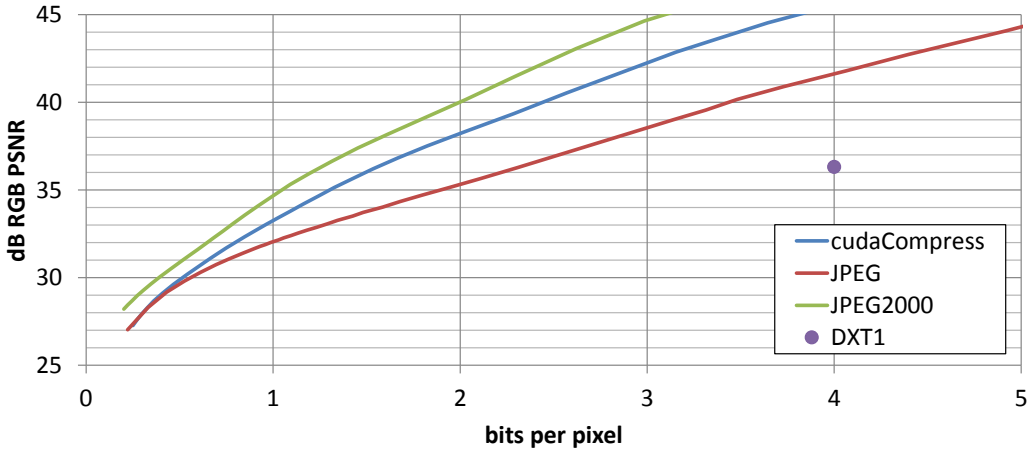

Figure 4.8: Graph of PSNR vs. bpp for 100 images of size 2048×2048 taken from the Vorarlberg orthophoto.

Fig. 4.8 shows the compression quality in dB of RGB PSNR depending on the bit

rate in bits per pixel (bpp). It can be seen that JPEG2000 gives the best results in terms of compression rate. The compression using cudaCompress outperforms JPEG, often significantly. However, at such low bit rates, neither algorithm can produce visually acceptable results. The fixed-rate DXT1 compression is clearly outperformed by all other approaches.

When compressing the entire Vorarlberg orthophoto, cudaCompress achieves 37.1 dB PSNR at 1.30 bpp, yielding a compression ratio of 18.4:1. For comparison, DXT1 achieves 36.4 dB at 5.33 bpp including mipmaps. The scalar height field stored as 16-bit integers was compressed at 1.54 bpp and a compression ratio of 10.4:1 by cudaCompress.

4.5.3 Compression Throughput

To produce realistic and robust performance numbers, I have measured the times required for encoding and decoding the entire Vorarlberg data set excluding disk I/O time. Encoding the 300 gigapixel orthophoto at 1.30 bpp using 11 DWT levels took 17.7 min, including the construction of the multi-resolution pyramid. This corresponds to a throughput of 290 MPix/s. Decoding took 4.7 min, giving a throughput of 1070 MPix/s.

Encoding the entire 4.9 gigasample height field using 8 DWT levels took 5.3 s at a 920 MPix/s throughput. Decoding took 2.1 s at a 2180 MPix/s throughput.

The encoding times include the download of the compressed data from the GPU, and the decoding times include the upload of the data to the GPU. Thus, the timings realistically reflect the performance that can be achieved when embedding the compression scheme into a terrain viewer, which streams compressed data from disk to the GPU, where it is decoded, displayed, modified, encoded again, and finally downloaded to the CPU and stored on disk.

For comparison, encoding to JPEG using the `tjbench` program from the libjpeg-turbo library v. 1.3.0 [lib13] at quality 90 and 4:4:4 chroma sampling achieves a throughput of 53 MPix/s on a single CPU core. The decoder achieves a throughput of 70 MPix/s. With 4:2:0 chroma sampling, the numbers improve to 80 MPix/s and 95 MPix/s, respectively. Extrapolating to four available CPU cores, the encoder throughput matches cudaCompress, but decoding is still significantly slower. It is

also worth noting here that the given performance measures for JPEG compression do not include building and compressing a multi-resolution pyramid. In the performance measures of cudaCompress, these operations are always included.

The Kakadu library v. 7.0 with speed pack [Kak13], one of the fastest JPEG2000 implementations, reports a throughput of up to 85 MPix/s for the encoding and 99 MPix/s for the decoding of a large image on a 2.2 GHz Intel Core i7-2720Q quad-core CPU. However, it is worth noting that the Kakadu software runs entirely on the CPU, so CPU-GPU bandwidth can become a bottleneck. CUJ2K [FWH*09], a CUDA implementation of a JPEG2000 encoder (but no decoder), achieves a throughput of only 22 MPix/s, excluding data transfer between CPU and GPU. Encoding of RGB pixel data to DXT1 using the NVTT GPU compressor [NVI10] achieves 21 MPix/s.

4.6 Conclusion

I have presented a prototype terrain editing system that allows altering and simultaneous rendering of high-resolution terrain fields at high quality and interactive rates. It employs a regular grid structure for the height field, and employs a GPU-based ray-caster for rendering. Both pixel and height data are compressed using the cudaCompress library, which provides both encoding and decoding at much higher rates than disk transfer and achieves compression rates that compare favorably to JPEG and JPEG2000 compression.

5

Turbulence Visualization: Volume Rendering

As the second application of GPU data compression, I present a system for the interactive exploration of very large turbulent flow fields [TBR*12]. Despite the ongoing efforts in turbulence research, the universal properties of the turbulence small-scale structure and the relationships between small- and large-scale turbulent motions are not yet fully understood. The visually guided exploration of turbulence features, including the interactive selection and simultaneous visualization of multiple features, can further progress our understanding of turbulence. Accomplishing this task for flow fields in which the full turbulence spectrum is well resolved is challenging on desktop computers. This is due to the extreme resolution of such fields, requiring memory and bandwidth capacities going beyond what is currently available. To overcome these limitations, I present a system for feature-based turbulence visualization that works on a compressed flow field representation. A GPU compression layer based on CUDACOMPRESS enables a drastic reduction of the data to be streamed from disk to GPU memory. The system derives turbulence properties directly from the velocity gradient tensor, and it either renders these properties in turn or generates and renders scalar feature volumes. The quality and efficiency of the system is demonstrated in the visualization of two unsteady turbulence simulations, each comprising a spatio-temporal resolution of 1024^4. On a desktop computer, the system can visualize each time step in 5 seconds, and it achieves about three times this rate for the visualization of a scalar feature volume.

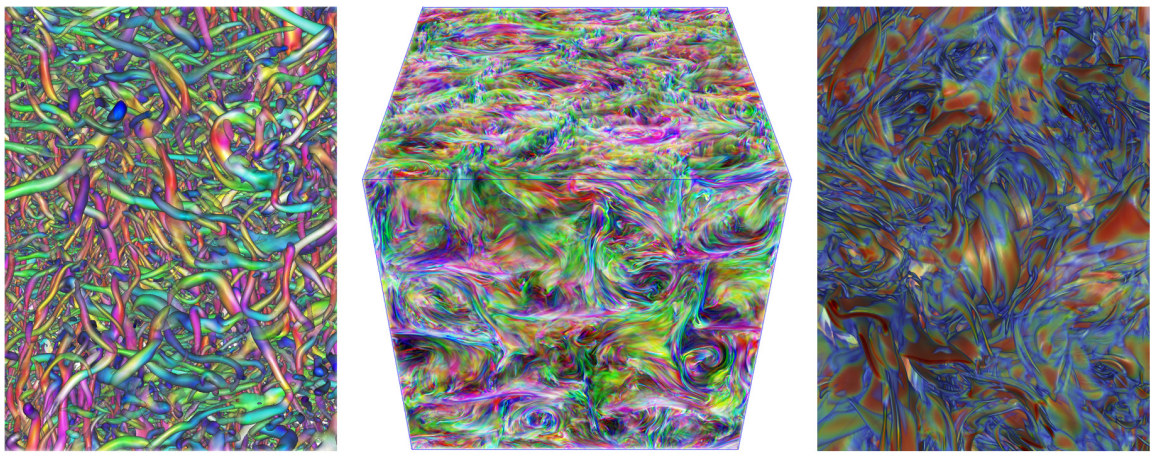

Figure 5.1: Visualizations of structures in 1024^3 turbulence data sets on 1024×1024 viewports, directly from the turbulent motion field. Left: Close-up of iso-surfaces of the Δ_{Chong} invariant with direct volume rendering of vorticity direction inside the vortex tubes. Middle: Direct volume rendering of color-coded vorticity direction. Right: Close-up of direct volume rendering of R_S. The visualizations are generated by my system in less than 5 seconds on a desktop PC equipped with 12 GB of main memory and an NVIDIA GeForce GTX 580 graphics card with 1.5 GB of video memory.

5.1 Introduction

Hydrodynamic turbulence is one of the most thoroughly explored phenomena among complex multiscale physical systems. It has important applications in engineering thermo-fluid systems, in the geosciences and environmental transport, even in astrophysics. In recent years, high performance computing [IGK09] and new experimental measurement techniques [KS10, WV10] applied to the study of various types of turbulent flows have enabled significant progress. Yet, modeling and understanding turbulent flows remains a scientifically deep, technologically relevant, but fundamentally unsolved problem.

One grand challenge that significantly increases the complexity of turbulence analysis is turbulence's inherently vectorial and tensorial structure: one describes turbulent flows using velocity and vorticity vector fields, and velocity gradient and stress tensor fields. Some of the most salient features of turbulent flows have emerged from

an examination of the velocity gradient tensor. It is defined according to

$$A_{ij} = \frac{\partial u_i}{\partial x_j},$$

where I use index notation; $u_i(\mathbf{x}, t),\ i = 1, 2, 3$ denote the three components of the velocity vector field which depend on the position vector $\mathbf{x}$ and time t. Such gradient fields of fluid velocity provide a rich characterization of the local quantitative and qualitative behavior of flows, which is evident from the linear approximation in the neighborhood of an arbitrary point. Since $\mathbf{A}$ is a second-rank tensor, it has nine components in 3D and these contain rich information about the local properties of the flow. Since the tensor $\mathbf{A}$ encodes much information through each of its matrix elements, analysis of its properties is quite challenging. Therefore, certain scalar quantities that characterize basic properties of $\mathbf{A}$ have been proposed and are often analyzed as scalar fields, e.g. the vorticity magnitude, the dissipation rate, the angle between vorticity and the strain-rate eigenvectors, or the magnitude of the rotation tensor, to name just a few.

One of the primary challenges in turbulence research is to endow the traditional statistical analysis of metrics with more geometrical insights into the overall structure of turbulence affecting more than one specific property. Even though a number of different feature metrics are known, no single feature can alone explain all relevant effects. This means that different features must be explored simultaneously and in an interactive fashion, to be seen in relation to each other. Only then can one proceed with evaluating more meaningful statistical metrics. For example, one would like to visualize the high vorticity, high rotation, or high Q regions in the flow, but in relation with the alignments of the local strain-rate eigen-directions, or together with another scalar field such as R. Particularly the question whether the geometric trends in the small-scale turbulence structures are also shared by the coarse-grained (or filtered) velocity gradient tensor plays a determining role in turbulence research. A visual indication of the relationship between velocity increments and the filtered velocity gradients at coarser scales can enable further insights into the complicated multiscale behavior of turbulence.

The visual exploration of many different intrinsic features of turbulence, however, is very challenging. The major reason is that the fine-scale structures are fully resolved

only at the very highest resolution in both space and time. For instance, I address the visualization of two terascale turbulence simulations, each comprised of one thousand time steps of size 1024^3, making every time step as large as 12 GB at 3 floating-point values per velocity sample. These data sets contain direct numerical simulations of forced isotropic turbulence (see Fig. 5.1, left) and magneto-hydrodynamic turbulence (see Fig. 5.1, middle and right), respectively. For a detailed descriptions of the simulation and database methods used let us refer to the work of Li et al. [LPW*08] and the web page at `http://turbulence.pha.jhu.edu`. For such data it is simply not feasible to precompute multiple feature volumes and inspect these volumes simultaneously, in particular because the number of potentially interesting features and scales is so large. Furthermore, to be able to faithfully represent even the smallest features in the data, highly accurate reconstruction schemes are necessary which work directly on the turbulence field by reconstructing features during visualization.

As a consequence, visualization systems necessary to explore the full turbulence spectrum require an innovative approach that provides extreme I/O capabilities, combined with computational resources that allow for an efficient feature reconstruction and rendering. Since the data to be visualized is so large that even storing one single time step in CPU memory can become problematic, bandwidth limitations in paging the data from disk become a major bottleneck.

Following the requirements in turbulence visualization, I have developed a new holistic approach which combines scalable data streaming and feature-based visualization with novel hardware and software solutions, such as a deep integration of GPU computing. I employ the capabilities of GPU-based data compression using cudaCompress to reduce memory access and bandwidth limitations. Because my approach reduces disk access and CPU-GPU data transfer, it is suitable for the analysis of small-scale turbulence structures on desktop systems which are not equipped with large main memory. To preserve even the finest structures, feature extraction is embedded into the visualization process, based on the direct computation of vector field derivatives and on-the-fly evaluation of feature metrics based on the gradient tensor.

My system distinguishes from previous approaches for visualizing turbulent flow fields in that it eases bandwidth and memory limitations throughout the entire visualization pipeline. In particular, the system

- compresses vector data at very high fidelity,
- works on the compressed data up to the GPU, using on-the-fly GPU decompression and rendering,
- enables caching of derived feature volumes via on-the-fly GPU compression, and
- provides multiscale feature visualization via on-the-fly gradient tensor evaluation.

The remainder of this chapter is structured in the following way: First, I review previous systems and algorithms for the visualization of large volumetric data sets. In Section 5.3, I then give an overview of the system, including its internal structuring as well as the basic functionality in a nutshell. The aim is describing what the system provides and how this is achieved, but not to answer the question why the particular choices have been made. This question is addressed in the upcoming Section 5.4 where I motivate the design decisions and discuss trade-offs involved in making the system practical for visualizing large turbulence simulations. This also involves the demonstration of some advanced features which are made possible by these choices. I describe the streaming and visualization performance of the system and discuss its preprocessing costs in Section 5.5. Finally, I conclude the chapter in Section 5.6 with some ideas for future enhancements and extensions of the system.

5.2 Related Work

Previous efforts in large volume visualization can be classified into two major categories: a) Parallelization and b) data compression and out-of-core strategies. There is a vast body of literature on parallelization strategies for volume visualization on parallel computer architectures and a comprehensive review is beyond the scope of this work; however, some of the most recent works have addressed volume rendering on both GPU [FCS*10, MAWM11] and CPU [HBC10] clusters.

A different avenue of research has addressed the visualization of large volumetric data on desktop PCs. These works employ out-of-core techniques to dynamically load only the required part of a data set into memory, and many employ some form of compression to reduce the immense data volume. The most recent works focussing

on the direct rendering of large-scale volume data [CNLE09, GMI08, FSK13] employ an octree of volume bricks. During rendering, the octree is traversed on the GPU and visited nodes are tagged for refinement or coarsening. The tags are read back to the CPU which then updates the GPU working set accordingly. Such approaches allow for on-demand streaming of data and efficient rendering in a single pass, provided that all data required for the current view is available in GPU memory. In turbulent flow fields, however, using a lower-resolution approximation of the velocity data results in a significant distortion of the extracted features and is thus not admissible.

5.3 System Functionality, Algorithms, and Features

My approach begins with a sequence of 3D turbulent motion fields, each given on a Cartesian grid. In a preprocess, each vector field is partitioned into a set of equally sized bricks. An overlap between adjacent bricks enables proper interpolation at brick boundaries. Every brick is compressed separately and written to disk. The preprocess is outlined in Fig. 5.2.

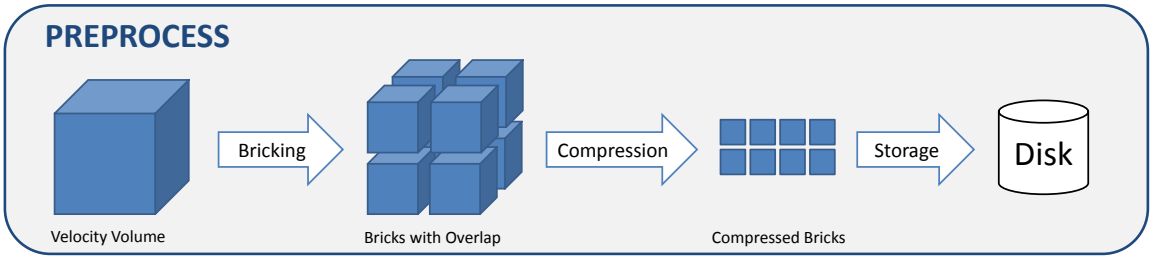

Figure 5.2: Preprocessing pipeline.

5.3.1 Compression Algorithm

Once the bricked volume representation has been constructed, each brick is compressed using cudaCompress. A two-level DWT is performed separately on each component of the velocity vectors using the CDF 9/7 wavelet [CDF92]. The floating-point wavelet coefficients C_i are quantized into integer values c_i via standard scalar dead-zone quantization, i.e. $c_i = \text{sign}(C_i)\,\lfloor |C_i|/\Delta_l \rfloor$, where Δ_l is the quantization step that is used at level l of the wavelet pyramid. Because the coefficients at coarser

scales carry more energy than the coefficients at smaller scales, the quantization steps are decreased with increasing scale, i.e. starting at the finest level $l = 0$ with a user-defined step size Δ_0, on subsequent levels the step size is set to $\Delta_l = \Delta_0 \ / \ (2\sqrt{2})^l$. Here, Δ_0 provides control over the compression ratio and reconstruction quality. The quantized wavelet coefficients are finally concatenated into a sequential coefficient stream in scan-line order. Finally, run-length encoding followed by a Huffman encoding converts the coefficient stream into a highly compact form.

5.3.2 Visualization Algorithm

For visualizing the 3D vector field, my system uses GPU ray-casting on the bricked volume representation [KW03]. At every sample point along a ray, one or multiple scalar features are derived from the velocity gradient tensor (see Section 5.3.3), and the respective values are mapped to color and opacity. The velocity gradient tensor is computed on-the-fly via central differences between interpolated velocity values. The system supports trilinear interpolation for fast previewing purposes and tricubic interpolation [SH05, RtHRS08] for high-quality visualization. The visual difference between trilinear and tricubic interpolation is demonstrated in Fig. 5.4. For shading purposes, gradients are approximated locally by central differences on six additional feature samples.

The bricks are traversed on the CPU in front-to-back order. If the compressed representation of the current brick is not residing in CPU memory, it is loaded from disk and cached in RAM using a LRU strategy. After all data for the current time step has been loaded, the CPU starts prefetching subsequent time steps asynchronously. The brick is then streamed to the GPU, where a CUDA kernel is executed to decompress the data and store it in texture memory. My system leverages GPU texture memory to take advantage of hardware-supported texture filtering. Streaming to the GPU works asynchronously, meaning that the transmission stalls neither the CPU nor the GPU.

Once the data has been decompressed, a CUDA ray-casting kernel is launched. It invokes one thread for every pixel covered by the screen-aligned rectangle enclosing the projected vertices of the brick's bounding box. Each thread first determines if the respective view ray intersects the bounding box and terminates if no hit was detected.

Otherwise, the current frame buffer content at the pixel position is read, and the brick data is re-sampled along the ray to obtain the color and opacity contribution to be accumulated with the current values. Early ray termination is performed whenever the opacity has reached a value of 0.99. Fig. 5.3 illustrates the basic system data flow at runtime.

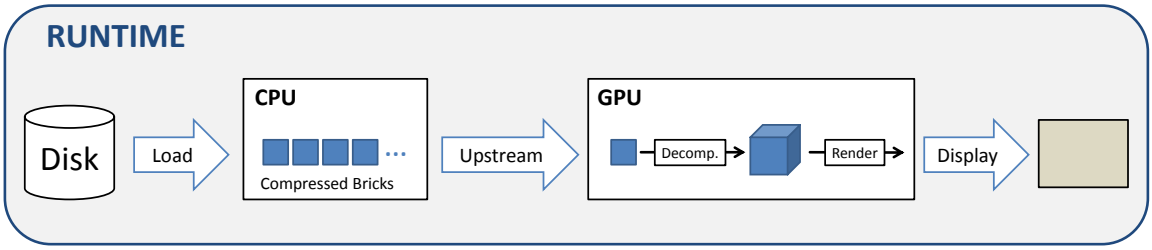

Figure 5.3: Basic system data flow at runtime.

Based on a set of basic rendering modalities, i.e. direct volume rendering (DVR) including iso-surface rendering and scale-invariant volume rendering [Kra05], my system supports a number of different visualization options, for example, the simultaneous rendering of iso-surfaces of multiple turbulence features, or a combination of different techniques such as DVR and iso-surface rendering. Furthermore, a comparative visualization of the same feature at different scales is supported by enabling simultaneous operations on the initial and filtered data (see Section 5.4.3). The visualization of features can also be made dependent on the existence or properties of other features. Some examples of different visualization options are shown in Figs. 5.1 and 5.5.

5.3.3 Turbulence Features

In my system, turbulence features are derived from the velocity gradient tensor. The gradient fields of fluid velocity provide a rich characterization of the local quantitative and qualitative behavior of flows, which is evident from the linear approximation in the neighborhood of an arbitrary point,

$$\mathbf{x}_0 : u_i(\mathbf{x}, t) = u_i(\mathbf{x}_0, t) + A_{ij}(\mathbf{x}_0, t)(x_j - x_{0j}) + \dots .$$

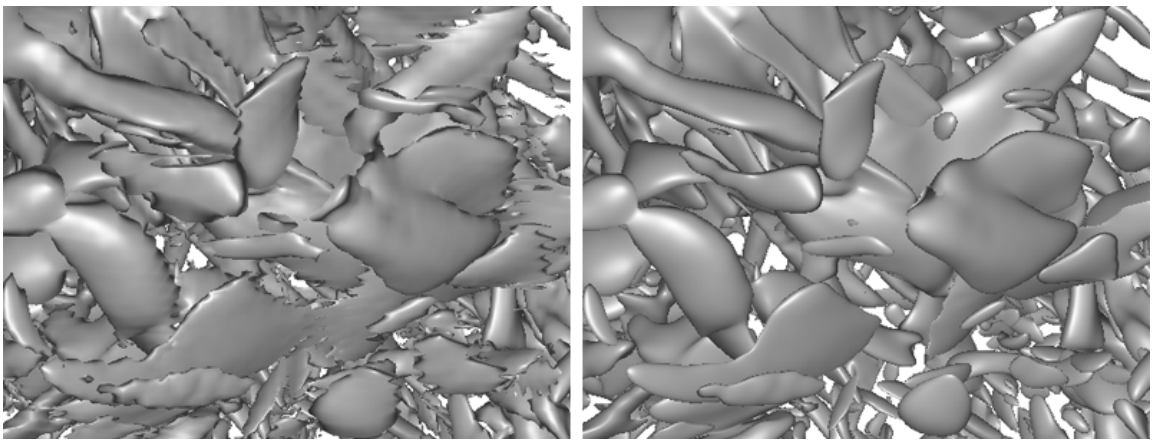

Figure 5.4: Comparison of trilinear (left) vs. tricubic (right) filtering when rendering iso-surfaces. With trilinear interpolation, the silhouettes of high-frequent iso-surfaces are poorly resolved.

