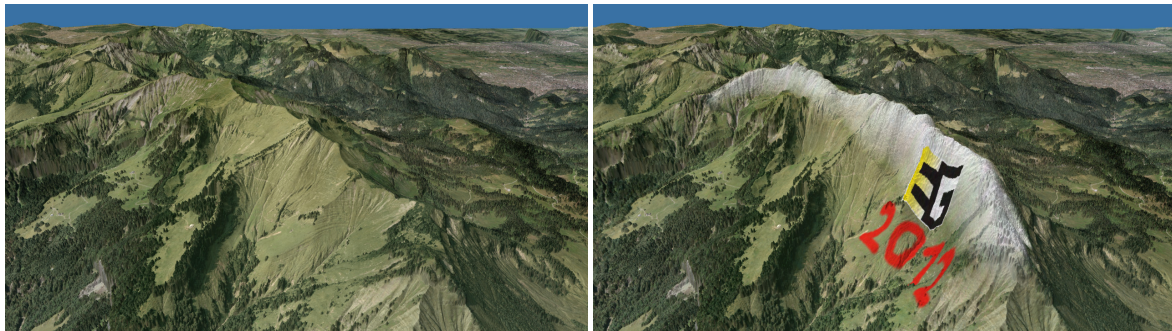# Interactive Editing of GigaSample Terrain Fields

Marc Treib[1], Florian Reichl[1], Stefan Auer[1], and Rüdiger Westermann[1]

[1]Computer Graphics & Visualization Group, Technische Universität München, Germany

Figure 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 using our approach.*

Abstract

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. We present a novel approach for editing gigasample terrain fields at interactive rates and high quality. To achieve high decoding and encoding throughput, we employ a compression scheme for height and pixel maps based on a sparse wavelet representation. On recent GPUs it can encode and decode up to 270 and 730 MPix/s of color data, respectively, at compression rates and quality superior to JPEG, and it achieves more than twice these rates for lossless height field compression. 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. We show the efficiency of our method for interactive editing and continuous level-of-detail rendering of terrain fields comprised of several hundreds of gigasamples.

Categories and Subject Descriptors (according to ACM CCS):   I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism— I.4.2 [Image Processing and Computer Vision]: Compression (Coding)—

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 us 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 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.

Figure 2: *Creating a street: A forest (left) is cleared along a path (middle) to build a street (right).*

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 pre-processing. 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 focussed on alterations of uncompressed terrain, not taking into account disk I/O bandwidth limitations. To the best of our knowledge, interactive visually-guided editing of terrain fields so large that they require compression has not been achieved so far.

Our contribution: We present a novel approach to interactively edit terrain fields which are so large that I/O bandwidth becomes the major bottleneck (see Fig. 1 for an example). To handle such fields efficiently, we propose a data compression scheme that combines an existing GPU realization of wavelet transforms [vdLJR11] with novel GPU approaches to efficiently perform run-length and Huffman encoding and decoding of quantized wavelet coefficients. The scheme has been tailored to exploit the GPU's capabilities via the CUDA API, and, thus, it allows both decoding and encoding of height and color samples to be faster than disk transfer. Special emphasis has been put on high-quality compression, and on efficiently combining level-of-detail editing and rendering. To accomplish this, our internal data representation is based on pixel and height field raster data, and rendering is performed directly on these rasters using ray-casting. Our particular contributions are

- a high-throughput GPU coder which can encode and decode up to 270 and 730 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 edit-

ing 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.

Our paper is structured as follows: In the next section we review work that is related to ours. Sec. 3 gives an overview of the different parts our method is comprised of and outlines their interplay. Following is the discussion of the GPU compression scheme, including details on the specific GPU parallelization of run-length and Huffman coding. Next, we analyze the compression rates, the reconstruction quality, and the coding performance our method can achieve on recent GPUs, and we demonstrate the efficient interplay between data compression and rendering in a prototype system. The paper is concluded with some ideas on future enhancements and additional applications.

2. Related Work

Terrain rendering approaches usually incorporate some form of height field compression to reduce disk and CPU-GPU bandwidth limitations 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 edit-
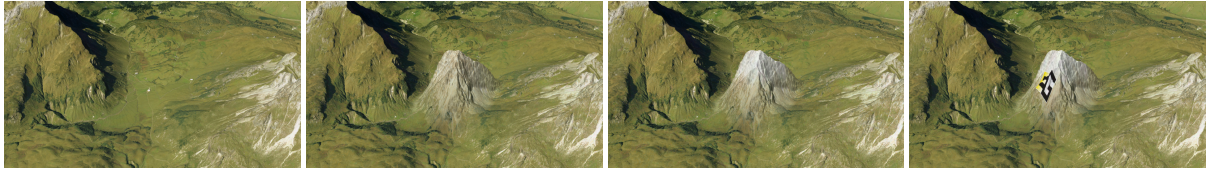
Figure 3: *Placing a mountain using a height stamp, painting snow on top, and using a color stamp to add the EG logo.*

ing 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 our approach.

For the compression of the color-valued terrain orthophoto, the most common scheme in terrain rendering applications is S3TC. It is popular since it enables hardware-supported random access on the GPU. 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 at half the compression rate can be achieved [vWC07]. The possibility to efficiently encode color fields into the S3TC format on the GPU has been demonstrated [vWC07].

An important topic related to our method is data compression using transform coding. In particular, Taubman and Marcellin [TM01] discuss many aspects specific to image compression, including wavelet-based compression and the JPEG/JPEG2000 standards. The capability of wavelet transforms to effectively compress scalar fields has been employed for terrain height fields [PSM*07, WZY08]. Our GPU coder builds in particular upon previous approaches for computing discrete wavelet transforms on the GPU [WLHW07, TSP*08, vdLJR11]. For a given Huffman table, Balevic [Bal09] has shown the efficient GPU realization of a Huffman encoder. For editing, we use concepts similar to those proposed by Perlin and Velho [PV95] for multi-resolution pixel image editing using wavelet transforms.

3. Gigasample Terrain Editing

Before discussing the GPU coder that is used to enable fast encoding and decoding of gigasample terrain fields, we first give an overview of the different components of our prototype editing system as well as their interplay. In particular, we describe the used 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 2048^2, with the leaf nodes corresponding to the original data. Our terrain representation is similar to the one proposed by Dick et al. [DSW09]. In our 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 pre-fetching region around the camera are loaded from disk into CPU memory. The world-space radius of the pre-fetching region is doubled with every coarser level.

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 (two different trees for the height and the color data are stored): 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 Sec. 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.

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. It is first checked, however, whether the data is already resident in GPU memory or was reconstructed at an earlier time and cached on the CPU. In the latter case the data is streamed from the CPU to the GPU. Otherwise, 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 or at least the CPU for as long as possible.

The reconstructed 2D raster data is then 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.

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, we 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). 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.

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, we keep 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
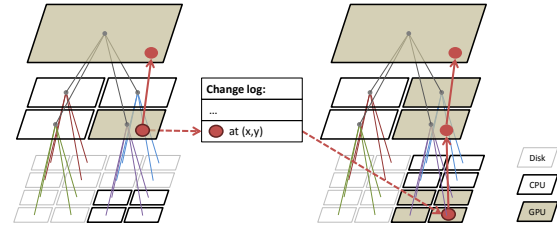

Figure 4: *Propagation of changes. Left: The effect of an editing operation in a visible tile is immediately propagated to the tile's ancestors. The operation is stored in a change log. Right: When a finer tile is required, operations in the change log are applied to this tile, and the modifications are propagated to the ancestors again to ensure consistency.*

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 as described. 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 CPU space is not available any more, or a tile falls out of the pre-fetching region, the modified tiles are written to disk.