Since $\mathbf{A}$ is a second-rank tensor, it has nine components in 3D and these contain rich information about the local properties of the flow. The decomposition

$$A_{ij} = S_{ij} + \Omega_{ij}, \quad \text{where} \quad S_{ij} = \frac{1}{2}\left(A_{ij} + A_{ji}\right), \quad \Omega_{ij} = \frac{1}{2}\left(A_{ij} - A_{ji}\right),$$

is commonplace and separates $\mathbf{A}$ into its symmetric part, the strain-rate tensor $\mathbf{S}$, and its antisymmetric part, the rotation-rate tensor $\mathbf{\Omega}$. The tensor $\mathbf{S}$ has three real eigenvalues λ_i that in incompressible flow add up to zero. In the non-degenerate case when they are different, the tensor $\mathbf{S}$ has three orthogonal eigenvectors that define the principal axes of $\mathbf{S}$. These indicate directions of maximum rate of fluid extension ($\lambda_\alpha > 0$) and contraction ($\lambda_\gamma < 0$), and an intermediate fluid deformation that can be either extending or contracting in the third direction. $\mathbf{\Omega}$ describes the magnitude and direction of the rate of rotation of fluid elements and is simply related to the vorticity vector $\boldsymbol{\omega} = \boldsymbol{\nabla} \times \mathbf{u}$. Since the tensor $\mathbf{A}$ encodes much information through each of its matrix elements, an analysis of its properties is quite challenging. Therefore, certain scalar quantities that characterize basic properties of $\mathbf{A}$ have been proposed and are often analyzed as scalar fields.

For example, it has been found convenient to define the following five scalar invari-

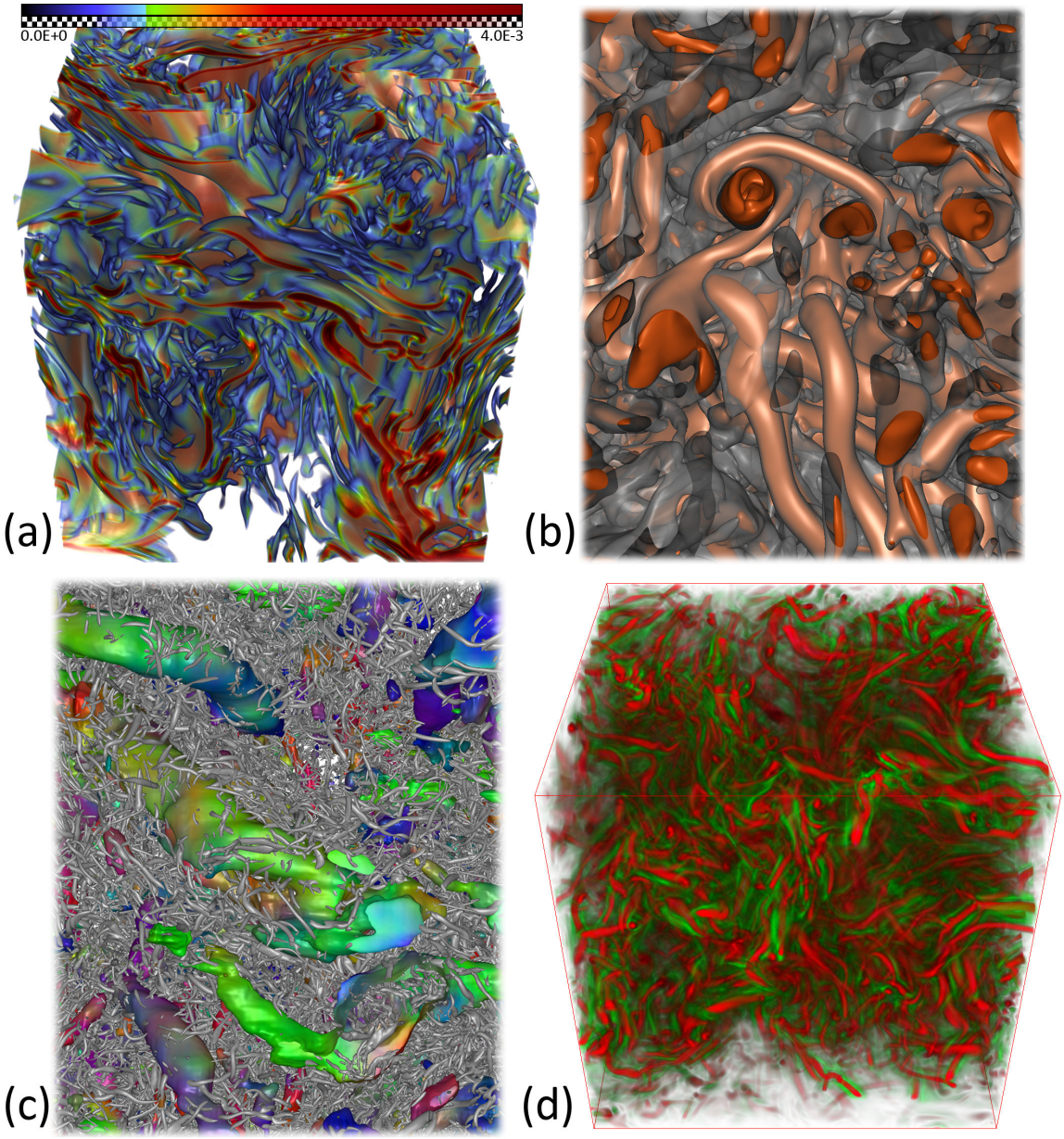

Figure 5.5: Turbulence visualizations. (a) Direct volume rendering of E. (b) Two semi-transparent iso-surfaces of Q_{Hunt}. (c) Fine-scale iso-surfaces (gray) and coarse-scale iso-surfaces colored by vorticity direction. (d) Direct volume rendering of λ_2; negative values are red, positive values green.

ants [Can92, MOCS98]:

$$Q = -\frac{1}{2}\text{Trace}(\mathbf{A}^2) = -\frac{1}{2}A_{ij}\ A_{ji} := -\frac{1}{2}\sum_{i=1}^{3}\ \sum_{j=1}^{3} A_{ij}\ A_{ji},$$

$$R = -\frac{1}{3}A_{ij}\ A_{jk}\ A_{ki},$$

$$Q_S = -\frac{1}{2}S_{ij}S_{ji}, \quad R_S = -\frac{1}{3}S_{ij}S_{jk}S_{ki}, \quad V^2 = S_{ij}S_{ik}\omega_j\omega_k.$$

Additional commonplace, Galilean invariant vortex definitions involve non-trivial combinations of $\mathbf{A}$, $\mathbf{S}$ and $\mathbf{\Omega}$, such as the Q_{Hunt} and Δ_{Chong} criteria [CPC90, Hal05, HWM88]:

$$Q_{\text{Hunt}} = \frac{1}{2}\left(|\mathbf{\Omega}|^2 - |\mathbf{S}|^2\right) > 0, \quad \Delta_{\text{Chong}} = \left(\frac{Q_{\text{Hunt}}}{3}\right)^3 + \left(\frac{\det \mathbf{A}}{2}\right)^2 > 0.$$

Further vortex classifications employ additional information from the vorticity, or eigenvalues through an eigendecomposition of symmetric tensors. For example, the λ_2 criterion [JH95] identifies vortex regions by $\lambda_2 < 0$, where λ_2 is the second largest eigenvalue of the symmetric tensor $\mathbf{S}^2 + \mathbf{\Omega}^2$. Another option is the enstrophy production, which is defined as $E = S_{ij}\omega_i\omega_j$.

One striking observation in turbulence research was the preferential vorticity alignment found by Ashurst et al. [AKKG87]. They observed that the most likely alignment of the vorticity vector $\boldsymbol{\omega}$ was with the intermediate eigenvector $\boldsymbol{\beta}_S$, the direction corresponding to the eigenvalue λ_β that could be either positive or negative. For a random structureless gradient field, no such preferred alignment would be expected, and on naïve grounds one might have expected the vorticity to align with the most extensive straining direction instead. Therefore, the observations generated sustained interest in the problem of alignment properties of the vorticity field and relationships with features related to the strain-rate tensor—e.g. its eigenvectors' directions. For this reason, my system provides mappings of the vector components of $\boldsymbol{\omega}$ as well as the eigenvectors $\alpha_{\mathbf{S}}, \beta_{\mathbf{S}}, \gamma_{\mathbf{S}}$ of the strain-rate tensor to RGB colors during volume ray-casting.

Besides analyzing the small-scale turbulence structures, a substantial amount of

research has been devoted to the statistical features of velocity increments in the inertial range [Fri95, SA97]. In particular, it has been shown that a relationship between fine-grained velocity gradients and coarse-grained or filtered velocity gradients can be established. The coarse-grained gradients are computed by a convolution kernel, usually an averaging box filter. To enable a visual multiscale analysis of turbulence, my system allows simultaneously extracting and visualizing features from filtered velocity fields at different user-selected scales. As filtering and differentiation are linear operations, filtering is performed on the velocity vector field instead of the gradient tensor field.

5.4 Design Decisions and Tradeoffs

I now consider some of the decisions made in the implementation of the system that make it suitable for visualizing large turbulence data. In particular, I want to emphasize the possible trade-offs that allow the user to choose between highest quality and highest speed. Despite the careful design of the system with regard to the application-specific requirements, not always can it respond interactively to the user inputs. This is because of the extreme amounts of data to be processed and the complex shaders to be evaluated. However, the results demonstrate a system performance that facilitates an interactive exploration for most of the supported visualization options.

The bricked data representation the system builds upon is necessary to keep the chunks of data that are processed at runtime manageable. In addition, the bricked representation has the advantage of enabling view frustum culling, resulting in a considerable reduction of the data to be streamed to the GPU. The integration of occlusion culling is also possible, but the turbulence structures are typically so small and scattered that no significant gain can be expected. I also want to mention that level-of-detail rendering strategies as they are typically employed in volume ray-casting have not been considered, because the continuous transition between multiple scales of turbulence in one image has been determined inappropriate by turbulence researchers.

To avoid access to neighboring bricks in trilinear/tricubic data interpolation and gradient computation, a 4-voxel-wide overlap is stored around each brick. Thus, the

smaller the bricks, the more additional memory is required to store the overlaps. Additionally, the larger the bricks, the fewer disk seek operations have to be performed for reading the bricks from disk. Consequently, the bricks are as large as possible, with the constraint that at least four decompressed bricks should fit into GPU memory as a working set. The system uses a brick size of 248^3, so that a brick including the overlap can be stored in a texture of size 256^3. This results in a memory overhead of about 10%.

5.4.1 Feature Reconstruction

The visualization system by default reconstructs turbulence features directly from the velocity field during ray-casting. It is also possible to precompute scalar feature volumes in a preprocess and to visualize these volumes. However, such an approach is problematic in the current scenario. First, it would cause a significant increase of the overall memory consumption. Second, the system would become inflexible to the extent of the precomputed features, prohibiting an interactive steering of the feature extraction processes. Third, quality losses are introduced by re-sampling a scalar feature volume instead of a direct feature reconstruction.

Two examples demonstrating the quality differences are shown in Fig. 5.6. The images clearly reveal that certain fine-scale structures can no longer be reconstructed from the scalar feature volumes. Even though the principal shapes are still maintained, a detailed analysis of the bending, stretching, merging, and separating behavior of the turbulence features is no longer possible.

On the other hand, ray-casting a feature volume can be a viable approach to obtain an overview of the turbulence structures. Therefore, my system supports the construction and storage of scalar feature volumes for fast previewing purposes (see also Section 5.5). Even though the construction of such a volume on the GPU is straightforward using the system's functionality, this volume might be too large to be stored on the GPU in uncompressed form. The requirement to tackle this problem has significantly steered the selection of the compression scheme.

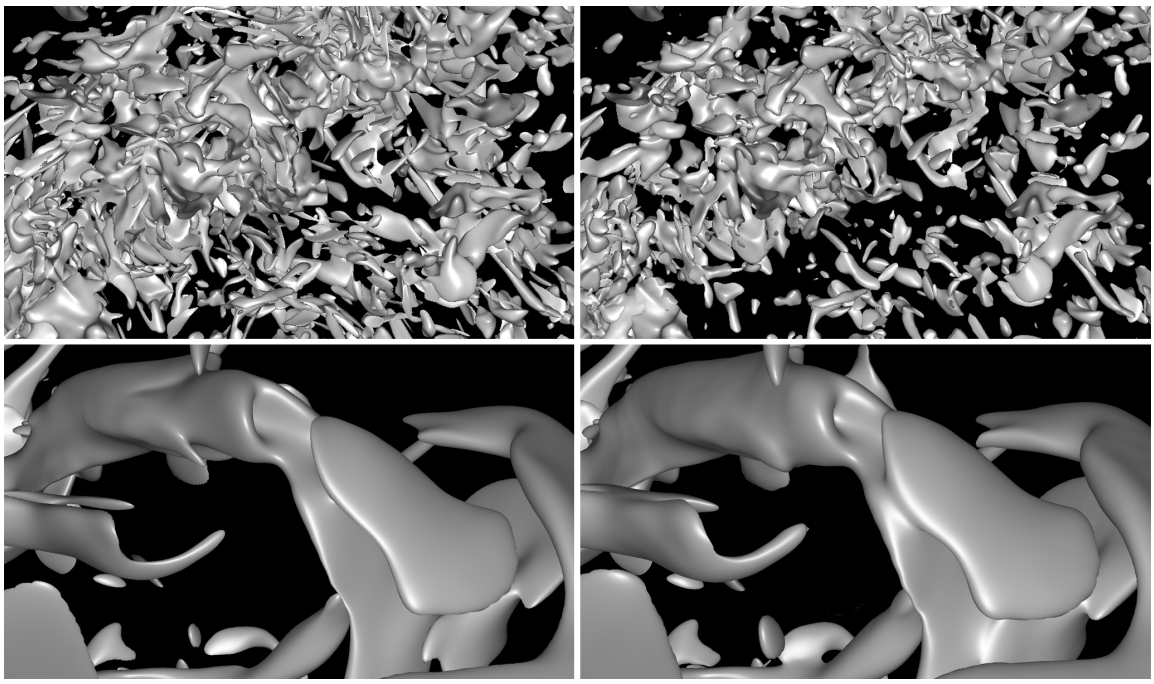

Figure 5.6: Features reconstructed from the turbulent motion field (left) and from a precomputed scalar feature volume (right).

5.4.2 Lossy Compression

Because of the extreme data volumes to be handled, the reduction of this volume becomes one of the most important requirements. Without any reduction, expensive disk-to-CPU data transfer becomes the major performance bottleneck since only a very small portion of an entire turbulence sequence can be stored in main memory. To meet this requirement, the system incorporates a data compression layer.

In the decision which compression to use, the following aspects have been considered. First of all, it is required that no features will be destroyed due to the compression, and that possible quality losses do not affect the features' shapes significantly. For floating-point data, compression schemes like S3TC and vector quantization [SW03] do not adhere to this constraint. Second, the compression ratio must be so high that the data can be streamed from disk fast enough to keep pace with the data processing speed. Especially due to this requirement, lossless compression schemes are problematic. In general, lossless schemes [BR09, LI06] can only achieve

a rather moderate compression ratio.

A last consideration arises from the particular requirement of my system to generate scalar feature volumes on-the-fly for the purpose of fast previewing. This functionality goes hand in hand with the possibility to efficiently compress the generated feature volumes on the GPU, so that they can be efficiently streamed to the CPU and buffered in RAM. While the compressed volumes could also be stored on the GPU, the time required to transfer the compressed data between the CPU and the GPU is negligible compared to the compression time. Thus, all generated data is always buffered on the CPU. To support this option, an alternative data flow as illustrated in Fig. 5.7 is realized in my system. This rules out compression schemes such as vector quantization, because the construction of the vector codebook in the coding phase can not be performed at sufficient rates in general.

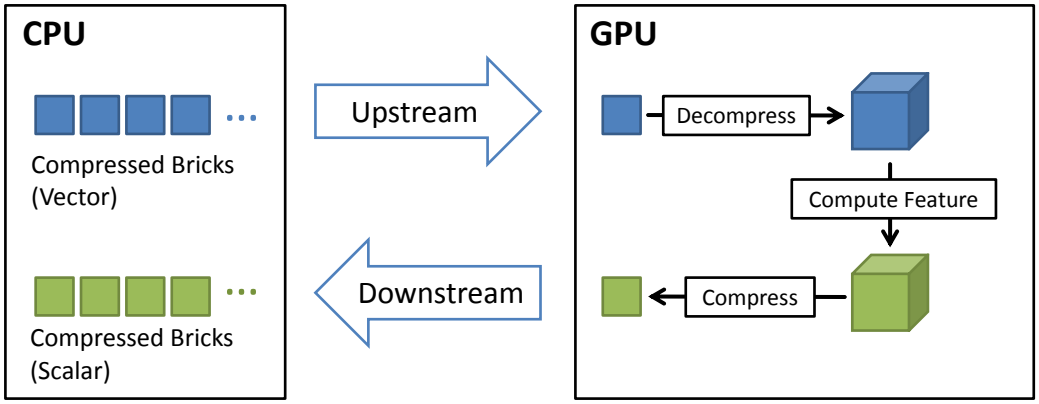

Figure 5.7: Alternative data flows at runtime support construction and re-use of derived feature volumes.

Lossy compression schemes based on the discrete wavelet transform, in combination with coefficient quantization and entropy coding, are well known to achieve very high compression rates at high fidelity [TM01]. Compression schemes based on transform coding also have a long tradition in visualization, for instance to reduce memory and bandwidth limitations in volume visualization [GWGS02, NS01, Wes94, YL95]. However, only with the possibility to perform the entire compression pipeline on the GPU [TRAW12]—including encoding *and* decoding—can the full potential of wavelet-based compression be employed for large data visualization.

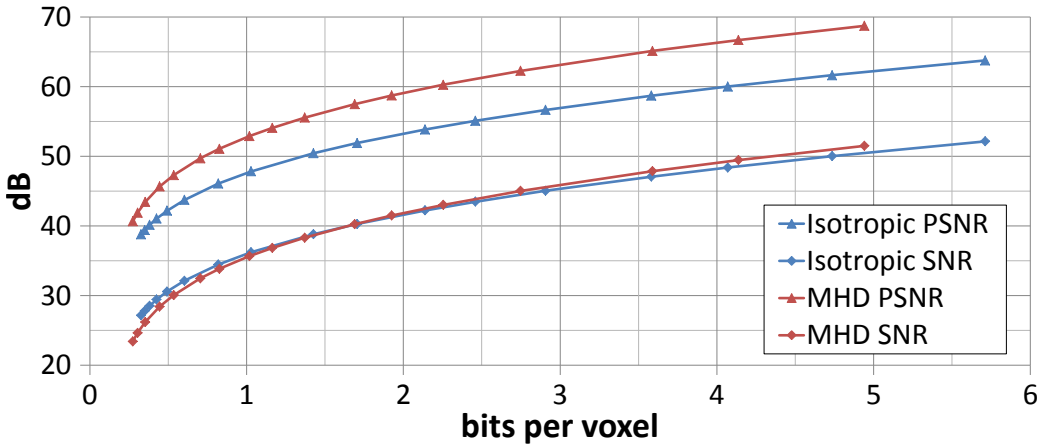

Figure 5.8: Rate-distortion curves showing dB (P)SNR vs. bits per voxel for two different turbulence fields.

To assess the compression ratio and reconstruction quality of the wavelet-based GPU coder, I have performed tests using two different turbulence simulations, each consisting of time steps of size 1024^3. On each brick a two-level DWT was performed, and the wavelet coefficients were compressed as described. Rate-distortion curves in (P)SNR vs. bits per voxel (where each voxel contains a 3-component floating-point vector) for both data sets are given in Fig. 5.8. In addition, Fig. 5.9 plots RMS error vs. quantization step as well as maximum error vs. RMS error. The graphs demonstrate that the user can directly control the compression error by choosing an appropriate quantization step size. The rate-distortion curves demonstrate the high reconstruction quality of the wavelet-based compression. On the other hand, they do not provide an intuitive notion of the visual quality. Therefore, Figs. 5.10 and 5.11 compare the visual quality of the rendered structures in the compressed motion fields to those in the original fields. For comparison purposes, different compression rates were used.

Even though it is clear that the compression quality depends on the visualization parameters, such as the transfer function, the selected iso-value, and on the feature that is visualized, a component-wise wavelet transform can very effectively reduce the memory consumption, yet it achieves a very high reconstruction quality. In particular, in the middle images in Figs. 5.10 and 5.11 the structures are reproduced

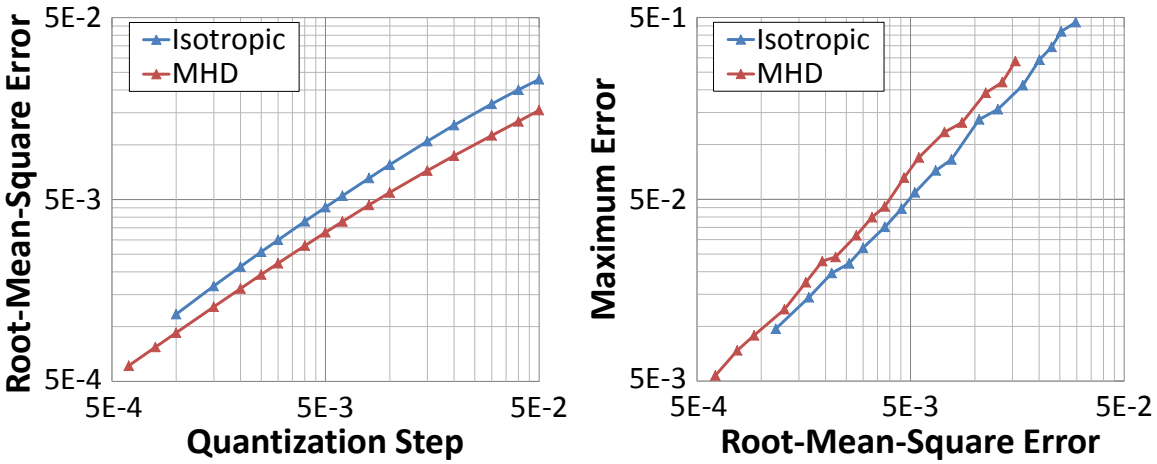

Figure 5.9: Graphs showing RMSE vs. quantization step, and maximum error vs. RMSE for two different turbulence fields. RMSE and quantization step are of the same order of magnitude, while the maximum error is consistently about 1 order of magnitude larger than RMSE.

at almost no visual difference at a remarkable compression ratio of 32:1.