To edit the terrain height field and orthophoto we provide several tools, such as a paint brush and a stamp to draw color and/or height offsets (see Fig. 1 and 3), a flatten tool to smooth high-frequency details (see Fig. 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. 2. This variety shows that our approach is flexible enough to integrate more sophisticated terrain editing tools, for instance as proposed by de Carpentier and Bidarra [dCB09].
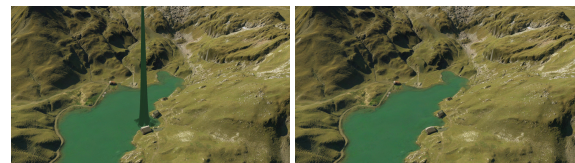
Figure 5: *Interactive flattening to remove scanning artifacts.*

4. Data Compression

In the following, we describe the four different stages our GPU coder is comprised of. We first discuss the compression of RGB pixel data, and we then outline the particular changes to accommodate the processing of scalar scalar-valued height fields. This is followed by a detailed description of the GPU realization of both the encoder and decoder stage.

Color space transform: The RGB color values of each tile's orthophoto are first transformed into the YCoCg color space [MS03] to exploit correlations between the color channels. The conversion de-correlates the channels, i.e., it concentrates most of the energy in the Y (luma) channel, thus improving the compression rate. The YCoCg values are transformed back into RGB values only for display. As in JPEG2000, chroma sub-sampling is not performed.

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. Our implementation of the DWT on the GPU mainly follows the wavelet lifting implementation using CUDA as proposed by van der Laan et al. [vdLJR11]. The lifting scheme gives rise to a reversible (i.e. lossless) integer-based DWT for the compression of each tile's height field, as described below. A floating-point DWT is used for lossy color compression to achieve improved compression rates.

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, we perform a push-pull error compensation (see Fig. 6). Therefore, at every node the difference is computed between the signal that is reconstructed from the parent node using the quantized coefficients and the signal that is computed by 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 using these coefficients and the corresponding detail coefficients. 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.

A different strategy to circumvent this problem is employed in JPEG2000, where coefficients at coarser levels
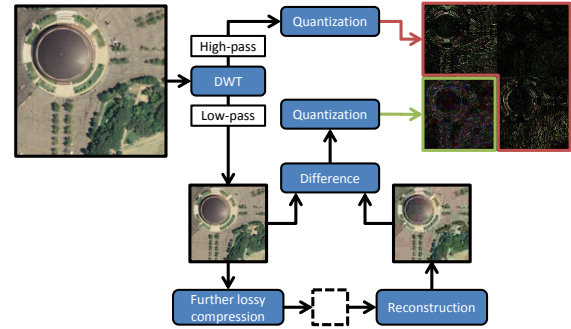
Figure 6: *Push-pull error compensation: To avoid propagating errors from coarser to finer levels, the difference between the original low-pass coefficients and their reconstruction is stored in addition to the detail coefficients.*

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. Our 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. Since typical color images contain many regions exhibiting only subtle color variations, this stream is very likely to contain many zeros or very small entries. Note that the same holds for typical terrain height fields, where over large regions only slight variations in height are present. We exploit this by using a run-length encoder followed by a Huffman encoder to further compress the data and, thus, to significantly reduce bandwidth limitations when streaming the data from disk to the GPU and vice versa. This coding backend allows for much higher throughput than JPEG2000's EBCOT at only slightly lower compression rates (see Sec. 5).

Scalar Field Compression: For scalar 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, we 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, too. 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.

4.1. Run-Length Coder

The entropy coder first performs a run-length encoding of the quantized wavelet coefficients (the *symbols* in the following discussion). 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. In our case, many runs of symbols equal to zero are expected, while runs of other symbols are rather unlikely. Therefore, we use 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*.

4.1.1. Encoder

The parallel CUDA implementation of run-length encoding makes use of data-parallel operations which can be realized efficiently on the GPU. Firstly, each symbol is flagged as either zero (marked by 0) or non-zero (marked by 1). An exclusive parallel prefix sum [SHGO11] 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. A graphical illustration of the implementation is shown in Fig. 7. Trailing zeros in the data do not need to be stored, since the total number of symbols is known to the decoder.
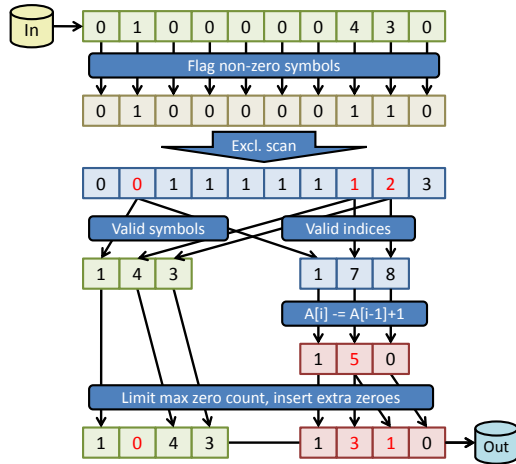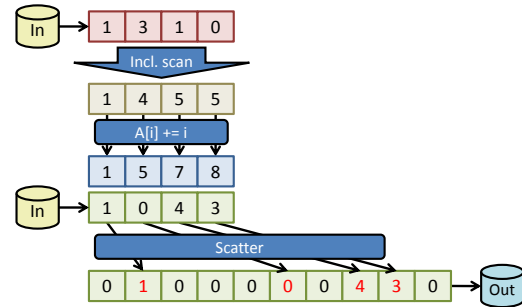

Figure 7: *Parallel implementation of run-length encoding. Input symbols (green arrays) are flagged and a scan operation generates output indices (blue arrays). The number of removed zeros is stored (red array), and zeros may be inserted into the symbol stream to limit the maximum zero count.*

To facilitate an efficient Huffman encoding (Sec. 4.2), the zero counts should not be too large since this may increase the length of the largest codeword significantly. To avoid this, additional zeros are inserted into the compacted symbol stream at locations where the zero count exceeds a given limit (255 in our implementation). 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 *o* for each entry. Finally, each symbol and associated zero count is re-positioned according to this offset in the symbol stream.

4.1.2. Decoder

The CUDA implementation of the run-length decoder first performs an inclusive prefix sum over the zero counts, and then adds to the resulting values the index of the respective element to determine the original indices of the non-zero symbols. Finally, the output array is cleared with zeros, and each symbol is written to its target position in a scattered write operation (see Fig. 8).

Figure 8: *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).*

Similar to the encoder, the run-length decoder uses only data-parallel operations and thus maps very well to the GPU. In particular, all memory operations except for the final scatter operation can be perfectly coalesced into contiguous memory transactions.

4.2. Huffman Coder

To further compress the run-length encoded symbol stream, we have realized a Huffman coder on the GPU. Huffman coding is an entropy coding technique which assigns a variable-length codeword to each input symbol, where the length of each codeword is determined by the symbol's probability of occurrence. The codewords adhere to a prefix code to allow unambiguous decoding. To map symbols to codewords, a lookup table is generated and stored along with the codeword stream. In our implementation, since the symbols and zero counts typically occur with very different distributions, we encode them separately to improve the compression rate.

4.2.1. 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.

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 might become very inefficient if write conflicts occur and 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, we store one histogram *per thread* in shared memory and combine these histograms in a second pass using a parallel reduce operation per bin [Pod07]. However, the number of histogram bins that can be processed per thread is strongly limited by the available shared memory. Thus, the occurrences are only computed for the first few bins for which we expect the most conflicts, and the execution then switches to a kernel that computes one histogram per warp [Pod07, SK07], i.e., a group of threads working in a SIMD fashion. Even though this can still lead to memory conflicts, our 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 first puts all symbols into a priority queue where their probabilities are used as keys, and then successively removes the two least probable symbols and inserts a new placeholder item with their combined probabilities. The number of times a symbol (or its placeholder) has been processed in this way corresponds to the length of the symbol's codeword. Since this algorithm is strongly sequential, it is realized on the CPU. This requires a round-trip to the CPU; however, since the Huffman table has to be stored to disk anyway, there is only little additional overhead. Table construction including 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.