Let us finally mention that the selected compression scheme can in principle be extended to also exploit the motion coherence between successive time steps for increasing the compression ratio. The basic idea is to not compress every time step separately, but rather to encode the differences between a time step and one or more preceding or succeeding time steps. Popular video compression algorithms such as MPEG typically employ block-based motion compensation, and this approach has been applied to volume compression as well [GS01]. However, any temporal prediction scheme obviously requires access to one or more other time steps to use as input. If these time steps can not be held in RAM in uncompressed form, then such a prediction necessarily triggers additional decompression steps and thus significantly increases the processing time. In this application, the possible gain in compression ratio does not justify the increase in compression/decompression time. Therefore, my system compresses each time step separately.

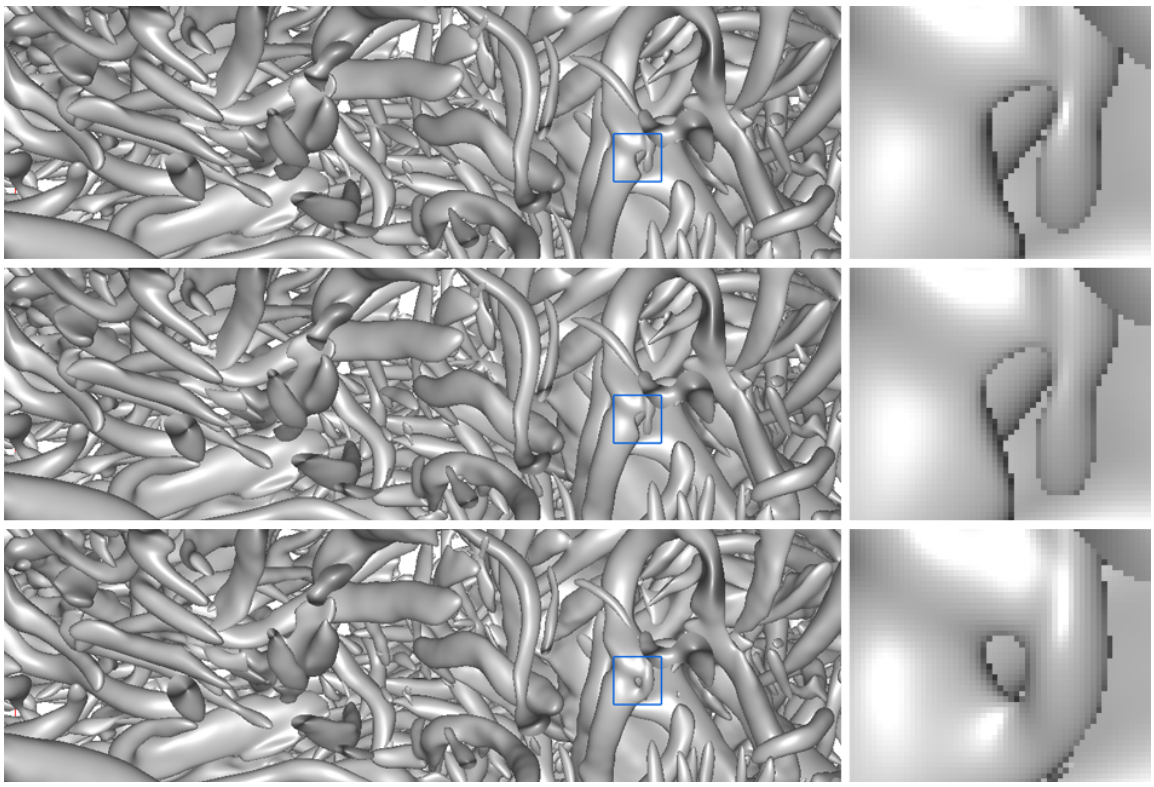

Figure 5.10: Visual quality comparison for an iso-surface in Q_{Hunt} in the isotropic turbulence data set. Structures are reconstructed from the original vector field (top), and a compressed version at 3.0 bpv (middle) and 1.3 bpv (bottom).

5.4.3 Multiscale Analysis

One particularly challenging endeavor in turbulence research is the analysis of the shape and evolution of structures at different scales. To enable such an analysis, it is necessary to filter the velocity field using a low-pass filter. A linear convolution filter is used in practice. It can then be instructive to compare the original data with the filtered version, or multiple instances filtered with different radii. Both the initial and the filtered data are made accessible to the shader, and they are ray-cast simultaneously. Many options for combined visualizations are now possible such as iso-surfaces for different iso-values and with different colorings (Fig. 5.5 (c)), or the conditional visualization of fine features depending on coarse features (Fig. 5.12).

Because a large range of scales may contain relevant features, these scales can not

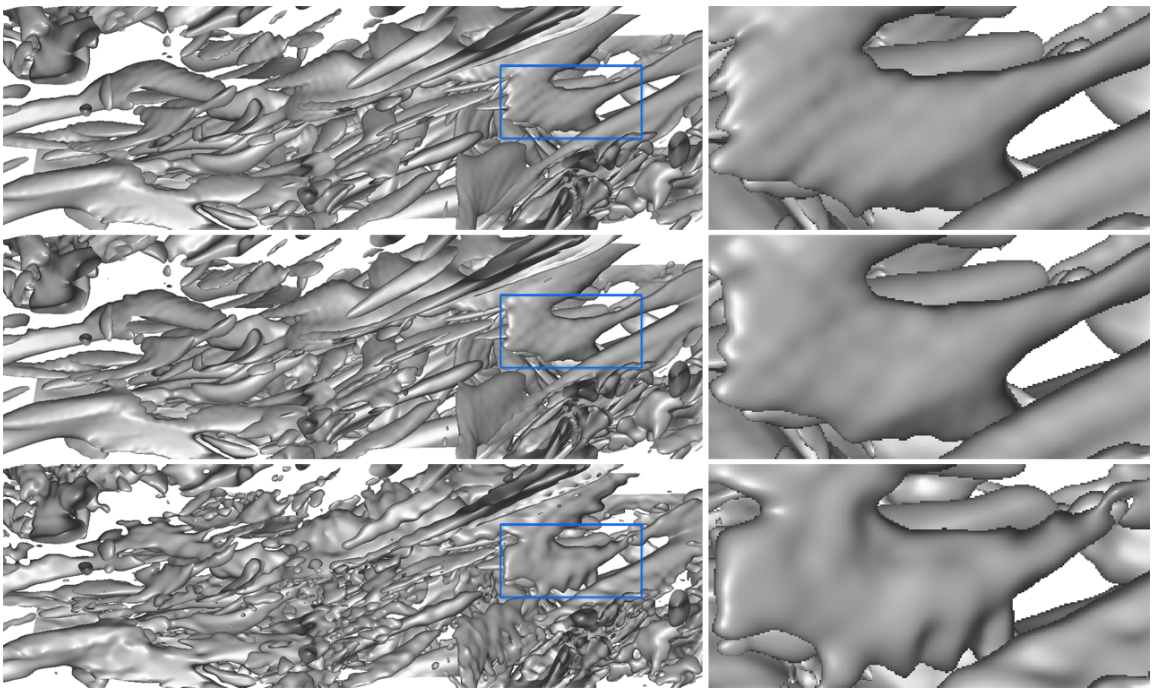

Figure 5.11: Visual quality comparison for an iso-surface in R_S in the MHD turbulence data set. Structures are reconstructed from the original vector field (top), and a compressed version at 3.0 bpv (middle) and 1.3 bpv (bottom).

be precomputed but must be determined interactively at runtime. Furthermore, the interesting scales are often quite large, requiring filter radii of 20 and more voxels, so that an on-the-fly filtering during rendering is not feasible. The large filter radii also make a separate filtering of each brick impossible, because values from adjacent bricks are required in the convolution. Unfortunately it is also impossible to keep all 26 neighbors of one brick in GPU memory at the same time, such that the bricks required for filtering have to be streamed and accessed sequentially. As a consequence, every brick needs to be loaded from the CPU and decompressed multiple times.

To avoid this, I have restricted my system to the execution of separable filters, i.e. filters that can be expressed as three successive 1D filters, one along each coordinate axis. In this case, only three filtering passes are required, and in each pass only the two neighbors along the current filter direction need to be available. In each pass, the bricks are traversed in an order which ensures that each brick needs to be

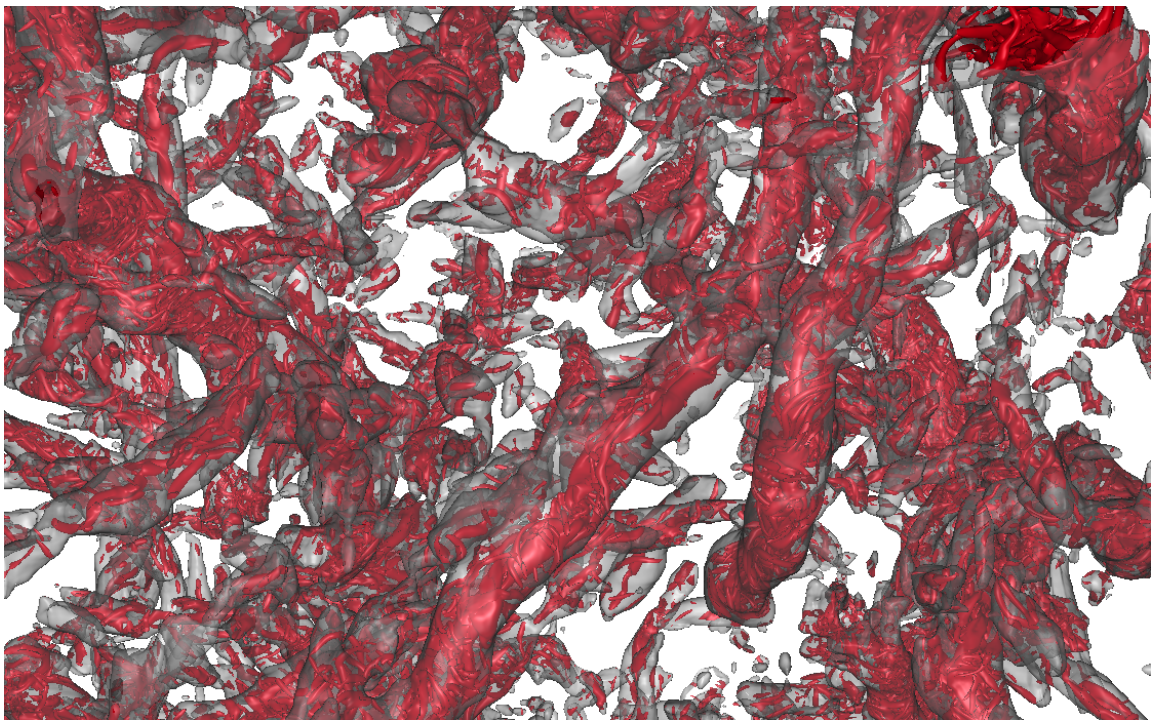

Figure 5.12: Multiscale turbulence analysis in a "focus+context" manner. Iso-surfaces in the fine-scale data (red) are extracted only within iso-surfaces of the coarse-scale version.

decompressed only once. Fig. 5.13 illustrates this ordering.

All intermediate results and the final filtered result are compressed on the GPU and buffered in CPU memory. Consequently, additional losses will occur besides those which are introduced by the encoding of the initial vector field. On the other hand, because the data becomes smoother and smoother after each filtering pass, at the same compression ratio ever better reconstruction quality can be achieved. In all of the examples, the additional losses were only very minor, and noticeable differences between single-pass and multi-pass filtering on the compressed vector field could hardly be observed. This is demonstrated in Fig. 5.14 for a particular feature iso-surface in the isotropic turbulence data set.

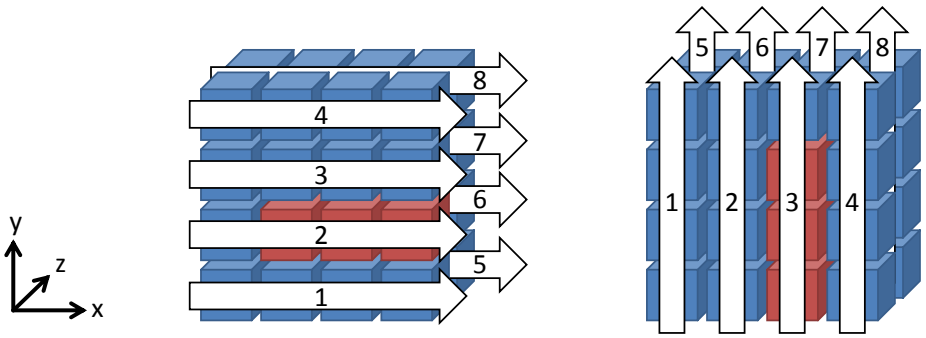

Figure 5.13: Brick ordering during separable filtering in the x and y dimension (left and right, respectively). The z dimension is analogous. One exemplary working set during each filtering pass is indicated in red.

5.5 Performance

In this section, I evaluate the performance of all components of my system and provide accumulated timings for the most time-consuming operations. All presented timings were performed on a desktop PC with an Intel Xeon E5520 CPU (quad core, 2266 MHz), 12 GB of DDR3-1066 RAM, an NVIDIA GeForce GTX 580 graphics card with 1.5 GB of video memory, and a standard hard disk providing a sustained data rate of about 100 MB/s.

The statistics are based on the two terascale turbulence simulations referenced in Section 5.1 and shown in Fig. 5.1. Each comprises one thousand turbulent motion fields of size 1024^3. The vector samples are stored as 3 floating-point values. Both data sets were compressed to 3.0 bpv at a compression ratio of 32:1. The compression ratios for individual bricks ranged from 53:1 to 22:1, depending on their content. The performance of all operations in my system scales roughly linearly in the spatial resolution of the data, so the system performance for data sets of different sizes can be easily extrapolated from the numbers listed below.

Although preprocessing time is not as important as visualization time, it is still significant for the practical visualization of very large data sets. On the test hardware, preprocessing takes about 3 minutes per time step. The majority of this time is spent reading the uncompressed data from disk; the compression of one time step—once stored in CPU memory—takes only about 10 seconds. This time includes the upload

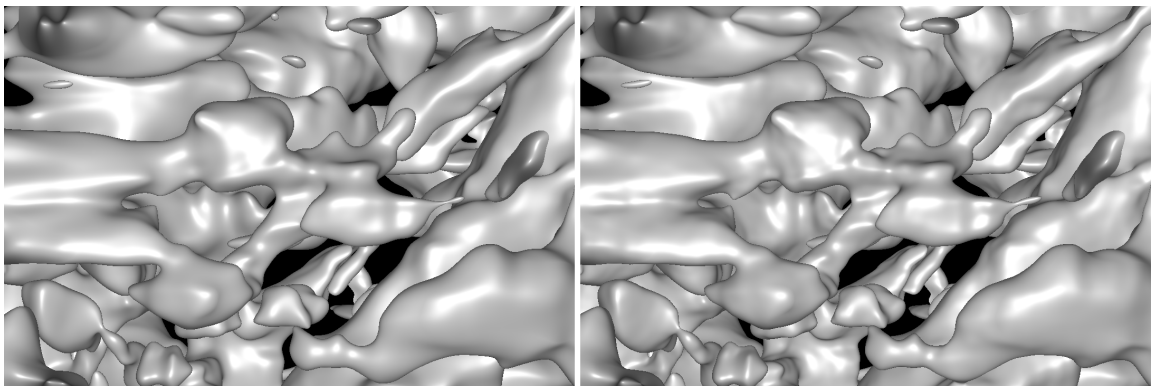

Figure 5.14: λ_2 iso-surface in an isotropic turbulence simulation. Left: A 3D smoothing filter with support 25 was applied to the compressed vector field in one pass. Right: The same filter as before was used, but now the filter was separated and filtering was performed in 3 passes. After each pass, the intermediate results were compressed and buffered on the CPU.

of the raw data to the GPU, the compression on the GPU, and the download of the compressed data to the CPU.

Table 5.1 summarizes typical timings for visualizing one compressed brick of size 256^3 and one compressed time step consisting of 5^3 bricks. I give separate times for data streaming and compression, performing data operations on the GPU, and rendering. For comparison, the statistics also include timings for an uncompressed data set. Rendering was always to a 1024×1024 viewport. The entire volume was shown so that view-frustum culling did not have any effect—in zoomed-in views where some bricks can be culled, decompression and ray-casting are faster accordingly. Furthermore, high transparency was assigned to the structures to eliminate any effects of early ray termination. Which feature metric was used had no significant effect on the performance. Since the visualization performance for different time steps and for the two different data sets was very similar, they are not listed separately in the table.

One can see that the upload and decompression of all bricks of one time step takes 3.0 seconds. This is slightly faster than the upload of the uncompressed data, which takes 3.2 seconds at a throughput of 4.2 GB/s over PCI-E. It has to be mentioned, however, that the decompression on the GPU blocks the GPU so that no other tasks can be performed. When uploading uncompressed data, the data transfer

Table 5.1: Timings for individual system components. Where appropriate, values are given as min–avg–max.

Data streaming	per brick	per time step
Read from disk	35–60–88 ms	4.2 s
Upload & decompress (vector-valued)	18–32–39 ms	3.0 s
Compress & download (vector-valued)	48–105–230 ms	10.3 s
Upload & decompress (scalar)	7–14–16 ms	1.3 s
Compress & download (scalar)	23–43–56 ms	4.2 s

GPU processing	per brick	per time step
Compute metric	28 ms	2.1 s
Filter (per pass)	3.9 ms	0.3 s

Rendering	per brick	per time step
Ray-cast, on-the-fly feature, trilinear	6–25–35 ms	2.1 s
Ray-cast, on-the-fly feature, tricubic	27–150–205 ms	12.1 s
Ray-cast, multiscale, tricubic	54–305–415 ms	24.4 s
Ray-cast, precomp. feature, trilinear	0.8–2.3–3.9 ms	0.2 s
Ray-cast, precomp. feature, tricubic	1.7–6.5–10 ms	0.6 s

Data streaming (uncompressed)	per brick	per time step
Read from disk	1.9 s	2.3 min
Upload or download (vector-valued)	34 ms	3.2 s
Upload or download (scalar)	11 ms	1.1 s

Figure 5.15: Visualizations of forced isotropic turbulence. (a) Direct volume rendering of the velocity magnitude in the whole data set. Images (b-f) show a closeup on a 256^3 subregion at the center of the simulation domain. (b) Velocity magnitude. (c) Vorticity magnitude. Images (d-f) show iso-surfaces of invariants of the velocity gradient tensor (from left to right: Q_{Hunt}, Q_Ω, Q_S). In all three images, iso-surfaces of a value equal to 1.0E-2 are shown.

Table 5.2: Aggregate timings for some common scenarios.

Scenario	startup	per frame
Preview rendering (precomp. feature, trilinear)	9.3 s	1.5 s
Standard rendering (on-the-fly feature, trilinear)	0.0 s	4.9 s
HQ rendering (on-the-fly feature, tricubic)	0.0 s	15.1 s
HQ rendering w/ multiscale analysis	40.8 s	29.8 s

could be performed in parallel with other GPU tasks, such as rendering. Thus, operations on uncompressed data are usually slightly faster than the operations on the compressed data, but only if the data is already available in CPU memory. This can not be assumed in general, e.g. when stepping through multiple time steps, or when multiple (e.g. filtered) volumes are required simultaneously (see Sections 5.3.2 and 5.4.3). Whenever disk access becomes necessary, working on the compressed data becomes significantly faster. Reading a single compressed time step from disk takes only about 4.2 seconds. Additionally, reading can be performed concurrently with decompression and rendering, and, thus, it can usually be hidden completely. In contrast, reading an uncompressed time step from disk takes about 2.3 minutes.

The display rates that my system achieves can not be compared to those reported for ray-casting of large scalar fields [CNLE09, GMI08], even though volume ray-casting on the bricked velocity field representation is used. This is because a) classical volume ray-casting systems make use of level-of-detail rendering, which is not admissible in this application, and b) exploit empty space skipping, which has no effect with data sets that do not contain empty space. It is also worth mentioning that my system executes much more complex shaders for feature reconstruction. Here, high-quality visualization using on-the-fly feature extraction and tricubic interpolation takes about 15.1 seconds, of which only 3 seconds are required for decompressing the velocity field and 12.1 seconds are required for ray-casting. The reason lies in the extremely complex shaders on the velocity gradient tensor, which are evaluated at every sample point to evaluate the selected feature metrics. As an alternative, a preview-quality visualization of a precomputed scalar feature volume using trilinear interpolation takes about 1.5 seconds. The computation and compression of a scalar

feature volume from a compressed vector field requires about 9.3 seconds. If the whole feature volume can be stored on the GPU, e.g. on a Quadro or Tesla card with 4-6 GB of video memory, rendering takes only 0.2 seconds. Since in this case the feature volume does not need to be compressed, it can be generated on the GPU in about 5.1 seconds.

Filtering of the 3D velocity field is a very expensive operation. It requires the entire data set to be uploaded to the GPU, decompressed, filtered, and compressed. All this must be done once per dimension. Even though the raw compute time to filter the data on the GPU is only 0.3 seconds per filtering pass for a filter with a support of 51, it takes about 40 seconds until the result is available in CPU memory. This time is vastly dominated by the GPU decompression, and in particular the GPU compression of the intermediate and final results. Compression is about three times as expensive as decompression because the run-length and Huffman encoders are more complex than the respective decoders [TRAW12]. In particular, Huffman encoding requires a round-trip to the CPU, where the Huffman table is constructed. Table 5.2 summarizes the startup and per-frame time required for some common scenarios as outlined above. These times do not include the times required to load the data from disk because disk access is performed concurrently with rendering and processing and can usually be hidden.

From the measured timing statistics it becomes clear that for the given data sets my system can not achieve fully interactive display rates. The reason is that the system was developed with the intent of visualizing extremely large turbulence data sets. This requires complex shaders for feature extraction to provide the significant amounts of fine detail at the smallest scale. However, the memory-efficient design of the system makes it well suited for implementation on desktop machines.

5.6 Conclusion

In this chapter, I have presented a system for the exploration of very large and time-dependent turbulence data on desktop PCs. The interactive exploration of terascale data with limited available memory and bandwidth is made possible by the integration of a GPU-based compression scheme. The cudaCompress library provides fast GPU implementations of both compression and decompression which are necessary

for fast streaming of velocity data and the efficient storage of derived data. The very high quality of the compression ensures the faithful preservation of turbulence features. A preview mode in the renderer based on precomputed feature volumes allows the interactive navigation to features of interest at only slightly reduced quality. On the other hand, the high-quality rendering of time-dependent image sequences is accelerated by an order of magnitude compared to the use of uncompressed data.

By using my system, a turbulence researcher can interactively explore terascale data sets and tune visualization parameters. For instance, a 70-year-old result on magnetic "flux-freezing" could recently be disproved using interactive visualization of fully resolved turbulence [EVL*13]. Additionally, a trend has been discovered with the help of my system which had not been observed before: In multiscale visualizations such as Fig. 5.12, the large-scale vortices contain small-scale vortices that appear to form helical bundles within the large-scale vortices. These visualizations suggest new statistical measures such as the alignment angle between large- and small-scale vorticity, to be implemented in future research as a result of the present observations.

While the current system is tailored for desktop PC systems, I believe that many of the presented techniques also have applications in supercomputing. When moving to the petascale, numerical simulations at unprecedented resolution and complexity will become possible, going beyond even the present turbulence data sets. Although the raw compute power of separate visualization computers keeps pace with those of supercomputers, bandwidth and memory issues in networking and file storage present significant restrictions. For visualizing the peta- or even exabytes of data we will be confronted with, writing the raw data to disk or moving it across the network has to be avoided. A promising direction for future research is the integration of a compression layer similar to the one used here, which could alleviate bandwidth limitations between compute nodes and the visualization system.

6 Turbulence Visualization: Particle Tracing

As the third and final application of GPU compression using cudaCompress, I present a particle tracing system designed for very large flow fields [TW13]. Since particles can move along arbitrary paths through large parts of the domain, particle integration requires access to the entire field in an unpredictable order. Thus, techniques for particle tracing in such fields require a careful design to reduce performance constraints caused by memory and communication bandwidth. One possibility to achieve this is data compression, but so far it has been considered rather hesitantly due to supposed accuracy issues. I investigate the use of data compression for turbulent vector fields, motivated by the observation that particle traces are always afflicted with inaccuracy. I therefore integrate a data compression layer based on cudaCompress into a block-based particle tracing approach. I quantitatively analyze the additional inaccuracies caused by lossy compression. Furthermore, I present a priority-based GPU caching scheme to reduce memory access operations. I confirm through experiments that the compression has only minor impact on the accuracy of the trajectories, and that on a desktop PC my application can achieve comparable performance to previous approaches on supercomputers.

6.1 Introduction

One of the most intriguing and yet to be fully understood aspects in turbulence research is the statistics of Lagrangian fluid particles transported by a fully developed

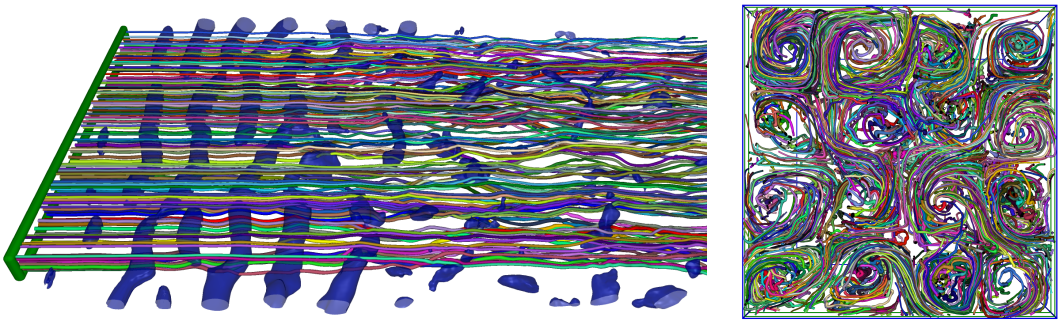

Figure 6.1: Stream lines in Mix ($3072 \times 600 \times 1024$; left; with iso-surfaces of vorticity magnitude) and MHD (1024^3; right), generated by my system in 2.0 and 4.4 seconds, respectively, including disk I/O.

turbulent flow. Here, a fluid particle is considered a point moving with the local velocity of the fluid continuum. The analysis of Lagrangian statistics is usually performed numerically by following the time trajectories of fluid particles in numerically simulated turbulent fields. Let $\mathbf{x}(\mathbf{y}, t)$ and $\mathbf{u}(\mathbf{y}, t)$ denote the position and velocity at time t of a fluid particle originating at position $\mathbf{y}$ at time $t = 0$. The equation of motion of the particle is

$$\frac{\partial \mathbf{x}(\mathbf{y}, t)}{\partial t} = \mathbf{u}(\mathbf{y}, t),$$

subject to the initial condition

$$\mathbf{x}(\mathbf{y}, 0) = \mathbf{y}.$$

The Lagrangian velocity $\mathbf{u}(\mathbf{y}, t)$ is related to the Eulerian velocity $\mathbf{u}^+(\mathbf{y}, t)$ via $\mathbf{u}(\mathbf{y}, t) = \mathbf{u}^+(\mathbf{x}(\mathbf{y}, t), t)$. By using a numerical integration scheme, the trajectory of a particle released into the flow can now be approximated.

Particle tracing in discrete velocity fields of a sufficient spatial and temporal resolution to resolve the higher wavenumber components in turbulent flows is nonetheless difficult. For reasonably-sized particle ensembles, the performance is strongly limited by the available memory bandwidth capacities due to the massive amount of data to be accessed during particle tracing. In particular the fact that particle trajectories often traverse large parts of the spatial domain and even turn back into regions already visited makes efficient data caching and parallelization strategies difficult to realize. As a consequence, up to now particle tracing in fully resolved turbulence

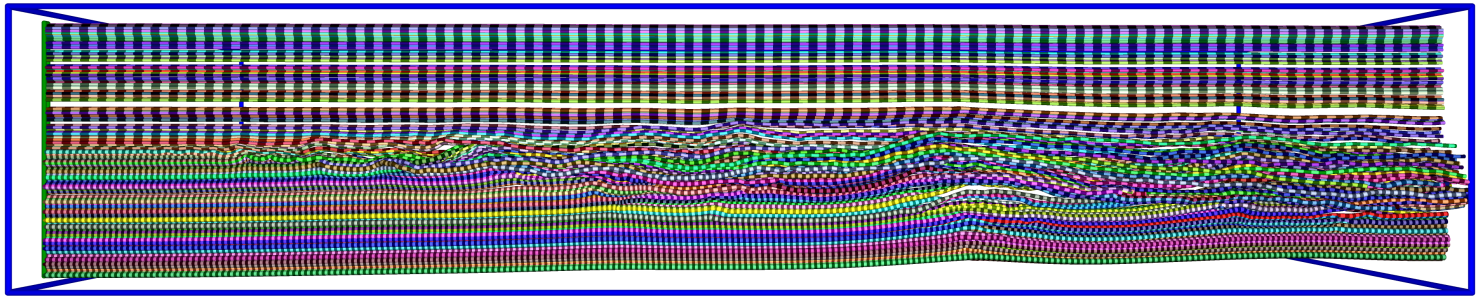

Figure 6.2: Stream lines in Mix, clearly showing the fast-moving fluid on the top and the slow-moving fluid on the bottom, as well as the turbulent mixing layer in between. The lines were generated in 6.4 seconds including disk I/O, and are rendered with a striped texture to indicate the local velocity. Each stripe corresponds to a time interval of equal size.

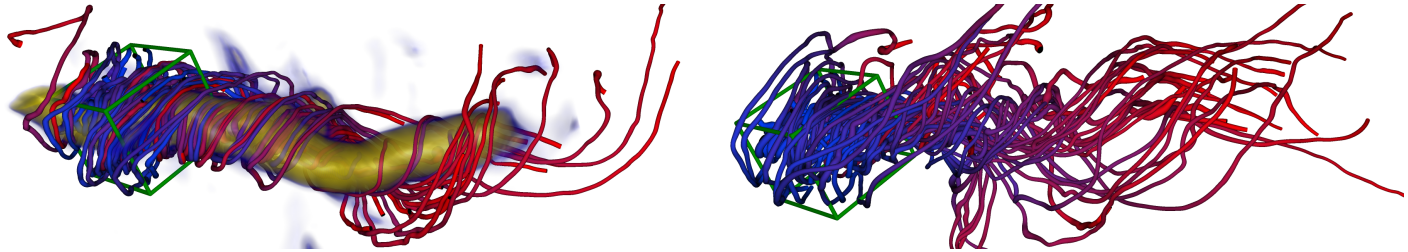

Figure 6.3: Stream lines (left) and path lines (right) in Iso ($1024^3 \times 1024$), created in 0.8 and 42 seconds including disk I/O.

fields has mainly been performed in a non-interactive way.

However, especially the possibility to interactively visualize the motion of turbulence features evolving with the flow can significantly improve our understanding of the properties of the turbulence small-scale structures [BTW*12]. To achieve similar insights related to Lagrangian statistics of turbulent flows, novel mechanisms and systems are required to enable interactivity.

In this chapter, I present a system for particle tracing in fully resolved turbulent flow fields which intertwines scalable data streaming and GPU particle integration to reduce bandwidth requirements and exploit parallelism among many particles. To handle the immense data volume, I employ the following strategies: The velocity field is subdivided in a brick-based partitioning scheme to create reasonably-sized units of data. The individual bricks are compressed using cudaCompress to alleviate bandwidth and memory constraints. Additionally, I introduce a novel caching strategy on the GPU to further reduce bandwidth limitations. This is necessary because in particle tracing the order in which the bricks are accessed depends on the selected trajectories.

I devote particular consideration to the quantization error that is introduced by the compression scheme. It is well known in turbulence research that in fully resolved turbulent flow fields, interpolation is the major source of errors in numerical particle tracing. This is due to the fact that turbulent velocity fields are highly nonlinear. The time-stepping error in numerical integration is generally much less significant, because the time-step is restricted to small values by enforcing the Courant number stability condition. The interpolation errors accumulate and are transmitted to the calculated trajectories. Thus, before embedding a lossy data compression scheme into particle tracing, one has to carefully analyze the additional inaccuracies that are introduced by that scheme.

I analyze the additional inaccuracies in particle trajectories which are caused by the compression of the turbulent vector field. As one major contribution, I show that these inaccuracies are in the same regions of variation as the inaccuracies due to interpolation.

I use four vector-valued data sets describing turbulent flow fields to validate my methods. Two terascale turbulence simulations originate from the JHU turbulence database cluster and are publicly available at `http://turbulence.pha.jhu.edu`.

Each is comprised of one thousand time steps of size 1024^3, making every time step as large as 12 GB at 3 floating-point values per velocity sample. The data sets contain direct numerical simulations of magneto-hydrodynamic (MHD) turbulence (see Fig. 6.1 (right) and Fig. 6.5) and forced isotropic turbulence (see Fig. 6.3), and are called "MHD" and "Iso" in the following. For a description of the simulation methods used to compute these data sets, refer to Li et al. [LPW*08]. The third data set, "Mix", involves a spatially developing mixing layer between two fluids entering the simulation volume at different velocities [AB12] (Fig. 6.1 (left) and Fig. 6.2). I use the final time step (at $t = 11650$) with a spatial resolution of $3072 \times 600 \times 1024$ (21.1 GB). The fourth data set, "Hom", contains homogeneous turbulence in flow around solid particles (see Fig. 6.4) and has a resolution of $2048 \times 2048 \times 4096$ (192 GB) [Uhl05, DU13].

The remainder of this chapter is organized as follows: Next, I review previous work on particle tracing. I then give an overview of the architecture of the proposed out-of-core particle tracing system in Section 6.3, including the underlying data structures and algorithms as well as the GPU caching scheme. Section 6.4 briefly describes the used compression scheme, and I discuss the error-sensitive adaptive compression. I continue in Section 6.5 with a detailed accuracy analysis of particle trajectories in the presence of lossy compression, and I relate the compression errors to those introduced in other parts of the computation. Next, a performance evaluation is given, including a comparison to previous approaches. I conclude the chapter in Section 6.6 with some remarks on future research.

6.2 Related Work

I do not attempt here to survey the vast body of literature related to flow visualization approaches based on stream and path line integration because they are standard in flow visualization. For a thorough overview, however, let me refer to the reports by Post et al. [PVH*03], Laramee et al. [LHD*04], and McLoughlin et al. [MLP*10].

Teitzel et al. [TGE97] put special emphasis on the investigation of the numerical integration error and the error introduced by interpolation. They conclude that an RK3(2) integration scheme provides sufficient accuracy compared to linear interpolation, but they do not consider higher-order interpolation methods. There is also

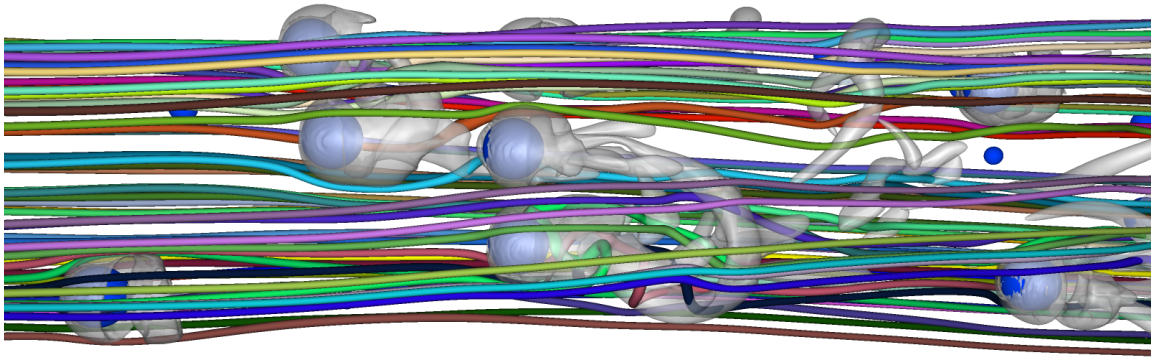

Figure 6.4: Stream lines in Hom ($2048 \times 2048 \times 4096$), generated in 1.8 seconds including disk I/O.

a number of works dealing especially with accuracy issues of particle tracing in turbulence fields [YP88, BM89, RHB94]. One of the conclusions was that Lagrange interpolation of order 4 to 6 provides sufficient accuracy, and it is therefore often used in practice (cf. Li et al. [LPW*08]).

The use of graphics hardware is popular for interactive particle tracing [SGvR*03, KKKW05, SBK06, BSK*07, Mur12]. The fundamental problem in GPU-based approaches is the limited memory available on such architectures, allowing only data sets of moderate size to be handled efficiently. To the best of my knowledge, no previous approach has addressed the problem of GPU particle tracing when even a single time step does not fit into GPU memory.

Precomputed particle traces: In a number of approaches it has been proposed to precompute and store particle trajectories for a number of prescribed seed points, and to restrict the visualization to subsets of these trajectories [Lan94, BKHJ01, EGM04]. In this way, all computation is shifted to the preprocessing stage, and storage as well as bandwidth limitations at runtime can be overcome.

Conceptually, the approach to restrict the flow field analysis to a set of precomputed trajectories can be seen as a kind of lossy data compression, where the seeding positions are quantized rather than the flow data itself. However, since even very small perturbations of the seeding positions can lead to vastly different trajectories, the resulting visualizations might not contain all relevant structures that are present in the data.

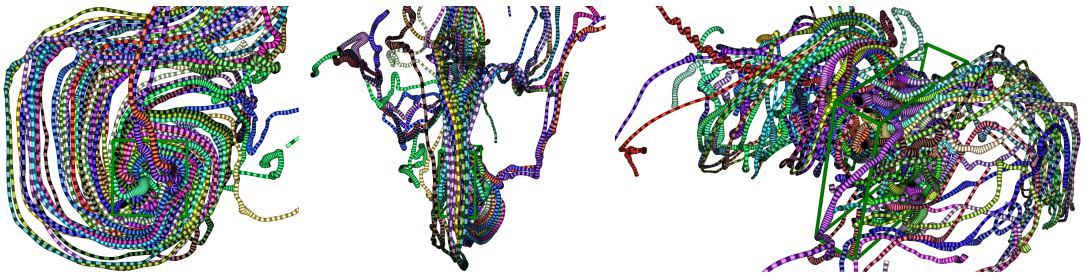

Figure 6.5: All eddies are not created equal. Left: Path lines in one eddy in the MHD data set. Middle: A side view reveals that most particles stay roughly in the plane of rotation. Right: In another eddy, there is much stronger movement in the normal direction.

Parallelization on compute clusters: Another possibility to address scalability issues in particle tracing is to employ parallel computing architectures such as tightly coupled CPU clusters or supercomputers. The larger memory capacities and I/O bandwidth on such systems make them attractive for handling large data sets. However, the highly data-dependent nature of particle tracing makes it difficult to effectively parallelize particle tracing on large distributed memory architectures.

There are two basic parallelization strategies, called *parallelize-over-seeds* (PoS) and *parallelize-over-blocks* (PoB) [PCG*09]. In both strategies, the data set is partitioned into blocks. In PoS, the seeding positions are distributed over the processors, and each processor dynamically loads those blocks required by its particles. This usually leads to fairly even load-balancing of computations, but it also results in the duplication of blocks in memory and increased I/O load since a block might be accessed by many processors. In PoB, the blocks are distributed across the processors, and each only handles particles within its assigned blocks. This avoids the duplication of blocks in memory, but causes severe load imbalance when many particles fall into the same processor's blocks while other processors remain idle. It also requires communication of particle positions between processors whenever a particle enters another processor's domain.

A number of approaches have been presented to mitigate the drawbacks of PoS or PoB. Pugmire et al. [PCG*09] have introduced a hybrid approach which seeks to combine the strengths of both. Camp et al. [CGC*11] have improved the efficiency of both PoB and PoS by making CPU cores in the same node share their memory. With PoS, this effectively increases the size of each CPU's block cache and thus low-

ers I/O load. With PoB, it increases the number of blocks available to each CPU and thus lowers the amount of communication between processors. Nouanesengsy et al. [NLS11] have attempted to improve load imbalances in PoB. They guide the assignment of blocks to processors by an estimated workload per block based on a precomputed flow graph which stores the probabilities of particles traveling from a given block to any of its neighbors. This typically improves the trajectory computation times significantly, but it comes at the cost of an expensive preprocess. In contrast to the aforementioned approaches, Peterka et al. [PRN*11] have also addressed large *unsteady* flow fields. They employ a PoB strategy and initially assign blocks to processors in a round robin manner. To improve load balancing, blocks are re-assigned in subsequent time steps based on the previous per-block load.

Yu et al. [YWM07] perform a hierarchical clustering of a time-dependent vector field based on the similarity between neighboring velocity vectors. Instead of fixed bricks, they use the resulting clusters for data partitioning among processing nodes. They avoid any inter-processor communication by tracing path lines only within each cluster. This results in very good performance and scaling. However, it artificially limits the length of the characteristic lines, and the seeding locations must be chosen at or close to the cluster centers.

As reported e.g. by Camp et al. [CGC*11], particle tracing approaches on clusters typically spend only a small fraction of the total time on the computation of particle traces. Most of the time is spent on either node-to-node communication, I/O, or waiting due to load imbalances. It can be concluded that despite its embarrassingly parallel nature, particle tracing is not very well suited for computation on distributed memory clusters. The main benefit of such systems appears to be the large amount of aggregated memory, which can often prevent expensive trips to external memory such as hard disks.

6.3 Out-of-Core Particle Tracing

The proposed system for out-of-core particle tracing takes as input a sequence of 3D velocity fields given on a Cartesian grid. Each field represents the state of a flow field at a different time step. In a preprocess, each grid is partitioned into a set of equally-sized bricks. A halo region is added around each brick to allow proper interpolation

at brick boundaries. The bricks are compressed before being stored sequentially on disk. An index structure is stored along with the brick data to enable fast access to individual bricks at runtime. For one time step consisting of 1024^3 velocity values, this process takes about 5 minutes.

At runtime, the computation of particle trajectories is performed on the GPU. For that, bricks which are required to perform the numerical integration are requested from the CPU. The CPU decides based on a particular strategy (see Section 6.3.3) which bricks to upload to the GPU from main memory or disk. Compressed brick data is cached in CPU memory. The compression reduces disk bandwidth requirements and allows caching a large number of bricks. For use on the GPU, the compressed brick data is uploaded into GPU memory and immediately decompressed. The decompressed brick is stored in a large 3D texture map, the so-called *brick atlas*. In this way, all GPU memory stores ready-to-use flow data, apart from a small temporary buffer for the upload of compressed data. In the current implementation I use bricks of size 128^3 each, including a halo region of size 4. I have found that this size provides the best trade-off between locality of access and storage overhead for the halo voxels. The size of the *brick atlas* is chosen based on the amount of available GPU memory.

6.3.1 Particle Tracing in Rounds

Fluid particles are advected in parallel on the GPU to exploit memory bandwidth and computational capacities. I use the CUDA programming API and issue one thread per particle, grouped into thread blocks of size 128. Each thread advects the position of its particle while the required flow data is available in the *brick atlas*. Since the set of bricks which are required to perform the computation of all trajectories does not fit into GPU memory in general, only a subset can be made available at a time. An *index* buffer stores the mapping from spatial brick index to position in the brick atlas. It is indexed by the 3D brick index $\mathbf{b}$, which can be computed from a particle's position $\mathbf{p}$ and the brick size $\mathbf{s}$ as $\mathbf{b} = \lfloor \mathbf{p}/\mathbf{s} \rfloor$. If a brick is not currently available in the *brick atlas*, the corresponding *index* entry contains the value -1. Fig. 6.6 illustrates the employed data structures and the CPU-GPU interaction using a 2D example.