To reduce the number of conflicts, we let each thread write k consecutive codewords. This also means that only every k-th element of the bit index array is needed, so we sum every k adjacent codeword lengths and then perform the prefix sum operation only on the smaller set of elements. For optimal memory bandwidth use, we perform the compaction step in shared memory and afterwards write the compacted array to global memory using coalesced memory transactions. The proposed data-parallel implementation is illustrated in Fig. 9. It is similar to the one proposed by Balevic [Bal09], but uses the data-dependent Huffman table constructed in the previous step.
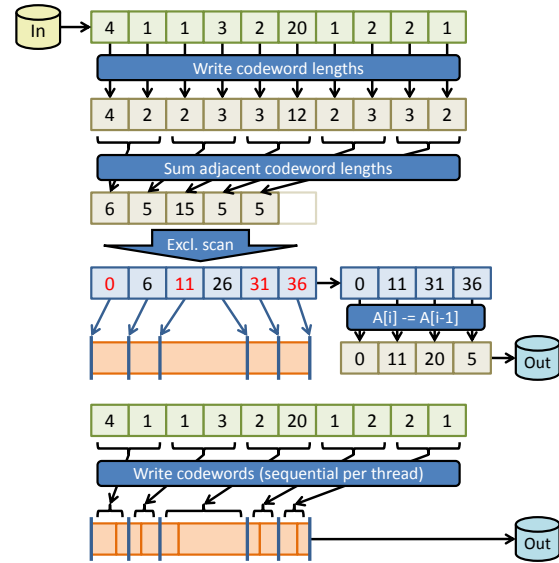
Figure 9: *Parallel implementation of Huffman encoding. Lengths of codewords corresponding to input symbols (green array) are written into an auxiliary buffer. Every k adjacent values (k = 2 in the example) are summed up. A scan operation computes the output bit index (blue array) for every k-th codeword. 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.*

Since the decoder needs one of the codeword bit indices to start each decoder thread, we store every m-th element of the index array, i.e. the bit index of every $k \cdot m =: n$-th codeword as side information. In our current implementation, $k = 8$ and $m = 16$, so every 128-th index is stored. To save memory, we do not store the absolute values of the indices, but the increments from one index to the next. 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].

4.2.2. 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 is performed first to compute the bit index of every n-th codeword. For each such index, one thread is started and decodes n symbols sequentially. The implementation of this step is similar to the one proposed by van Waveren [vW06], but we construct the lookup table to handle short symbols on the GPU.

To allow coalesced memory write operations, 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. Each block of $t \times n$ interleaved values is then read into shared memory, where it is reordered and written back to global memory. Reordering allows accessing the data values in a coalesced way, resulting in an improvement of the decoder throughput by about 40%. Fig. 10 illustrates the parallel implementation of the Huffmann decoder.
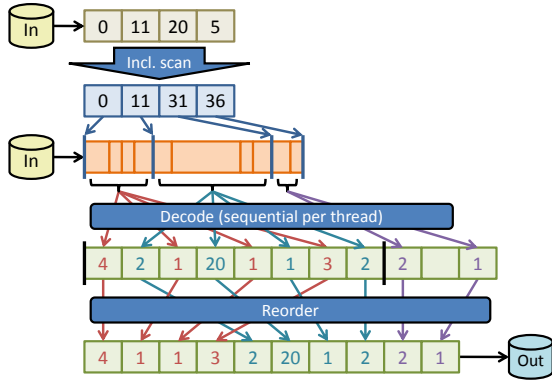
Figure 10: *Parallel implementation of Huffman decoding. A scan operation over relative bit indices gives the bit indices (blue array) of every n-th codeword (n = 4 in the example). n symbols per thread are sequentially decoded (green array) from their codewords (orange array). To achieve coalesced memory accesses, t consecutive threads (t = 2 in the example) write their output in an interleaved order.*

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 never be coalesced. To avoid frequent accesses to the input data, 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.

5. Results

To demonstrate the efficiency of our terrain editing approach we have used a textured terrain height field of Vorarlberg,

Austria. The orthophoto has a size of 447000×677000 pixels (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 presented in this work 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.

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 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.

5.2. Compression Rate and Quality

To assess the compression rate and reconstruction quality of the GPU coder, we have performed a number of tests using the Kodak test image suite, the "New Test Images" suite (RGB 8 bit) [Raw11], and a set of 100 sub-images of the Vorarlberg orthophoto, each of 2048×2048 pixels. On each image a four-level DWT was performed, and the resulting coefficients were compressed as described.