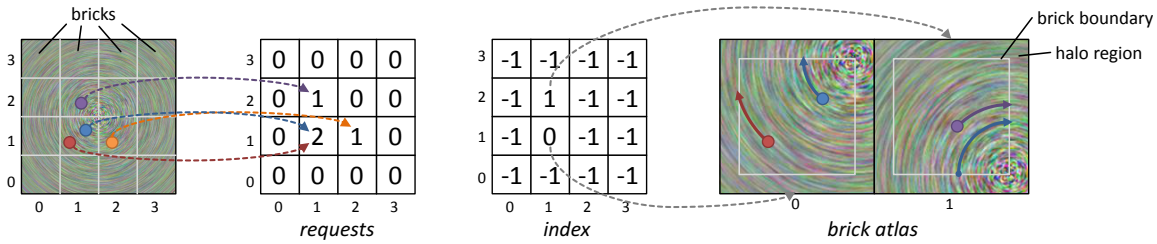

Figure 6.6: Out-of-core particle tracing (2D example): (1) Particle positions are uploaded to the GPU where trajectories are computed in parallel. Requests of required bricks are issued (corresponding entries in the *requests* buffer are incremented). (2) The CPU downloads the *requests* buffer, uploads some requested bricks into the GPU *brick atlas*, and clears the *requests* buffer. The *index* buffer stores each brick's index in the atlas and is also uploaded to the GPU. (3) The GPU advects each particle until it requires a brick not yet resident in GPU memory. The *requests* buffer is incremented again based on the new particle positions, and the process continues at step (2).

To start the computation, the user specifies the number of fluid particles to trace and the seed region in which they are spawned. Random positions inside the seed region are stored in the *particle* buffer on the GPU (not shown in the figure). Particle tracing then proceeds in a ping-pong fashion between the GPU and the CPU: The GPU advances all particles in parallel. Whenever a particle enters a brick which is not stored in the *brick atlas* (indicated by a -1 entry in the *index*), the GPU requests this brick for the next round of tracing. This is realized by atomically incrementing the corresponding entry in a *requests* buffer. The particle's last position and any additional information, such as the current step size for adaptive integrators, are stored in the *particle* buffer. The GPU stops when all particles a) must stop because they are waiting for a brick to be uploaded to the GPU, b) have reached their maximum age, or c) have been advanced by a fixed maximum number of steps (64 in the current implementation). The CPU then downloads the *requests* buffer and determines the bricks to be uploaded next into the atlas. With these bricks being available on the GPU, particle tracing is restarted. The process is finished once all particles have either reached their maximum age or left the domain.

6.3.2 Tracing Across Brick Boundaries

Special care has to be taken whenever a particle moves close to a brick boundary. In this case it has to be ensured that all velocity values required in the integration step are available in the current brick. The number of required values depends on the support of the interpolation kernel. Fig. 6.7 depicts the admissible locations for velocity interpolation near a brick boundary for several interpolation schemes. For a multi-stage integration method, not only the initial particle location but also all intermediate stages of the integrator must lie within the admissible area. This can be guaranteed by limiting the maximum integration step size Δt appropriately: When the distance of the particle to the boundary of the admissible region in dimension i is b_i, then Δt must be limited to $\min_i(b_i/v_{i,\max})$, where $v_{i,\max}$ is the maximum absolute value of the i'th velocity component in the region. This value is computed in the preprocess and stored along with the brick data.

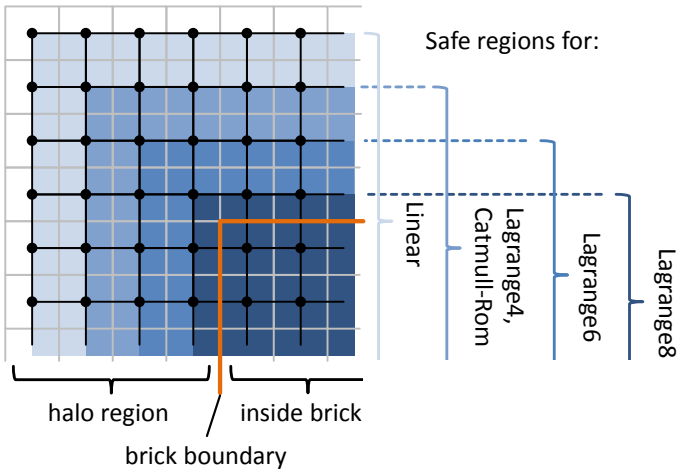

Figure 6.7: Velocity sampling near a brick boundary. Texel boundaries are shown in gray; grid points and cell boundaries in black. A halo region is stored to allow sampling of velocity values in a small region outside of the brick. The size of the required halo region depends on the interpolation scheme.

6.3.3 Heuristic Brick Selection and Paging

Since the full set of bricks required to trace a given set of particles cannot be stored in GPU memory, subsets of these bricks have to be paged in and out, and processed sequentially in a number of rounds. However, it can always happen that a brick which has been paged out is later visited by some particle and has to be paged in again. Thus, an appropriate paging strategy is required to reduce the number of bricks which are uploaded multiple times.

Besides multiple uploads of the same brick, the paging strategy also has to take into account the number of particles which can be advected using the currently available bricks. The massively parallel nature of GPUs can only be exploited to its full potential when many particles can be processed in parallel.

These two requirements, however, contradict each other: According to the first requirement, a brick should be kept in GPU memory as long as possible to avoid multiple uploads. Conversely, the second requirement demands that a brick through which few or no particles are moving should be paged out immediately, so that bricks required by a large number of particles can be paged in.

Since it is not known in advance which bricks are to be visited at which times, it is not possible to devise an optimal paging strategy. Instead, I have devised a simple yet very effective heuristic paging strategy which attempts to balance the two goals. It is based on the following observations:

- Paging out a brick which is currently required by some particles will always result in a repeated upload of this brick later on.
- A brick which is not required by any particle might be visited again later on, so it might still be beneficial to keep it on the GPU. However, since keeping such bricks at least currently wastes GPU memory, a balance must be found between minimizing premature swap-outs and maximizing GPU occupancy.
- Processing spatially close bricks at the same time tends to improve brick reuse and thus helps to avoid repeated uploads. It also increases the chances of particles moving between available bricks within one round, which increases the average number of active particles at any time and thus improves GPU utilization.

Based on these observations, the paging strategy operates as follows. Note that paging out a brick does not entail any data transfer, but simply means clearing the corresponding entry in the *index* buffer.

- Only bricks which are not required by any particle are ever paged out.
- A brick which is not required by any particle is kept on the GPU for a fixed number of rounds, r, before it is paged out. I have found that paging out such bricks after $r = 4$ rounds works well, with each round limited to a maximum of 64 integration steps per particle. In my experiment, increasing the value of r could only reduce the number of brick uploads by up to 10%, while increasing the particle advection time by an order of magnitude due to the less efficient GPU usage. Smaller values, on the other hand, significantly increased the number of brick uploads.

Whenever a slot in the brick atlas is available, the next brick to upload is selected according to the following priorities:

- Only bricks required by at least one particle are considered in order to avoid spurious uploads.
- Bricks which are required by a large number of particles are preferred.
- Bricks are favored or disfavored based on the availability of their neighbors in GPU memory, as well as the numbers of particles their neighbors contain. Particles expected to travel between the brick under consideration and an available neighbor result in a priority bonus in order to enhance locality and data re-use. Particles traveling in from an unavailable neighbor result in a penalty—in this case, it would be better to process the neighbor first, so these particles can coalesce with those in the current brick. Finally, particles traveling out into an unavailable neighbor carry neither bonus nor penalty.

Taking these rules into account, a heuristic load parameter, l, is computed for each brick b as follows:

$$l(b) = \sqrt{c_b} \; + \; h \cdot \sum_{n \in N(b)} \begin{cases} \sqrt{c_b \cdot p_{b,n}} + \sqrt{c_n \cdot p_{n,b}} \; , & n \text{ available in GPU memory} \\ -\sqrt{c_n \cdot p_{n,b}}, & \text{else} \end{cases}$$

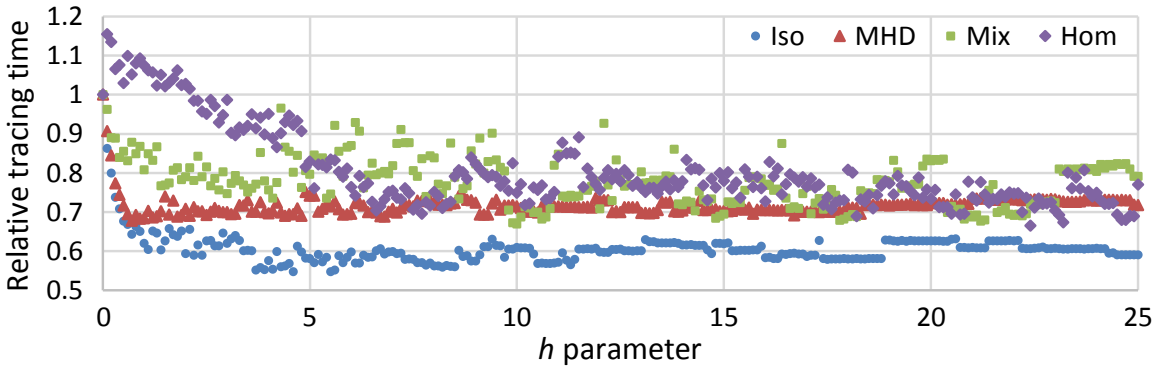

Figure 6.8: Relative times for tracing 4096 stream lines in four data sets (including CPU-GPU data transfer and GPU advection), depending on the heuristic parameter h. The absolute times at $h = 0$ are 27.2, 23.3, 29.0, and 99.8 seconds for Iso, MHD, Mix, and Hom, respectively.

This parameter is then used to assign priorities to the requested bricks. Here, c_b is the number of particles in brick b, and $N(b)$ is the set of neighbors of b. $p_{a,b}$ is the probability of a particle from brick a traveling into brick b. Note that in general $p_{a,b} \neq p_{b,a}$. These probabilities can be precomputed by tracing a number of particles within each brick and storing in which direction they leave the brick, i.e. a flow graph (cf. [NLS11]). Computing such a flow graph for a 1024^3 flow field takes around 1 minute in my system excluding disk I/O. However, I have found that when a flow graph is not available, simply substituting a constant value for the probabilities works surprisingly well, increasing the total trajectory computation time by less than 10% is most cases.
The user-defined parameter $h \geq 0$ is used to weight the neighborhood-based bonus and penalty terms. Larger values correspond to a larger preference for locality. Fig. 6.8 shows the time required to trace 4096 stream lines in four data sets for different values of h. Choosing an appropriate value for h reduces the total tracing time by about 30-40% in all data sets. In the following experiments, I always use a value of $h = 8$.

Bricks are loaded from disk into CPU memory using the same priorities. However, bricks which currently do not contain any active particles are also prefetched into the cache if the disk would otherwise be idle.

6.3.4 Unsteady Flow

So far, I have addressed only the case of steady flow, i.e. stream line computation. Path lines are computed in much the same way, with some straightforward extensions to account for the time-dependent nature of the flow. Each slot in the brick atlas now contains multiple time steps of the same spatial brick; the number of slots is reduced accordingly. The *index* records which time steps are currently available. Similarly, the *requests* buffer holds not only the number of requests to a spatial brick but also the earliest time step that was requested. The CPU also tracks the earliest time step which was requested globally and pages out all brick time steps older than that, since they will not be visited again by the current particles. Finally, during the selection of bricks to upload, priority is given to older time steps, so that all path lines advance at roughly the same speed and multiple uploads of the same data are avoided.

6.3.5 Interpolation Schemes

My system supports a number of different interpolation schemes which are used in numerical particle integration. The simplest one is linear interpolation, which comes "for free" on the GPU. Lagrange interpolation of order n fits a polynomial of degree $n-1$ through the n grid points centered around the interpolation point. Second-order Lagrange interpolation is thus equivalent to linear interpolation. I have implemented 4th, 6th, and 8th order Lagrange interpolation (called Lagrange4/6/8), corresponding to cubic, quintic, and septic polynomials, respectively. It is worth noting here that Lagrange6 can be considered one of the standard methods in turbulence research. The application of higher-order schemes can hardly be found in the literature. Additionally, I have implemented an interpolation scheme based on Catmull-Rom splines. This approach fits a cubic polynomial to the values and first derivatives, estimated via central differences, at two grid points. Compared to 4th order Lagrange interpolation, this has the advantage of creating a globally C1-continuous interpolant, while Lagrange interpolation is only C0. All described interpolation schemes are extended to multiple dimensions by a tensor product approach.

The minimum size required by the halo around each brick depends on the size of the chosen interpolation kernel, e.g. 2 voxels for Lagrange4 and 4 voxels for Lagrange8 (cf. Fig. 6.7). Thus, a selected halo size of 4 allows for any interpolation scheme with

a support of up to 8^3 voxels. If only lower-order interpolation is required, a smaller halo size can be used which slightly reduces memory and bandwidth requirements.

6.4 Turbulent Vector Field Compression

In the absence of data compression, the performance of the proposed system for particle tracing in large turbulence fields is vastly restricted by bandwidth limitations when reading the data from disk. For instance, the computation of stream lines as shown in Fig. 6.1 (right) in one single *uncompressed* time step involves a working set of almost 5 GB. The visualization takes roughly 45 seconds, of which over 98% are spent waiting for data from disk. Thus, there is a dire need for compression in order to reduce the amount of data to be streamed.

When using data compression, the introduced compression error has to be carefully examined. Since no error is introduced by lossless compression schemes, they might be an attractive choice in particle tracing applications. However, for floating-point data the achieved compression ratio is usually quite modest. For instance, the lossless schemes proposed by Isenburg and Lindstrom [ILS05, LI06] can only compress the turbulence data to roughly $\frac{2}{3}$ of its original size. They achieve a decoding throughput of about 10 million floating point values per second, corresponding to over 600 ms for the decompression of a single 128^3 grid of 3D velocities. More sophisticated prediction schemes can slightly improve the compression ratio [FM12], but they come at the expense of lower throughput.

In comparison, lossy GPU compression based on cudaCompress provides high compression ratio and decompression throughput. For the present flow data, it achieves a decoding throughput of over 650 million floating-point values per second.

6.4.1 Interpolation Error Estimate

When a lossy scheme for vector field compression is used, it is clear that the reconstructed field is afflicted with some error compared to the initial field. At first, this seems to preclude lossy compression schemes in particle tracing, because the local reconstruction errors accumulate along the particle trajectories. On the other hand, this error has to be seen in relation to the error that is inherent to particle

trajectories even when computed in the original data. Even without compression the reconstructed samples are not exact in general, due to the interpolation which is used to reconstruct the data values from the initially given discrete set of samples. This interpolation makes assumptions on the continuous field which, in general, do not hold. As a consequence, it has to be accepted that the trajectories computed numerically using interpolation diverge from those in reality, even without compression.

It therefore makes sense to choose the compression quality so that the additional error introduced by the compression scheme is in the order of the error introduced by interpolation. It is worth noting, however, that without additional information about a data set it is impossible to accurately compute or even estimate the interpolation error. In some cases, theoretical error bounds depending on higher-order derivatives of the continuous function can be given; see, for instance, Fout and Ma [FM13] for such a bound when linear interpolation is used. On the other hand, the derivatives of the continuous function are typically not known exactly. In that case, such bounds themselves come with some uncertainty. In addition, even with exact knowledge of the derivatives, they often overestimate the actual error significantly [ZXM10].

Therefore, I have adopted a different approach to estimate the interpolation error: I take the difference between interpolation results from a reference high-order interpolator and a lower-order interpolator as an estimate for the error in the low-order interpolator. For two of the discrete turbulence data sets analyzed in this work, Iso and MHD, an exact interpolator is known. Due to the pseudo-spectral method that was used to simulate the turbulent motion [LPW*08], the velocity field is guaranteed to be band-limited in the Fourier sense. As a consequence, Fourier or trigonometric interpolation using trigonometric polynomials of infinite support gives exact velocity values between grid points [RHB94]. Due to efficiency reasons, however, what is used in practice for particle tracing is some interpolation scheme of "sufficiently high order" which resembles Fourier interpolation, e.g. Lagrange6. For instance, in Fig. 6.9 (left) the trajectories using trigonometric and Lagrange6 interpolation are compared. It is worth noting that even though in the turbulence community it is usually agreed that Lagrange6 is of sufficient accuracy for particle tracing, significant deviations from the ground truth can be observed.
In cases where a "good" interpolator for a given data set is not known, i.e. for Mix and Hom, I use Lagrange16 interpolation as the reference. It is clear that without

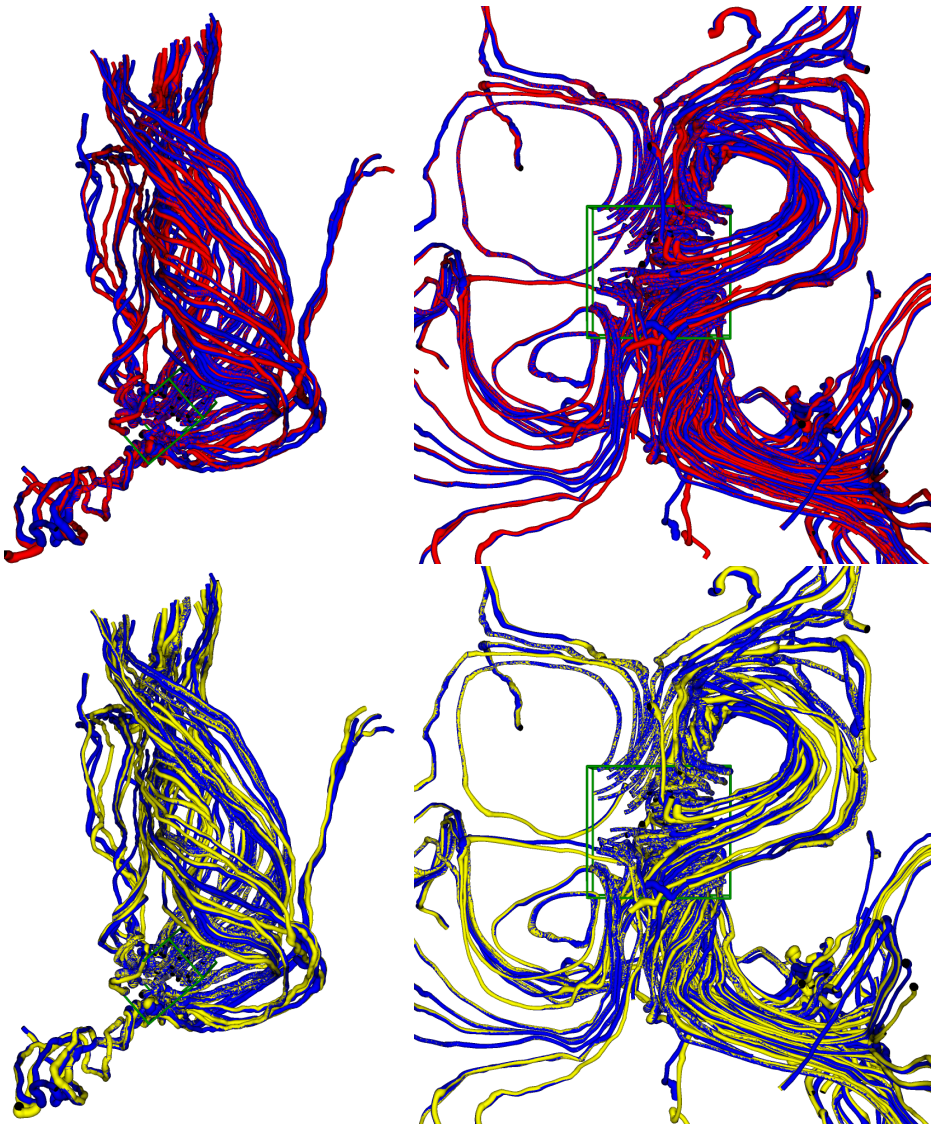

Figure 6.9: Top: Particle trajectories using trigonometric (blue) and Lagrange6 (red) interpolation for velocity sampling. Bottom: Particle trajectories using trigonometric interpolation in the original data (blue) and Lagrange6 interpolation in compressed data (yellow) for velocity sampling. It is worth noting that the yellow lines appear to be of similar accuracy as the red lines.

knowledge of the exact continuous velocity field a reliable statement about the interpolation error is impossible. However, in the absence of any further information, the

implied assumption of "sufficient smoothness" seems very reasonable. Otherwise, the given discrete set of velocity samples have to be deemed insufficient for accurately representing the continuous field.

The interpolation error over the whole volume for a given interpolator can now be computed. I upsample the volume to four times the original resolution using the interpolator under consideration as well as the reference interpolator. The root-mean-square (RMS) of the difference between the upsampled volumes then is a good approximation of the average error introduced by the interpolation. With local interpolators like Lagrange, this process can be done block-by-block in a straightforward manner. For a 1024^3 velocity field, it takes about 2 hours to evaluate the errors in linear and Lagrange4/6/8 interpolation vs. Lagrange16. Trigonometric interpolation, on the other hand, has to be evaluated globally. To generate the trigonometric interpolant in a computationally efficient way, I use the following approach: First, I perform a fast Fourier transform (FFT) on the flow field using the FFTW library [FJ05]. In the frequency domain, I then quadruple the data resolution in each dimension by zero padding. Finally, I perform an inverse FFT to generate a flow field of four times the original resolution. This field agrees with the original field at every fourth vertex, and the other vertices lie on the trigonometric interpolant between the original data samples. Fig. 6.10 illustrates FFT-based upsampling in 1D, but only doubling the data resolution. Generating the 4096^3 trigonometric interpolant from a 1024^3 velocity field in this way takes about 1.5 hours including disk I/O. Given the 4096^3 trigonometric interpolant, evaluating the interpolation errors in a 1024^3 velocity field for all four listed interpolation schemes takes another 2 hours including disk I/O.

6.4.2 Error-Guided Data Compression

Equipped with the average interpolation error for any chosen interpolation scheme, a quantization step can be chosen so that the compression error is equal to or falls below the interpolation error. In the wavelet-based scheme provided by cudaCompress, the average error is roughly equal in magnitude to the quantization step and, thus, the acceptable error is a reasonable choice for the quantization step. Table 6.1 lists the RMS interpolation errors in the four data sets for a number of different interpolation

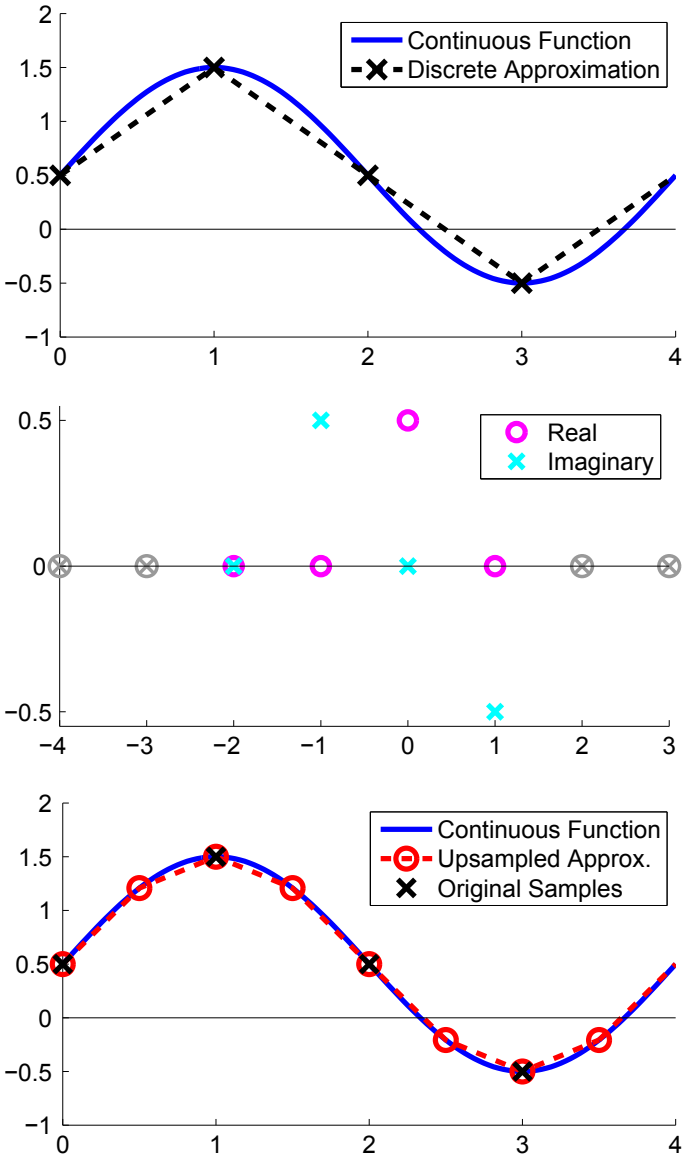

Figure 6.10: FFT-based upsampling process. Left: A periodic band-limited function with a period of 4, and its discrete approximation sampled at a frequency of 1. Middle: FFT coefficients of the function in magenta and cyan. Because the input function was real, the coefficients have a Hermitian symmetry. The coefficients are padded with zeros to the left and right, corresponding to higher frequencies with an amplitude of zero. Right: The inverse FFT of the padded coefficients results in a higher-resolution approximation to the continuous function. Note that the even grid points of the upsampled approximation agree with the original grid points.

schemes. To verify that the lossy compression does not unduly affect the interpolant, I have computed the interpolation errors a second time after compression, comparing the reconstructed volumes to the original reference solution. It can be seen that by setting the quantization step equal to the RMS interpolation error, the error is increased by less than 50% in all cases. It is worth mentioning that performing the same test with an upsampling factor of only 2 instead of 4 yields results within 5% of the listed numbers. This indicates that the discrete computation approximates the actual interpolation error very closely. This is expected, as the reference interpolant is by definition band-limited with respect to the original resolution, so no high-frequency deflections can occur between the original grid points.

It remains to show that the accumulation of the additional quantization errors does not introduce significantly larger regions of variation in the trajectories. A first experiment can be seen in Fig. 6.9 (right), where the trajectories computed on the compressed field using Lagrange6 interpolation are compared to the ground truth trajectories. Compared to Fig. 6.9 (left), the deviations seem to be in the same order of variation. A detailed quantitative accuracy analysis is given in the following section.

6.5 Evaluation

To evaluate the performance of the system as well as the accuracy of the resulting trajectories, I have conducted a number of experiments. In the first set of experiments I analyze the accuracy of trajectories in the presence of lossy compression. In the scope of a second set of experiments I evaluate the performance of the system. In the following, I first introduce the error metrics used to analyze the accuracy of the computed trajectories.

6.5.1 Error Metrics

Due to errors induced by the employed interpolation scheme and by lossy compression, a trajectory may gradually diverge from the ground truth over time. To evaluate the accuracy of computed trajectories, an error metric is required to quantitatively measure the difference between two trajectories starting at the same seed point.

One obvious metric is the maximum or average distance between trajectories $s_0(u)$, $s_1(u)$ along their parameter u. In addition, several metrics exist which measure different kinds of distance between two curves, such as the (discrete) Fréchet distance [EM94] and the distance under dynamic time warping (DTW). While the Fréchet distance corresponds to a type of maximum distance, the DTW distance is akin to an average distance. Both disregard the u parametrization and instead are concerned only with the shape of the curves. All these metrics measure the distance along the complete trajectories. However, once two particles have diverged by some critical distance, their further behavior depends only on the characteristics of the flow field: They might diverge further or even converge again, but this provides no insight into the accuracy of the trajectory computation. Therefore, I introduce a new metric taking this into account, called the (clamped) divergence rate. Instead of measuring a distance between trajectories, it computes the rate at which they diverge. Given two trajectories $s_0(u), s_1(u)$ over a parameter interval $[u_0, u_{\max}]$, I define their divergence rate as

$$d_{s_0,s_1} := \frac{\mathrm{dist}_{s_0,s_1}(u_{\mathrm{div}})}{u_{\mathrm{div}} - u_0}, \text{ where}$$

$$\mathrm{dist}_{s_0,s_1}(u) := \|s_0(u) - s_1(u)\| \text{ and}$$

$$u_{\mathrm{div}} := \max\left\{u \in [u_0, u_{\max}] \;\middle|\; \forall \tilde{u} \in [u_0, u] : \mathrm{dist}_{s_0,s_1}(\tilde{u}) \leq \Delta s\right\}.$$

u_{div} is the last point along the trajectories at which they have not yet diverged by more than Δs. In the following experiments, I have set the critical distance Δs equal to the grid spacing.

My definition of the trajectory divergence rate is similar in spirit to the idea of the finite-size Lyapunov exponent (FSLE) [ABC*97]. The FSLE measures the time it takes for two particles, initially separated only by an infinitesimal ϵ, to diverge by some given distance, usually specified as a multiple of ϵ. A fundamental difference is that here both trajectories start at exactly the same position, and I measure their divergence as an absolute distance rather than relative to their initial separation.

Table 6.1: Root-mean-square interpolation errors for different interpolation schemes in four turbulent flow fields before (*orig*) and after compression (*comp*), compared to the reference interpolant (trigonometric for Iso and MHD, Lagrange16 for Mix and Hom). The interpolation error has been evaluated in a grid of four times the original resolution.

	Iso (range: 6.67)		MHD (range: 2.77)		Mix (range: 3.21)		Hom (range: 2.37)	
Interpolation	orig	comp	orig	comp	orig	comp	orig	comp
Lagrange8	0.86E-3	1.26E-3	3.48E-4	4.97E-4	1.31E-4	1.47E-4	1.92E-5	2.11E-5
Lagrange6	1.10E-3	1.60E-3	4.52E-4	6.32E-4	2.16E-4	2.41E-4	3.26E-5	3.48E-5
Lagrange4	1.71E-3	2.41E-3	7.20E-4	9.63E-4	4.01E-4	4.45E-4	6.65E-5	6.84E-5
Linear	5.15E-3	6.65E-3	2.29E-3	2.81E-3	1.25E-3	1.37E-3	2.46E-4	2.57E-4

Table 6.2: File sizes and compression factors. For *Very Low*, *Low*, *Medium*, and *High*, the quantization step was chosen equal to the error in linear and Lagrange4/6/8 interpolation, respectively (cf. Table 6.1); for *Very High*, to half the error in Lagrange8.

	Iso		MHD		Mix		Hom	
Quality	size	factor	size	factor	size	factor	size	factor
Uncompressed	14.7 GB	–	14.7 GB	–	25.7 GB	–	235.0 GB	–
Very High	1.79 GB	8.21	1.55 GB	9.48	1.11 GB	23.1	5.57 GB	42.2
High	1.25 GB	11.8	1.06 GB	13.9	883 MB	29.8	4.29 GB	54.8
Medium	1.08 GB	13.6	942 MB	16.0	720 MB	36.6	3.51 GB	67.0
Low	843 MB	17.9	712 MB	21.1	544 MB	48.4	2.71 GB	86.7
Very Low	387 MB	38.9	331 MB	45.5	296 MB	88.9	1.69 GB	139.1

6.5.2 Accuracy Analysis

To compare the accuracy of particle trajectories computed in the original and compressed data sets, and via different interpolation schemes, a reference solution is required to which the trajectories can be compared. For two of the test data sets, Iso and MHD, trigonometric interpolation is known to be exact. Since evaluating the trigonometric interpolant during particle tracing is impracticable, I have upsampled the data sets to four times the original resolution (see Section 6.4.1) as the ground truth. Particle trajectories traced in the upsampled versions using 16th order Lagrange interpolation then act as the reference solution. While this is not equivalent to true trigonometric interpolation in the original data, the remaining error is expected to be negligible since the difference between the two times and four times upsampled versions is already very small (cf. Section 6.4.1). For the other two data sets, Mix and Hom, I assume Lagrange16 interpolation as the reference solution, in line with the analysis of the interpolation error in Section 6.4.1.

For analyzing the accuracy of computed trajectories, I have generated a set of 4096 seed points in each data set. In Iso and MHD, the seeds are distributed randomly over the entire domain. In Mix and Hom, they are on a plane near the inflow. Particles were traced from the seed points through different versions of the data sets: The upsampled reference version (for Iso and MHD), the original uncompressed data, and compressed versions at different compression ratios. The quantization steps for the compressed versions were chosen equal to the errors in linear and Lagrange4/6/8 interpolation as listed in Table 6.1. In addition, in one high-quality compressed version of each data set the quantization step was set to half the Lagrange8 interpolation error. The compressed file sizes and compression ratios are listed in Table 6.2.

To minimize the impact of inaccuracies due to numerical integration errors, I used the Runge-Kutta method by Dormand and Prince [DP80] in all experiments. The method provides a 5th order solution and a 4th order error estimate which is used for adaptive step size control. The error tolerance for step size control was reduced until the accuracy of the results did not improve any further.

Figs. 6.11 and 6.12 provide the main results of the accuracy analysis. The graphs show the RMS of the average, maximum, Fréchet, and DTW distance, as well as the divergence rate, over all trajectories for different compression ratios and interpolation

schemes. For reference, the grid spacing is approximately 0.00614 in Iso and MHD, 0.154 in Mix, and 0.00192 in Hom. The most prominent finding is that linear interpolation performs very poorly and eclipses the errors introduced at even the highest compression ratios. The differences between the other interpolation schemes are comparatively small; as expected, with some advantage of the higher-order schemes. All distance metrics give qualitatively similar results. However, all metrics except for my novel divergence rate display a significant amount of noise in the results, especially in Hom. This is caused by a few individual trajectories with a very large distance to their reference. These trajectories have a very large impact on the RMS distance, but actually carry little information on the accuracy of the results, as explained in Section 6.5.1. The divergence rate, on the other hand, handles such trajectories well.

The most important observation with regard to the lossy compression is that when the quantization step is chosen smaller than the interpolation error (e.g. Lagrange4 interpolation and a compression quality of "Medium" or higher), the additional error introduced by the compression is extremely small. For example, switching from Lagrange6 to Lagrange4 interpolation has a larger impact on the accuracy than switching from uncompressed data to the "High" compression quality in all four data sets.

6.5.3 Performance Analysis

The performance of any particle tracing system depends on a multitude of factors, such as the characteristics of the data set, the number and placement of seeding locations, and the total integration time. This makes an exhaustive performance evaluation and comparison to other approaches fairly difficult. Instead, I tried to capture the typical performance characteristics of my system. For Iso and MHD, I investigated the following two scenarios:

1. **Sparse:** This is the same scenario that was used for pursuing the accuracy analysis. 4096 seeding locations are distributed uniformly in the domain, and particles are traced for 2.5 and 5 time units in Iso and MHD, respectively.
2. **Dense:** This scenario models an interactive exploration. 1024 seed points are placed within a small box with an edge length of 10% of the domain size. The particles are traced over 5 and 10 time units in Iso and MHD, respectively.

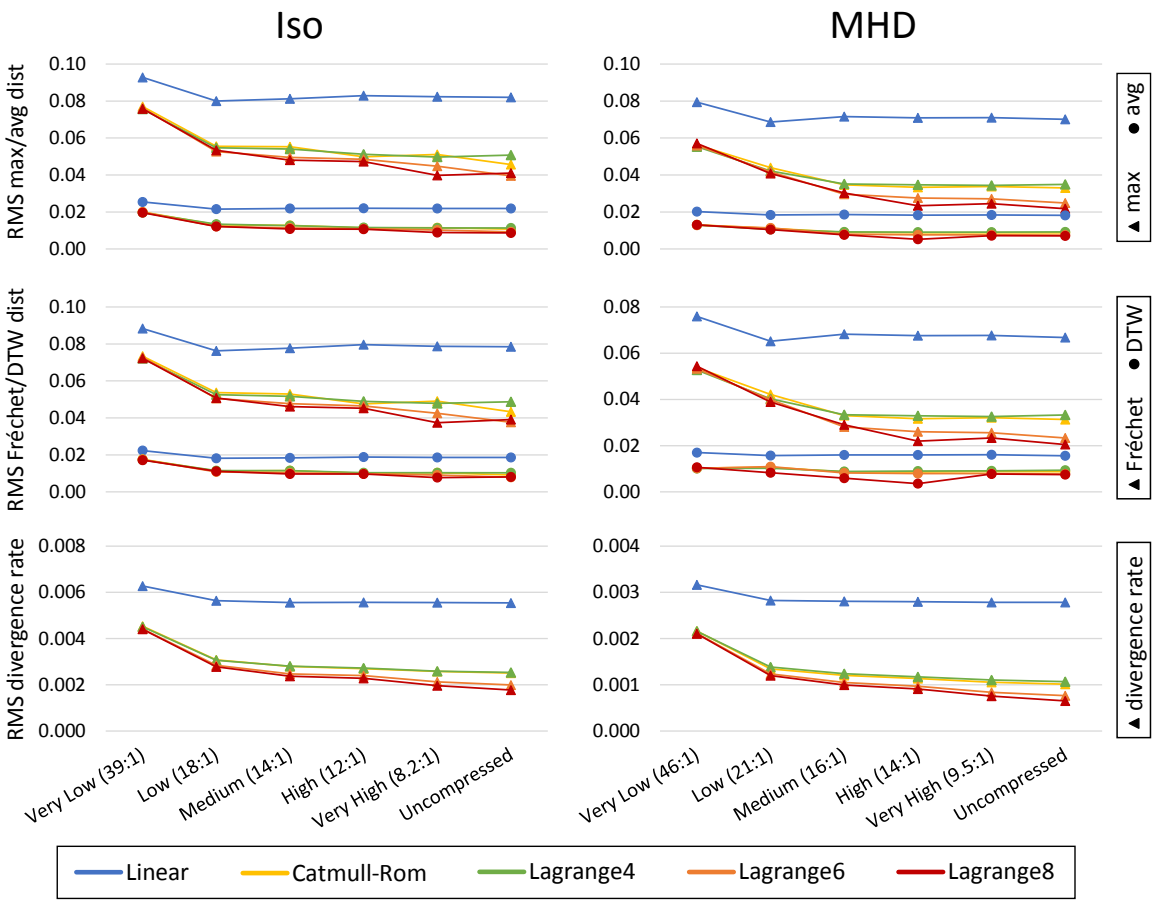

Figure 6.11: Accuracy of stream lines vs. compression quality in the Iso and MHD data sets using different interpolation schemes. Accuracy is reported as the root mean square (RMS) of the individual trajectory distances (see Section 6.5.1), computed against trajectories traced using the reference trigonometric interpolation. For comparison, the grid spacing in both data sets is $2\pi/1024 \approx 0.0061$.

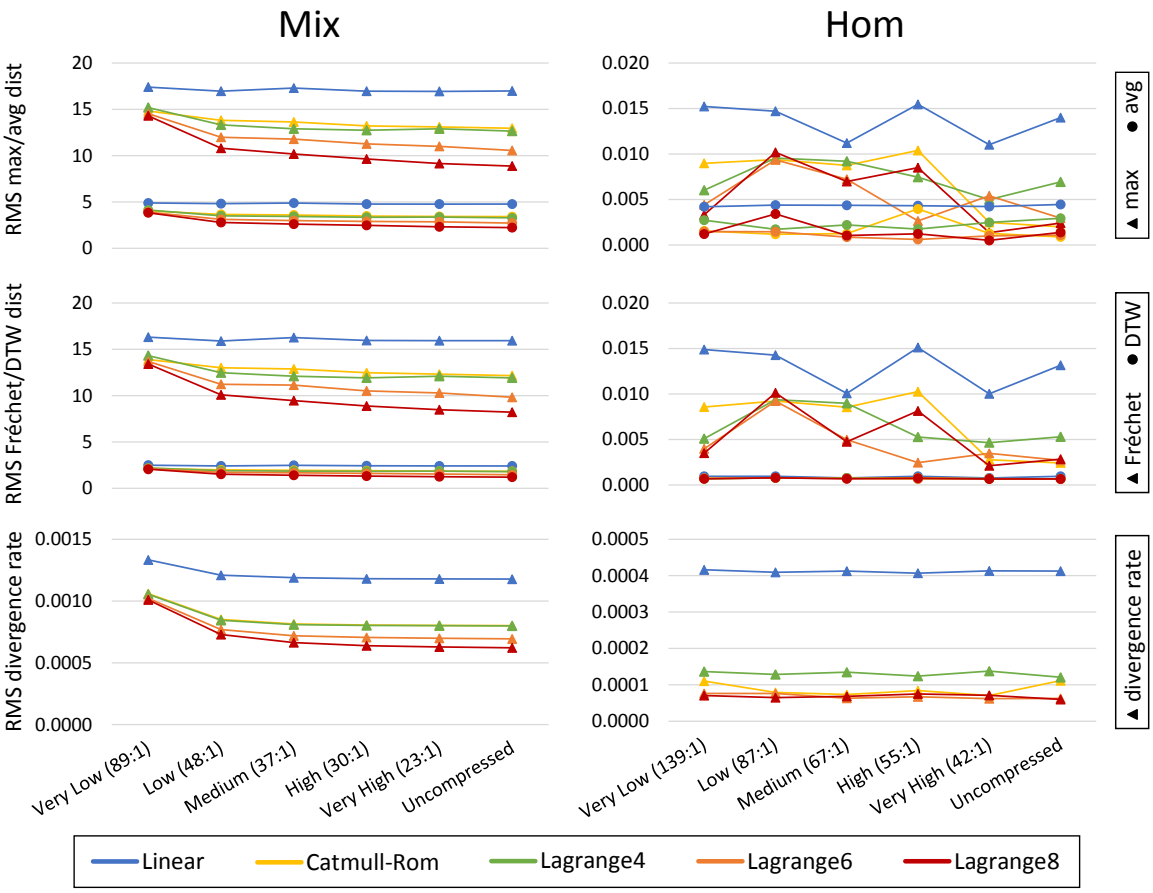

Figure 6.12: Accuracy of stream lines vs. compression quality in the Mix and Hom data sets using different interpolation schemes. Accuracy is reported as the root mean square (RMS) of the individual trajectory distances (see Section 6.5.1), computed against trajectories traced using the reference Lagrange16 interpolation. For comparison, the grid spacing is 0.154 in Mix and 0.00192 in Hom.

For Mix and Hom, where there is a primary flow direction, 4096 seeding locations are placed on a plane near the inflow. In Mix, the seeding plane spans 80% of the domain in the spanwise and 10% in the crosswise direction, so that most stream lines travel through the region where the two fluids mix. In Hom, the size of the seeding plane is half of the domain size in the x and y dimension.

All timings were measured on a PC with an Intel Core i5-3570 CPU (quad-core, 3.4 GHz) with 8 GB of DDR3-1600 RAM, equipped with an NVIDIA GeForce GTX 680 GPU with 4 GB of video memory. The size of the brick atlas was set to 64 bricks of size 128^3 each, corresponding to 2 GB of video memory. Because CUDA does not support 3-channel textures, each velocity value had to be padded by an additional "w" component.

I have traced particles starting from the selected seed points in both the uncompressed and the compressed data sets to demonstrate the performance gains that can be achieved via compression. In all experiments, Lagrange6 interpolation was performed; the particle integration times are about 3× higher with Lagrange8, and about 4× lower with Lagrange4 or Catmull-Rom interpolation. When particle tracing was performed on the compressed data, the timings refer to the "High" compression quality. The decompression times for other compression ratios differ only very slightly. To measure the impact of disk I/O, I ran every benchmark a second time, so that all required data was already cached in CPU memory. With uncompressed data, however, this was only possible for the dense scenario in Iso and MHD; in all other cases, the size of the working set exceeded the available CPU memory. Table 6.3 lists the time required for running each scenario, and Table 6.4 lists the sizes of the corresponding working sets.

It can be seen that the use of compression facilitates the tracing of thousands of characteristic lines within seconds in the dense seeding scenario in Iso and MHD. In the sparse seeding case as well as in Mix and Hom, the required time is around an order of magnitude higher. The reason becomes clear when looking at the size of the working sets, which are larger by roughly the same factor in those cases.

Without compression, the overall system performance is clearly limited by disk bandwidth. In particular, in Iso sparse, MHD sparse, and Hom, the working set is so much larger than main memory (cf. Table 6.4) that some bricks had to be loaded from disk multiple times. Even when all required data is already cached in

Table 6.3: Times in seconds for computing stream lines, both for the cached case (*C*) and the uncached case including disk access times (*U*). Individual times for uploading the data to the GPU (*Upl*, including decompression), particle integration (*Int*), and disk I/O (*IO*, overlapping *Upl* and *Int*) are listed separately.