We have compressed the same images using JPEG, JPEG2000, and the S3TC DXT1 format. For JPEG and JPEG2000 compression, we used the ImageMagick library v. 6.7.0 [Ima11]. The S3TC compression was performed via the Squish library v. 1.10 [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.
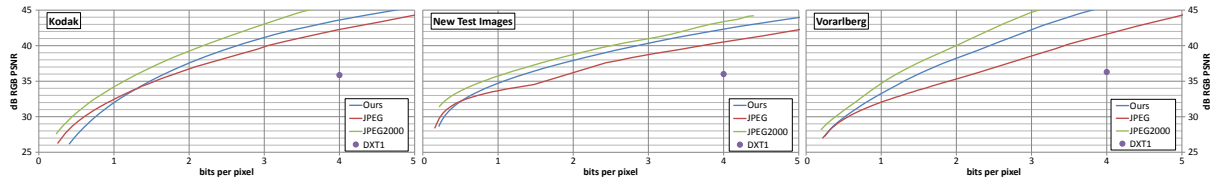
Figure 11: *Graphs of PSNR vs. bpp for the test image suites "Kodak", "New Test Images" and "Vorarlberg".*

Fig. 11 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. Our method usually outperforms JPEG, often significantly, except for bit rates below 1.3 bpp for "Kodak" and below 0.5 bpp for "New Test Images". 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, our method 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 our approach.

5.3. Compression Throughput

To produce realistic and robust performance numbers, we 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 19.0 min, including the construction of the multi-resolution pyramid. This corresponds to a throughput of 270 MPix/s. Decoding took 6.9 min, giving a throughput of 730 MPix/s.
Encoding the entire 4.9 gigasample height field using 8 DWT levels took 7.0 s at a 700 MPix/s throughput. Decoding took 2.6 s at a 1780 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 jpgtest program from the libjpeg-turbo library v. 1.1.1 [lib10] at quality 90 and 4:4:4 chroma sampling achieves 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 comes close to our approach, 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 our approach, these operations are always included.

The Kakadu library [Kak11], one of the fastest JPEG2000 implementations, reports a throughput of 25 and 35 MPix/s, respectively, for encoding and decoding, on a 2.4 GHz Core 2 Duo CPU. On a quad core architecture similar to ours, it can be expected that a considerable increase in throughput can be achieved. 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 on our GeForce GTX 580, excluding data transfer between CPU and GPU. Encoding of RGB pixel data to DXT1 using the NVTT GPU compressor [NVI10] achieves 21 MPix/s.

6. Conclusion and Future Work

We 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 a custom scheme that provides both encoding and decoding at much higher rates than disk transfer and achieves compression rates that compare favorably to JPEG and JPEG2000 compression.

Since currently our work serves as a proof-of-concept tool demonstrating the possibility to interactively edit gigasample terrain fields, in the future we will focus more on the specific editing operations that are required in real-world applications. In addition, we will analyze the potential of the GPU decoder for (parallel) gigavoxel volume rendering, and we will investigate the possibility to perform visually guided segmentation or annotation on large pixel and voxel data.

Acknowledgments

References

[AG06] ATLAN S., GARLAND M.: Interactive multiresolution editing and display of large terrains. *Computer Graphics Forum 25*, 2 (2006), 211–223. 2

[AGD10] AMMANN L., GÉNEVAUX O., DISCHLER J.-M.: Hybrid rendering of dynamic heightfields using ray-casting and mesh rasterization. In *Proc. Graphics Interface* (2010), pp. 161–168. 2

[AMM00] ABU-MOSTAFA Y. S., MCELIECE R. J.: Maximal codeword lengths in huffman codes. *Computers & Mathematics with Applications 39*, 11 (2000), 129 – 134. 7

[Bal09] BALEVIC A.: Parallel variable-length encoding on GPG-PUs. In *Proc. Parallel and Distributed Computing (Euro-Par)* (2009), pp. 26–35. 3, 7