Scenario	Quality	U	C	Upl	Int	IO
Iso dense	High	2.3	1.4	0.8	0.6	1.9
	Uncomp	18.6	1.3	0.7	0.6	17.8
Iso sparse	High	21.6	16.4	12.4	3.8	14.9
	Uncomp	156.9	n/a	10.7	3.8	156.4
MHD dense	High	3.4	2.3	1.4	0.8	2.8
	Uncomp	26.3	2.3	1.4	0.8	25.6
MHD sparse	High	19.9	16.2	11.8	4.2	13.6
	Uncomp	139.7	n/a	10.4	4.2	138.8
Mix	High	19.8	18.6	8.8	9.4	10.0
	Uncomp	72.1	n/a	9.8	9.4	68.1
Hom	High	63.7	63.5	46.9	16.1	16.2
	Uncomp	890.0	n/a	60.9	16.1	886.0

Table 6.4: Working set sizes in both compressed (*High*) and uncompressed (*Uncomp*) form. Also shown is the number of bricks in the working set ($\#B$) as well as the number of brick uploads during particle integration ($\#U$).

Scenario	High	Uncomp	#B	#U
Iso dense	165.9 MB	2155.5 MB	92	92
Iso sparse	1280.6 MB	15058.9 MB	728	1341
MHD dense	243.8 MB	3231.0 MB	138	155
MHD sparse	1095.5 MB	15066.5 MB	729	1298
Mix	745.0 MB	9322.1 MB	408	1093
Hom	1174.3 MB	70836.0 MB	3023	7164

CPU memory (which was only possible in the dense scenario in Iso and MHD), the performance of the compressed and uncompressed cases is very similar—the runtime overhead caused by the additional decompression step is very minor.

It is clear that when tracing path lines, the working sets are much larger because often many different time steps of the same spatial brick are required. In particular, in the two time-dependent data sets the temporal distance between successive time steps is extremely small: 0.002 time units for Iso, 0.0025 for MHD. Because of this, the time required for path line computation is spent almost exclusively on disk-to-CPU data transfer and GPU decompression, and less than 1% of the total time is spent on the actual particle integration. For example, tracing a set of path lines with the dense seeding configuration through Iso takes about 6 minutes, with a working set size of over 25 GB of compressed data. In the uncompressed setting, the working set comprises over 300 GB. Correspondingly, tracing these path lines in the uncompressed data set takes almost an hour, and most of that time is spent on disk I/O. In MHD, the time required for path line tracing is similar; in all cases, the time scales proportionally to the working set size.

Comparison to Previous Work

To the best of my knowledge, all previous techniques for particle tracing in very large flow fields have employed large compute clusters. Pugmire et al. [PCG*09] have used 512 CPUs to trace 10K stream lines in two steady flow fields comprising 512 million grid cells each. They report wall times of 10 to 100 seconds. Camp et al. [CGC*11] later improved those timings to a few seconds for tracing thousands of stream lines on 128 cores. Nouanesengsy et al. [NLS11] achieve timings between 10 and 100 seconds using 4096 cores for the computation of 256K stream lines in regular grids of up to 1.67 billion grid points, but at the cost of an expensive preprocess. Peterka et al. [PRN*11] report computation times of about 20 seconds using 8192 cores for 128K stream lines in a 1024^3 steady flow, and several minutes for 32K lines in a $2304 \times 4096 \times 4096$ steady flow. In contrast to all other mentioned approaches, they have also addressed large *unsteady* flow fields. In a $1408 \times 1080 \times 1100 \times 32$ unsteady flow, the processing time is several minutes for 16K path lines on 4096 cores.
While an exact performance comparison is not possible due to the different data

sets and interpolation/integration schemes used, an order-of-magnitude comparison reveals that my method achieves competitive timings to the previous approaches in many cases, particularly in dense seeding scenarios, while making use of only a single desktop PC.

All in all it can be said that due to the use of an effective compression scheme, the performance of particle tracing in extremely large flow fields can be improved significantly. It is clear that due to the immense working set that is required when computing path lines, fully interactive rates cannot be achieved in this case. However, a simple preview mode which shows the already-computed parts of the current trajectories enables the interactive exploration of very large flow fields. For example, the preview allows the user to quickly discard trajectories originating from "uninteresting" seed points, instead guiding the process interactively towards more interesting regions.

6.6 Conclusion

In this chapter, I have presented an out-of-core system for particle tracing in very large and time-dependent flow fields. It does not require a high-performance computing architecture but runs entirely on a desktop PC. Thus, the system can be used on demand by a turbulence researcher to explore data sets and validate hypotheses. I have employed lossy data compression to overcome bandwidth limitations due to the extreme data volumes that have to be processed. In a number of experiments I have demonstrated that compared to interpolation errors, the compression errors do not significantly affect the accuracy of the computed trajectories. In the statistical sense, the quality of the computed trajectories remains in the same order. A performance analysis indicates that my system achieves a throughput that is comparable to that of previous systems running on high-performance architectures.

The most challenging future avenue of research will be the investigation of the effect of lossy data compression in scenarios other than turbulence research. The question will be whether lossy data compression can also be applied to other flow fields without unduly affecting the accuracy of the resulting trajectories. The main difficulty is that for most flow fields a "correct" interpolation scheme is not available, so the interpolation error can not be estimated accurately. However, different criteria

might be found to steer the compression quality, e.g. given confidence intervals for the velocity values.

7 Conclusion and Future Work

In this thesis, I have presented a highly efficient GPU compression technique. After a general introduction to data compression, I have analyzed the available compression algorithms regarding their compression ratio, their performance, and their feasibility for an efficient data-parallel implementation. The analysis identified transform coding using the discrete wavelet transform followed by run-length and Huffman coding as an efficient and effective compression technique. Based on these findings, I have implemented the CUDACOMPRESS library. This library provides optimized implementations of the chosen compression algorithms. It makes use of NVIDIA's CUDA API for an efficient low-level implementation on current GPUs.

I have then presented three applications which employ CUDACOMPRESS. The first was an interactive terrain rendering and editing system. Thanks to an efficient compression layer, it could handle arbitrarily large and highly resolved terrain data. The second application was a turbulence visualization system applying volume rendering techniques to derived properties of the flow field. Here, the use of compression made it possible to visualize data sets so large that even a single time step would not fit into main memory. Finally, I have presented a particle tracing system for very large flow fields. Data compression in combination with a novel GPU caching scheme allowed my system, running on a single desktop PC, to process data sets which previously required the use of a supercomputer.

There are several ways in which CUDACOMPRESS could be further improved. So far, only Cartesian grids are supported. By adding support for other decorrela-

tion techniques besides the DWT, support for data on additional grid types could be added. Regarding performance, cudaCompress was developed and optimized mostly on Fermi-class GPUs (NVIDIA GeForce 400/500 series). Making use of the new features available in current and future GPUs could likely improve the performance even further. On the other hand, a reimplementation using OpenCL would make AMD GPUs available for compression as well, thus significantly increasing the number of supported PCs.

However, the most important and interesting future work will be to identify additional application areas where efficient data compression can push the frontiers of what is possible. A recent work by Reichl et al. [RTW13] is a first example of another application making use of cudaCompress. They employ an octree grid to resample large SPH simulations and visualize them interactively. Compressing the individual octree nodes significantly improves the streaming performance.

In conclusion, I have demonstrated that effective data compression on GPUs is possible. By making my code publicly available, I am enabling others to use my work in their own applications. This significantly eases the handling of large data sets, and thus helps to tackle one of the main challenges in contemporary scientific visualization.

Bibliography

[AB12] Attili A., Bisetti F.: Statistics and scaling of turbulence in a spatially developing mixing layer at $\mathrm{Re}_\lambda = 250$. *Physics of Fluids 24*, 3 (2012), 035109–1–035109–21. doi:10.1063/1.3696302. 129

[ABC*97] Aurell E., Boffetta G., Crisanti A., Paladin G., Vulpiani A.: Predictability in the large: an extension of the concept of Lyapunov exponent. *J. Physics A: Mathematical and General 30*, 1 (1997), 1–26. doi:10.1088/0305-4470/30/1/003. 146

[AG06] Atlan S., Garland M.: Interactive multiresolution editing and display of large terrains. *Computer Graphics Forum 25*, 2 (2006), 211–223. doi:10.1111/j.1467-8659.2006.00936.x. 85

[AGD10] Ammann L., Génevaux O., Dischler J.-M.: Hybrid rendering of dynamic heightfields using ray-casting and mesh rasterization. In *Proc. Graphics Interface (GI)* (2010), pp. 161–168. URL: http://dl.acm.org/citation.cfm?id=1839214.1839243. 84

[AKKG87] Ashurst W., Kerstein R., Kerr R., Gibson C.: Alignment of vorticity and scalar gradient with the strain rate in simulated Navier-Stokes turbulence. *Physics of Fluids 30* (1987), 2343–2353. doi:10.1063/1.866513. 107

[AMM00] Abu-Mostafa Y. S., McEliece R. J.: Maximal codeword lengths in Huffman codes. *Computers & Mathematics with Applications 39*, 11 (2000), 129 – 134. doi:10.1016/S0898-1221(00)00119-X. 71

[ANR74] Ahmed N., Natarajan T., Rao K.: Discrete cosine transform. *IEEE Trans. Comput. C-23*, 1 (1974), 90–93. doi:10.1109/T-C.1974.223784. 25

[Bal09] Balevic A.: Parallel variable-length encoding on GPGPUs. In *Proc. Parallel and Distributed Computing (Euro-Par)* (2009), pp. 26–35. doi:10.1007/978-3-642-14122-5_6. 56

[BGI*13] Balsa Rodriguez M., Gobbetti E., Iglesias Guitián J., Makhinya M., Marton F., Pajarola R., Suter S.: A survey of compressed GPU-based direct volume rendering. In *Eurographics 2013 - STARs* (2013), pp. 117–136. doi:10.2312/conf/EG2013/stars/117-136. 56

[BGMP07] Bettio F., Gobbetti E., Marton F., Pintore G.: High-quality networked terrain rendering from compressed bitstreams. In *Proc. 3D Web Technology (Web3D)* (2007), pp. 37–44. doi:10.1145/1229390.1229396. 57, 84

[BGP09] Bösch J., Goswami P., Pajarola R.: RASTeR: Simple and efficient terrain rendering on the GPU. In *Eurographics 2009 - Areas Papers* (2009), pp. 35–42. doi:10.5167/uzh-29729. 57, 84

[BIMW*10] Brandstetter III W. E., Mahsman J. D., White C. J., Dascalu S. M., Harris Jr. F. C.: Multi-resolution deformation in out-of-core terrain rendering. In *Proc. Computer Applications in Industry and Engineering (CAINE)* (2010), pp. 98–104. 84

[BKHJ01] Bruckschen R., Kuester F., Hamann B., Joy K. I.: Real-time out-of-core visualization of particle traces. In *Proc. IEEE Parallel and Large-Data Visualization and Graphics (PGV)* (2001), pp. 45–50. doi:10.1109/PVGS.2001.964403. 130

[BM89] Balachandar S., Maxey M. R.: Methods for evaluating fluid velocities in spectral simulations of turbulence. *J. Computational Physics 83*, 1 (1989), 96–125. doi:10.1016/0021-9991(89)90224-6. 130

[BN08] Bruneton E., Neyret F.: Real-time rendering and editing of vector-based terrains. *Computer Graphics Forum 27*, 2 (2008), 311–320. doi:10.1111/j.1467-8659.2008.01128.x. 85

[BPN08] Bhattacharjee S., Patidar S., Narayanan P.: Real-time rendering and manipulation of large terrains. In *Proc. Computer Vision, Graphics & Image Processing (ICVGIP)* (2008), pp. 551–559. doi:10.1109/ICVGIP.2008.85. 85

[BR09] Burtscher M., Ratanaworabhan P.: FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput. 58*, 1 (2009), 18–31. doi:10.1109/TC.2008.131. 55, 110

[Bro08] Brown S.: Squish library, version 1.11, 2008. URL: http://code.google.com/p/libsquish/. 77, 93

[BSK*07] BÜRGER K., SCHNEIDER J., KONDRATIEVA P., KRÜGER J., WESTERMANN R.: Interactive visual exploration of unsteady 3D flows. In *Proc. EG/IEEE VGTC Visualization (EuroVis)* (2007). doi:10.2312/VisSym/EuroVis07/251-258. 130

[BTW*12] BÜRGER K., TREIB M., WESTERMANN R., WERNER S., LALESCU C. C., SZALAY A., MENEVEAU C., EYINK G. L.: Vortices within vortices: hierarchical nature of vortex tubes in turbulence, 2012. arXiv:1210.3325. 128

[Can92] CANTWELL B. J.: Exact solution of a restricted Euler equation for the velocity gradient tensor. *Physics of Fluids A 4* (1992), 782–793. doi:10.1063/1.858295. 107

[CDF92] COHEN A., DAUBECHIES I., FEAUVEAU J.-C.: Biorthogonal bases of compactly supported wavelets. *Comm. Pure and Applied Mathematics 45*, 5 (1992), 485–560. doi:10.1002/cpa.3160450502. 35, 89, 91, 102

[CGC*11] CAMP D., GARTH C., CHILDS H., PUGMIRE D., JOY K.: Streamline integration using MPI-hybrid parallelism on a large multicore architecture. *IEEE Trans. Vis. Comput. Graphics 17*, 11 (2011), 1702–1713. doi:10.1109/TVCG.2010.259. 131, 132, 154

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. ACM SIGGRAPH Interactive 3D Graphics and Games (I3D)* (2009). doi:10.1145/1507149.1507152. 102, 121

[Col13] COLLET Y.: LZ4 library, version 1.4, 2013. URL: https://code.google.com/p/lz4/. 24, 55

[CPC90] CHONG M. S., PERRY A. E., CANTWELL B. J.: A general classification of three-dimensional flow fields. *Physics of Fluids 2* (1990), 765–777. doi:10.1063/1.857730. 107

[CT65] COOLEY J. W., TUKEY J. W.: An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation 19* (1965), 297–301. doi:10.1090/S0025-5718-1965-0178586-1. 27

[Dau88] DAUBECHIES I.: Orthonormal bases of compactly supported wavelets. *Comm. Pure and Applied Mathematics 41* (1988), 909–996. doi:10.1002/cpa.3160410705. 34

[dCB09] DE CARPENTIER G. J. P., BIDARRA R.: Interactive GPU-based procedural heightfield brushes. In *Proc. Foundations of Digital Games (FDG)* (2009), pp. 55–62. doi:10.1145/1536513.1536532. 88

[Deu96] DEUTSCH P.: DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), 1996. URL: http://www.ietf.org/rfc/rfc1951.txt. 23

[DKW09] Dick C., Krüger J., Westermann R.: GPU ray-casting for scalable terrain rendering. In *Eurographics 2009 - Areas Papers* (2009), pp. 43–50. URL: http://diglib.eg.org/EG/DL/conf/EG2009/areas/043-050.pdf. 87

[DP80] Dormand J. R., Prince P. J.: A family of embedded Runge-Kutta formulae. *J. Computational and Applied Mathematics 6*, 1 (1980), 19–26. doi:10.1016/0771-050X(80)90013-3. 148

[DS98] Daubechies I., Sweldens W.: Factoring wavelet transforms into lifting steps. *J. Fourier Analysis and Applications 4*, 3 (1998), 247–269. doi:10.1007/BF02476026. 40

[DSW09] Dick C., Schneider J., Westermann R.: Efficient geometry compression for GPU-based decoding in realtime terrain rendering. *Computer Graphics Forum 28*, 1 (2009), 67–83. doi:10.1111/j.1467-8659.2008.01298.x. 57, 84, 85

[DU13] Doychev T., Uhlmann M.: Settling of finite-size particles in an ambient fluid: A numerical study. In *Proc. Multiphase Flow (ICMF)* (2013). URL: http://www-turbul.ifh.uni-karlsruhe.de/uhlmann/particle/report/icmf13.pdf. 129

[ECW04] Ebrahimi F., Chamik M., Winkler S.: JPEG vs. JPEG 2000: An objective comparison of image encoding quality. *Applications of Digital Image Processing XXVII/Proc. SPIE 5558* (2004), 300–308. doi:10.1117/12.564835. 48

[EGM04] Ellsworth D., Green B., Moran P.: Interactive terascale particle visualization. In *Proc. IEEE Visualization* (2004), pp. 353–360. doi:10.1109/VISUAL.2004.55. 130

[EHK*06] Engel K., Hadwiger M., Kniss J. M., Rezk-Salama C., Weiskopf D.: *Real-Time Volume Graphics*. A K Peters, Ltd., 2006. doi:10.1201/b10629. 56

[EM94] Eiter T., Mannila H.: *Computing Discrete Frèchet Distance*. Tech. Rep. CD-TR 94/64, Technische Universität Wien, 1994. URL: http://www.kr.tuwien.ac.at/staff/eiter/et-archive/cdtr9464.pdf. 146

[EVL*13] Eyink G., Vishniac E., Lalescu C., Aluie H., Kanov K., Bürger K., Burns R., Meneveau C., Szalay A.: Flux-freezing breakdown in high-conductivity magnetohydrodynamic turbulence. *Nature 497*, 7450 (2013), 466–469. doi:10.1038/nature12128. 123

[FBS07] Fraedrich R., Bauer M., Stamminger M.: Sequential data compression of very large data in volume rendering. In *Proc. Vision, Modeling and Visualization (VMV)* (2007), pp. 41–50. 56

[FCS*10] Fogal T., Childs H., Shankar S., Krüger J., Bergeron R. D., Hatcher P.: Large data visualization on distributed memory multi-GPU clusters. In *Proc. High Performance Graphics (HPG)* (2010), pp. 57–66. doi:10.2312/EGGH/HPG10/057-066. 101

[FE99] Franzen R., Eastman Kodak Company: Kodak lossless true color image suite, 1999. URL: http://r0k.us/graphics/kodak/. 29, 37, 42, 61, 77

[FJ05] Frigo M., Johnson S. G.: The design and implementation of FFTW3. *Proc. IEEE 93*, 2 (2005), 216–231. doi:10.1109/JPROC.2004.840301. 143

[FM12] Fout N., Ma K.-L.: An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Trans. Vis. Comput. Graphics 18*, 12 (2012), 2295–2304. doi:10.1109/TVCG.2012.194. 140

[FM13] Fout N., Ma K.-L.: Fuzzy volume rendering. *IEEE Trans. Vis. Comput. Graphics 18*, 12 (2013), 2335–2344. doi:10.1109/TVCG.2012.227. 141

[FMA05] Fout N., Ma K.-L., Ahrens J.: Time-varying, multivariate volume data reduction. In *Proc. ACM Applied Computing (SAC)* (2005), pp. 1224–1230. doi:10.1145/1066677.1066953. 56

[Fri95] Frisch U.: *Turbulence, the legacy of A.N. Kolmogorov*. Cambridge University Press, 1995. 108

[FSK13] Fogal T., Schiewe A., Krüger J.: An analysis of scalable GPU-based ray-guided volume rendering. In *Proc. IEEE Large-Scale Data Analysis and Visualization (LDAV)* (2013), pp. 43–51. doi:10.1109/LDAV.2013.6675157. 55, 102

[FWH*09] Fürst N., Weiss A., Heide M., Papandreou S., Balevic A.: CUJ2K library, version 1.1, 2009. URL: http://cuj2k.sourceforge.net. 95

[GMC*06] Gobbetti E., Marton F., Cignoni P., Di Benedetto M., Ganovelli F.: C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum 25*, 3 (2006), 333–342. doi:10.1111/j.1467-8659.2006.00952.x. 57, 84

[GMI08] Gobbetti E., Marton F., Iglesias Guitián J.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7-9 (2008), 797–806. doi:10.1007/s00371-008-0261-9. 102, 121

[Gol66] Golomb S. W.: Run-length encodings. *IEEE Trans. Inf. Theory 12*, 3 (1966), 399–401. doi:10.1109/TIT.1966.1053907. 15

[GS01] Guthe S., Strasser W.: Real-time decompression and visualization of animated volume data. In *Proc. IEEE Visualization* (2001), pp. 349–356. doi:10.1109/VISUAL.2001.964531. 56, 113

[GWGS02] Guthe S., Wand M., Gonser J., Strasser W.: Interactive rendering of large volume data sets. In *Proc. IEEE Visualization* (2002), pp. 53–60. doi:10.1109/VISUAL.2002.1183757. 56, 111

[Hal05] Haller G.: An objective definition of a vortex. *J. Fluid Mechanics 525* (2005), 1–26. doi:10.1017/S0022112004002526. 107

[HB13] Hoberock J., Bell N.: Thrust: A parallel template library, version 1.7.0, 2013. URL: http://thrust.github.io. 54, 60

[HBC10] Howison M., Bethel E. W., Childs H.: MPI-hybrid parallelism for volume rendering on large, multi-core systems. In *Proc. EG Parallel Graphics and Visualization (EGPGV)* (2010). doi:10.2312/EGPGV/EGPGV10/001-010. 101

[HCP02] He Y., Cremer J. F., Papelis Y. E.: Real-time extendible-resolution display of on-line dynamic terrain. In *Proc. Graphics Interface (GI)* (2002), pp. 151–160. URL: http://www.graphicsinterface.org/proceedings/2002/139/. 84

[HOS*13] Harris M., Owens J. D., Sengupta S., Tzeng S., Zhang Y., Davidson A., Patel R.: CUDPP: CUDA data-parallel primitives library, version 2.1, 2013. URL: http://cudpp.github.io. 54, 60

[HSO07] Harris M., Sengupta S., Owens J. D.: Parallel prefix sum (scan) with cuda. In *GPU Gems 3*. Addison Wesley, 2007. 60

[Huf52] Huffman D. A.: A method for the construction of minimum-redundancy codes. *Proc. Institute of Radio Engineers 40*, 9 (1952), 1098–1101. doi:10.1109/JRPROC.1952.273898. 9

[HWM88] Hunt J., Wray A., Moin P.: Eddies, streams, and convergence zones in turbulent flows. In *Proc. Studying Turbulence Using Numerical Simulation Databases* (1988), pp. 193–208. URL: http://ctr.stanford.edu/Summer/201306111537.pdf. 107

[IGK09] Ishihara T., Gotoh T., Kaneda Y.: Study of high-Reynolds number isotropic turbulence by direct numerical simulation. *Annu. Rev. Fluid Mechanics 41* (2009), 165–180. doi:10.1146/annurev.fluid.010908.165203. 98

[ILS05] Isenburg M., Lindstrom P., Snoeyink J.: Lossless compression of predicted floating-point geometry. *Computer Aided Design 37*, 8 (2005), 869–877. doi:10.1016/j.cad.2004.09.015. 140