[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. 2

[BGP09] BÖSCH J., GOSWAMI P., PAJAROLA R.: RASTeR: Simple and efficient terrain rendering on the GPU. In *Proc. Eurographics 2009 - Areas Papers* (2009), pp. 35–42. 2

[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. 2

[BN08] BRUNETON E., NEYRET F.: Real-time rendering and editing of vector-based terrains. *Computer Graphics Forum 27*, 2 (2008), 311–320. 3

[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. 3

[Bro08] BROWN S.: Squish library, version 1.10, 2008. http://code.google.com/p/libsquish/. 8

[CDF92] COHEN A., DAUBECHIES I., FEAUVEAU J. C.: Biorthogonal bases of compactly supported wavelets. *Comm. on Pure and Applied Mathematics 45*, 5 (1992), 485–560. 5

[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. 4

[DKW09] DICK C., KRÜGER J., WESTERMANN R.: GPU raycasting for scalable terrain rendering. In *Proc. Eurographics 2009 - Areas Papers* (2009), pp. 43–50. 4

[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. 2, 3

[FWH*09] FÜRST N., WEISS A., HEIDE M., PAPANDREOU S., BALEVIC A.: CUJ2K library, version 1.1, 2009. http://cuj2k.sourceforge.net. 9

[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. 2

[HCP02] HE Y., CREMER J. F., PAPELIS Y. E.: Real-time extendible-resolution display of on-line dynamic terrain. In *Proc. Graphics Interface* (2002), pp. 151–160. 2

[Ima11] IMAGEMAGICK STUDIO LLC: ImageMagick library, version 6.7.0, 2011. http://www.imagemagick.org. 8

[Kak11] KAKADU SOFTWARE: Kakadu SDK v6 with Speed Pack, 2011. http://www.kakadusoftware.com. 9

[LC10] LINDSTROM P., COHEN J. D.: On-the-fly decompression and rendering of multiresolution terrain. In *Proc. Interactive 3D Graphics and Games (I3D)* (2010), pp. 65–73. 2

[lib10] LIBJPEG-TURBO PROJECT: libjpeg-turbo library, version 1.1.1, 2010. http://www.libjpeg-turbo.org. 9

[LK10] LAMBERS M., KOLB A.: Dynamic terrain rendering. *3D Research 1*, 4 (2010), 1–8. 2

[MS03] 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, 2003. 5

[NVI10] NVIDIA CORP.: NVIDIA Texture Tools, version 2.08, 2010. http://developer.nvidia.com/gpu-accelerated-texture-compression. 9

[PG07] PAJAROLA R., GOBBETTI E.: Survey of semi-regular multiresolution models for interactive terrain rendering. *Visual Computer 23*, 8 (2007), 583–605. 1, 2

[Pod07] PODLOZHNYUK V.: *Histogram calculation in CUDA*. NVIDIA Corp., 2007. 7

[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. 3

[PV95] PERLIN K., VELHO L.: Live paint: Painting with procedural multiscale textures. In *Proc. Computer Graphics (SIGGRAPH)* (1995), pp. 153–160. 3

[Raw11] RAWZOR: The new test images, 2011. http://www.imagecompression.info/test_images/. 8

[SHGO11] SENGUPTA S., HARRIS M., GARLAND M., OWENS J. D.: Efficient parallel scan algorithms for many-core GPUs. In *Scientific Computing with Multicore and Accelerators*. Taylor & Francis, 2011, ch. 19, pp. 413–442. 6

[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. 7

[TM01] TAUBMAN D. S., MARCELLIN M. W.: *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, 2001. 3

[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 and Distributed Systems 19*, 3 (2008), 299 –310. 3

[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 and Distributed Systems 22*, 1 (2011), 132–146. 2, 3, 5

[vW06] VAN WAVEREN J. M. P.: *Real-Time Texture Streaming & Decompression*. Tech. rep., id Software, Inc., 2006. 8

[vWC07] VAN WAVEREN J. M. P., CASTAÑO I.: *Real-Time YCoCg-DXT Compression*. Tech. rep., id Software, Inc. and NVIDIA Corp., 2007. 2, 3

[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. 3

[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. 3