[Ima13] ImageMagick Studio LLC: ImageMagick library, version 6.8.7-10-q16, 2013. URL: http://www.imagemagick.org. 77, 93

[INH03] Iourcha K., Nayak K., Hong Z.: System and method for fixed-rate block-based image compression with inferred pixel values, 2003. US Patent 6658146. 57

[ITU92] ITU: CCITT Recommendation T.81 (09/92) / ISO/IEC 10918-1: 1994: Digital compression and coding of continuous-tone still images, 1992. URL: http://www.w3.org/Graphics/JPEG/itu-t81.pdf. 44

[ITU02] ITU: ITU-T Recommendation T.800 (08/02) / ISO/IEC 15444-1: 2002: JPEG2000 image coding system, 2002. URL: http://www.itu.int/rec/T-REC-T.800-200208-I/en. 47

[ITU03] ITU: ITU-T Recommendation H.264 / ISO/IEC 14496-10: Information technology – jpeg xr image coding system – image coding specification, 2003. URL: http://www.itu.int/rec/T-REC-H.264. 28, 29

[ITU09] ITU: ITU-T Recommendation T.832 / ISO/IEC 29199-2: Advanced video coding for generic audiovisual services, 2009. URL: http://www.itu.int/rec/T-REC-T.832. 28

[JCH01] Jensen A., Cour-Harbo A. l.: *Ripples in Mathematics: The Discrete Wavelet Transform*. Springer, 2001. doi:10.1007/978-3-642-56702-5. 30

[JH95] Jeong J., Hussain F.: On the identification of a vortex. *J. Fluid Mechanics 285* (1995), 69–94. doi:10.1017/S0022112095000462. 107

[Kak13] Kakadu Software: KDU-S7: Kakadu SDK with speed pack, version 7.0, 2013. URL: http://www.kakadusoftware.com. 95

[KKKW05] Krüger J., Kipfer P., Kondratieva P., Westermann R.: A particle system for interactive visualization of 3D flows. *IEEE Trans. Vis. Comput. Graphics 11*, 6 (2005), 744–756. doi:10.1109/TVCG.2005.87. 130

[Kra05] Kraus M.: Scale-invariant volume rendering. In *Proc. IEEE Visualization* (2005), pp. 295–302. doi:10.1109/VIS.2005.88. 104

[KS10] Katz J., Sheng J.: Applications of holography in fluid mechanics and particle dynamics. *Annu. Rev. Fluid Mechanics 42* (2010), 531–555. doi:10.1146/annurev-fluid-121108-145508. 98

[KW03] Krüger J., Westermann R.: Acceleration techniques for GPU-based volume rendering. In *Proc. IEEE Visualization* (2003), pp. 287–292. doi:10.1109/VISUAL.2003.1250384. 103

[Lan94] Lane D. A.: UFAT—a particle tracer for time-dependent flow fields. In *Proc. IEEE Visualization* (1994), pp. 257–264. doi:10.1109/VISUAL.1994.346311. 130

[LC10] Lindstrom P., Cohen J. D.: On-the-fly decompression and rendering of multiresolution terrain. In *Proc. ACM SIGGRAPH Interactive 3D Graphics and Games (I3D)* (2010), pp. 65–73. doi:10.1145/1730804.1730815. 57, 84

[LHD*04] Laramee R. S., Hauser H., Doleisch H., Vrolijk B., Post F. H., Weiskopf D.: The state of the art in flow visualization: Dense and texture-based techniques. *Computer Graphics Forum 23*, 2 (2004), 203–221. doi:10.1111/j.1467-8659.2004.00753.x. 129

[LI06] Lindstrom P., Isenburg M.: Fast and efficient compression of floating-point data. *IEEE Trans. Vis. Comput. Graphics 12*, 5 (2006), 1245–1250. doi:10.1109/TVCG.2006.143. 55, 110, 140

[lib13] libjpeg-turbo Project: libjpeg-turbo library, version 1.3.0, 2013. URL: http://www.libjpeg-turbo.org. 94

[LK10] Lambers M., Kolb A.: Dynamic terrain rendering. *3D Research 1*, 4 (2010), 1–8. doi:10.1007/3DRes.04(2010)01. 84

[LMC02] Lum E. B., Ma K.-L., Clyne J.: A hardware-assisted scalable solution for interactive volume rendering of time-varying data. *IEEE Trans. Vis. Comput. Graphics 8*, 3 (2002), 286–270. doi:10.1109/TVCG.2002.1021580. 56

[LPW*08] Li Y., Perlman E., Wan M., Yang Y., Meneveau C., Burns R., Chen S., Szalay A., Eyink G.: A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence. *J. Turbulence 9* (2008), N31. doi:10.1080/14685240802376389. 100, 129, 130, 141

[Mal89] Mallat S. G.: A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. Pattern Anal. Mach. Intell. 11*, 7 (1989), 674–693. doi:10.1109/34.192463. 30

[Mal98] Malvar H.: Biorthogonal and nonuniform lapped transforms for transform coding with reduced blocking and ringing artifacts. *IEEE Trans. Signal Process. 46*, 4 (1998), 1043–1053. doi:10.1109/78.668555. 28

[Mal08] Mallat S.: *A Wavelet Tour of Signal Processing: The Sparse Way*, third ed. Academic Press, 2008. 30

[MAWM11] Moloney B., Ament M., Weiskopf D., Möller T.: Sort-first parallel volume rendering. *IEEE Trans. Vis. Comput. Graphics 17*, 8 (2011), 1164–1177. doi:10.1109/TVCG.2010.116. 101

[MLP*10] McLoughlin T., Laramee R. S., Peikert R., Post F. H., Chen M.: Over two decades of integration-based, geometric flow visualization. *Computer Graphics Forum 29*, 6 (2010), 1807–1829. doi:10.1111/j.1467-8659.2010.01650.x. 129

[MOCS98] Martin J., Ooi A., Chong M. S., Soria J.: Dynamics of the velocity gradient tensor invariants in isotropic turbulence. *Physics of Fluids 10* (1998), 2336–2346. doi:10.1063/1.869752. 107

[Moo65] Moore G. E.: Cramming more components onto integrated circuits. *Electronics 38*, 8 (1965), 114–117. Reprinted in *Proc. IEEE 86*, 1 (1998), 82–85. doi:10.1109/jproc.1998.658762. 1

[MS89] Malvar H., Staelin D.: The LOT: Transform coding without blocking effects. *IEEE Trans. Acoust., Speech, Signal Process. 37*, 4 (1989), 553–559. doi:10.1109/29.17536. 28

[MS03a] Malvar H., Sullivan G.: *Transform, Scaling & Color Space Impact of Professional Extensions*. Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG (ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q.6), 2003. URL: http://wftp3.itu.int/av-arch/jvt-site/2003_05_Geneva/JVT-H031.doc. 43

[MS03b] Malvar H., Sullivan G.: *YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range*. Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG (ISO/IEC JTC1/SC29/WG11 and ITU-T SG16 Q.6), 2003. URL: http://wftp3.itu.int/av-arch/jvt-site/2003_09_SanDiego/JVT-I014.doc. 43, 89

[Mur12] Murray L.: GPU acceleration of Runge-Kutta integrators. *IEEE Trans. Parallel Distrib. Syst. 23*, 1 (2012), 94–101. doi:10.1109/TPDS.2011.61. 130

[NIH06] Nagayasu D., Ino F., Hagihara K.: Two-stage compression for fast volume rendering of time-varying scalar data. In *Proc. Computer Graphics and Interactive Techniques (GRAPHITE)* (2006), pp. 275–284. doi:10.1145/1174429.1174478. 57

[NIH08] Nagayasu D., Ino F., Hagihara K.: A decompression pipeline for accelerating out-of-core volume rendering of time-varying data. *Computers & Graphics 32*, 3 (2008), 350–362. doi:10.1016/j.cag.2008.04.007. 57

[NLS11] Nouanesengsy B., Lee T.-Y., Shen H.-W.: Load-balanced parallel streamline generation on large scale vector fields. *IEEE Trans. Vis. Comput. Graphics 17*, 12 (2011), 1785–1794. doi:10.1109/TVCG.2011.219. 132, 138, 154

[NS01] Nguyen K. G., Saupe D.: Rapid high quality compression of volume data for visualization. *Computer Graphics Forum 20*, 3 (2001), 49–57. doi:10.1111/1467-8659.00497. 56, 111

[NVI10] NVIDIA Corp.: NVIDIA Texture Tools, version 2.08, 2010. URL: http://developer.nvidia.com/gpu-accelerated-texture-compression. 95

[NVI13a] NVIDIA Corp.: *CUDA C Best Practices Guide, version 5.5*, 2013. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf. 55

[NVI13b] NVIDIA Corp.: *CUDA C Programming Guide, version 5.5*, 2013. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. 55

[OB11] O'Neil M. A., Burtscher M.: Floating-point data compression at 75 Gb/s on a GPU. In *Proc. General Purpose Processing on Graphics Processing Units (GPGPU)* (2011), pp. 7:1–7:7. doi:10.1145/1964179.1964189. 55

[Obe11] Oberhumer M. F. X. J.: LZO library, version 2.06, 2011. URL: http://www.oberhumer.com/opensource/lzo/. 24, 55

[OBGB11] Olano M., Baker D., Griffin W., Barczak J.: Variable bit rate GPU texture decompression. *Computer Graphics Forum 30*, 4 (2011), 1299–1308. doi:10.1111/j.1467-8659.2011.01989.x. 56, 57

[OK08] Obukhov A., Kharlamov A.: *Discrete Cosine Transform for 8x8 Blocks with CUDA*. NVIDIA Corp., 2008. URL: http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/dct8x8/doc/dct8x8.pdf. 58

[OS11] Ozsoy A., Swany M.: CULZSS: LZSS lossless data compression on CUDA. In *Proc. IEEE Cluster Computing (CLUSTER)* (2011), pp. 403–411. doi:10.1109/CLUSTER.2011.52. 56

[OSC12] Ozsoy A., Swany M., Chauhan A.: Pipelined parallel LZSS for streaming data compression on GPGPUs. In *Proc. IEEE Parallel and Distributed Systems (ICPADS)* (2012), pp. 37–44. doi:10.1109/ICPADS.2012.16. 56, 58

[Pas76] Pasco R. C.: *Source coding algorithms for fast data compression.* PhD thesis, Stanford University, 1976. 17

[PCG*09] Pugmire D., Childs H., Garth C., Ahern S., Weber G. H.: Scalable computation of streamlines on very large datasets. In *Proc. High Performance Computing, Networking, Storage and Analysis (SC)* (2009), pp. 16:1–16:12. doi:10.1145/1654059.1654076. 131, 154

[PG07] Pajarola R., Gobbetti E.: Survey of semi-regular multiresolution models for interactive terrain rendering. *Visual Computer 23*, 8 (2007), 583–605. doi:10.1007/s00371-007-0163-2. 57, 81, 84

[PM93] Pennebaker W. B., Mitchell J. L.: *JPEG: Still Image Data Compression Standard*. Kluwer Academic Publishers, 1993. 44

[Pod07a] Podlozhnyuk V.: *Histogram calculation in CUDA*. NVIDIA Corp., 2007. URL: http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/histogram/doc/histogram.pdf. 59, 68, 69

[Pod07b] Podlozhnyuk V.: *Image Convolution with CUDA*. NVIDIA Corp., 2007. URL: http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/convolutionSeparable/doc/convolutionSeparable.pdf. 64

[PRN*11] Peterka T., Ross R., Nouanesengsy B., Lee T.-Y., Shen H.-W., Kendall W., Huang J.: A study of parallel particle tracing for steady-state and time-varying flow fields. In *Proc. IEEE Parallel & Distributed Processing (IPDPS)* (2011), pp. 580–591. doi:10.1109/IPDPS.2011.62. 132, 154

[PSM*07] Pradhan B., Sandeep K., Mansor S., Ramli A. R., Sharif A. R. B. M.: Second generation wavelets based GIS terrain data compression using Delaunay triangulation. *Engineering Computations 24*, 2 (2007), 200–213. doi:10.1108/02644400710729572. 57

[PV95] Perlin K., Velho L.: Live paint: Painting with procedural multiscale textures. In *Proc. Computer Graphics and Interactive Techniques (SIGGRAPH)* (1995), pp. 153–160. doi:10.1145/218380.218437. 85

[PVH*03] Post F. H., Vrolijk B., Hauser H., Laramee R. S., Doleisch H.: The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum 22*, 4 (2003), 775–792. doi:10.1111/j.1467-8659.2003.00723.x. 129

[PZM*12] Patel R. A., Zhang Y., Mak J., Davidson A., Owens J. D.: Parallel lossless data compression on the GPU. In *Proc. Innovative Parallel Computing (InPar)* (2012), pp. 1–9. doi:10.1109/InPar.2012.6339599. 56, 69

[Raw11] Rawzor: The new test images, 2011. URL: http://www.imagecompression.info/test_images/. 61, 77

[RHB94] Rovelstad A. L., Handler R. A., Bernard P. S.: The effect of interpolation errors on the Lagrangian analysis of simulated turbulent channel flow. *J. Computational Physics 110*, 1 (1994), 190–195. doi:10.1006/jcph.1994.1015. 130, 141

[Ric79] Rice R. F.: *Some Practical Universal Noiseless Coding Techniques.* Tech. Rep. JPL Publication 79-22, Jet Propulsion Laboratory, California Institute of Technology, 1979. URL: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19790014634_1979014634.pdf. 15

[Ris76] Rissanen J.: Generalized kraft inequality and arithmetic coding. *IBM J. Research and Development 20*, 3 (1976), 198–203. doi:10.1147/rd.203.0198. 17

[RL79] Rissanen J., Langdon G.G. J.: Arithmetic coding. *IBM J. Research and Development 23*, 2 (1979), 149–162. doi:10.1147/rd.232.0149. 17

[RP71] Rice R. F., Plaunt J. R.: Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Trans. Circuit Theory 19*, 6 (1971), 889–897. doi:10.1109/TCOM.1971.1090789. 15

[RtHRS08] Ruijters D., ter Haar Romeny B. M., Suetens P.: Efficient GPU-based texture interpolation using uniform B-splines. *J. Graphics, GPU, and Game Tools 13*, 4 (2008), 61–69. doi:10.1080/2151237X.2008.10129269. 103

[RTW13] Reichl F., Treib M., Westermann R.: Visualization of big SPH simulations via compressed octree grids. In *Proc. IEEE Big Data* (2013), pp. 71–78. doi:10.1109/BigData.2013.6691717. 158

[SA97] Sreenivasan K., Antonia R.: The phenomenology of small-scale turbulence. *Annu. Rev. Fluid Mechanics 29* (1997), 435–472. doi:10.1146/annurev.fluid.29.1.435. 108

[Say12] Sayood K.: *Introduction to Data Compression*, fourth ed. Morgan Kaufmann Publ. Inc., 2012. doi:10.1016/B978-0-12-415796-5.00006-5. 5, 10, 11

[SBK06] Schirski M., Bischof C., Kuhlen T.: Interactive particle tracing on tetrahedral grids using the GPU. In *Proc. Vision, Modeling, and Visualization (VMV)* (2006), pp. 153–160. 130

[SGvR*03] Schirski M., Gerndt A., van Reimersdahl T., Kuhlen T., Adomeit P., Lang O., Pischinger S., Bischof C.: ViSTA FlowLib – a framework for interactive visualization and exploration of unsteady flows in virtual environments. In *Proc. EG Virtual Environments (EGVE)* (2003), pp. 77–86. doi:10.1145/769953.769963. 130

[SH05] Sigg C., Hadwiger M.: Fast third-order texture filtering. In *GPU Gems 2*. Addison-Wesley, 2005, pp. 313–329. URL: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter20.html. 103

[Sha48a] Shannon C. E.: A mathematical theory of communication. *The Bell System Technical Journal 27*, 3 (1948), 379–423. Part 1. doi:10.1002/j.1538-7305.1948.tb01338.x. 7

[Sha48b] Shannon C. E.: A mathematical theory of communication. *The Bell System Technical Journal 27*, 4 (1948), 623–656. Part 2. doi:10.1002/j.1538-7305.1948.tb00917.x. 7

[SK07] Shams R., Kennedy R. A.: Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. Signal Processing and Communication Systems (ICSPCS)* (2007), pp. 418–422. URL: http://users.cecs.anu.edu.au/~ramtin/papers/2007/ICSPCS_2007.pdf. 69

[SM10] Salomon D., Motta G.: *Handbook of Data Compression*, fifth ed. Springer, 2010. doi:10.1007/978-1-84882-903-9. 5

[SS82] Storer J. A., Szymanski T. G.: Data compression via textual substitution. *J. ACM 29*, 4 (1982), 928–951. doi:10.1145/322344.322346. 22

[Ste46] Stevens S. S.: On the theory of scales of measurement. *Science 103*, 2684 (1946), 677–680. doi:10.1126/science.103.2684.677. 24

[SW03] Schneider J., Westermann R.: Compression domain volume rendering. In *Proc. IEEE Visualization* (2003), pp. 293–300. doi:10.1109/VISUAL.2003.1250385. 56, 110

[Swe98] Sweldens W.: The lifting scheme: A construction of second generation wavelets. *SIAM J. Mathematical Analysis 29*, 2 (1998), 511–546. doi:10.1137/S0036141095289051. 38

[TBR*12] Treib M., Bürger K., Reichl F., Meneveau C., Szalay A., Westermann R.: Turbulence visualization at the terascale on desktop PCs. *IEEE Trans. Vis. Comput. Graphics 18*, 12 (2012), 2169–2177. doi:10.1109/TVCG.2012.274. 97

[TGE97] Teitzel C., Grosso R., Ertl T.: Efficient and reliable integration methods for particle tracing in unsteady flows on discrete meshes. In *Visualization in Scientific Computing*. Springer, 1997, pp. 31–41. URL: http://cumbia.visus.uni-stuttgart.de/eng/research/pub/pub1997/egvis97teitzel.pdf. 129

[TM01] Taubman D. S., Marcellin M. W.: *JPEG2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publ., 2001. 38, 44, 111

[TRAW12] Treib M., Reichl F., Auer S., Westermann R.: Interactive editing of gigasample terrain fields. *Computer Graphics Forum 31*, 2 (2012), 383–392. doi:10.1111/j.1467-8659.2012.03017.x. 81, 111, 122

[Tre13] Treib M.: cudaCompress: GPU data compression using CUDA, 2013. URL: https://github.com/m0bl0/cudaCompress. 4, 54, 71, 73

[TSP*08] Tenllado C., Setoain J., Prieto M., Pinuel L., Tirado F.: Parallel implementation of the 2D discrete wavelet transform on graphics processing units: Filter bank versus lifting. *IEEE Trans. Parallel Distrib. Syst. 19*, 3 (2008), 299–310. doi:10.1109/TPDS.2007.70716. 58, 64

[TW13] Treib M., Westermann R.: Compression and heuristic caching for GPU particle tracing in turbulent vector fields. *Submitted for publication* (2013). 125

[Uhl05] Uhlmann M.: An immersed boundary method with direct forcing for the simulation of particulate flows. *J. Computational Physics 209*, 2 (2005), 448–476. doi:10.1016/j.jcp.2005.03.017. 129

[vdLJR11] van der Laan W. J., Jalba A. C., Roerdink J. B. T. M.: Accelerating wavelet lifting on graphics hardware using CUDA. *IEEE Trans. Parallel Distrib. Syst. 22*, 1 (2011), 132–146. doi:10.1109/TPDS.2010.143. 58, 64

[vW06] van Waveren J. M. P.: *Real-Time Texture Streaming & Decompression*. Tech. rep., id Software, Inc., 2006. URL: http://software.intel.com/file/17248/. 71

[vWC07] van Waveren J. M. P., Castaño I.: *Real-Time YCoCg-DXT Compression*. Tech. rep., id Software, Inc. and NVIDIA Corp., 2007. URL: http://developer.download.nvidia.com/whitepapers/2007/Real-Time-YCoCg-DXT-Compression/Real-TimeYCoCg-DXTCompression.pdf. 57, 82

[Wel84] Welch T.: A technique for high-performance data compression. *Computer 17*, 6 (1984), 8–19. doi:10.1109/MC.1984.1659158. 23

[Wes94] Westermann R.: A multiresolution framework for volume rendering. In *Proc. Volume Visualization (VVS)* (1994), pp. 51–58. doi:10.1145/197938.197963. 56, 111

[WLHW07] Wong T.-T., Leung C.-S., Heng P.-A., Wang J.: Discrete wavelet transform on consumer-level graphics hardware. *IEEE Trans. Multimedia 9*, 3 (2007), 668–673. doi:10.1109/TMM.2006.887994. 58, 64

[WV10] Wallace J., Vukoslavcevic P.: Measurement of the velocity gradient tensor in turbulent flows. *Annu. Rev. Fluid Mechanics 42* (2010), 157–181. doi:10.1146/annurev-fluid-121108-145445. 98

[WZY08] Wang X., Zheng X., Yin Q.: Large scale terrain compression and real-time rendering based on wavelet transform. In *Proc. Computational Intelligence and Security (CIS)* (2008), vol. 2, pp. 489–493. doi:10.1109/CIS.2008.115. 57

[YL95] Yeo B.-L., Liu B.: Volume rendering of DCT-based compressed 3D scalar data. *IEEE Trans. Vis. Comput. Graphics 1*, 1 (1995), 29–43. doi:10.1109/2945.468390. 56, 111

[YP88] Yeung P. K., Pope S. B.: An algorithm for tracking fluid particles in numerical simulations of homogeneous turbulence. *J. Computational Physics 79*, 2 (1988), 373–416. doi:10.1016/0021-9991(88)90022-8. 130

[YWM07] Yu H., Wang C., Ma K.-L.: Parallel hierarchical visualization of large time-varying 3D vector fields. In *Proc. ACM/IEEE Supercomputing (SC)* (2007), pp. 24:1–24:12. doi:10.1145/1362622.1362655. 132

[ZL77] Ziv J., Lempel A.: A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 3 (1977), 337–343. doi:10.1109/TIT.1977.1055714. 21

[ZL78] Ziv J., Lempel A.: Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory 24*, 5 (1978), 530–536. doi:10.1109/TIT.1978.1055934. 21

[ZXM10] Zheng Z., Xu W., Mueller K.: VDVR: Verifiable volume visualization of projection-based data. *IEEE Trans. Vis. Comput. Graphics 16*, 6 (2010), 1515–1524. doi:10.1109/TVCG.2010.211. 141