

# Data-Parallel Decompression of Triangle Mesh Topology

Quirin Meyer<sup>1</sup>, Benjamin Keinert<sup>1</sup>, Gerd Sußner<sup>2</sup>, Marc Stamminger<sup>1</sup>

<sup>1</sup>Computer Graphics Group, University Erlangen-Nuremberg, <sup>2</sup>RTT AG

---

## Abstract

*We propose a lossless, single-rate triangle mesh topology codec tailored for fast data-parallel GPU decompression. Our compression scheme coherently orders generalized triangle strips in memory. To unpack generalized triangle strips efficiently, we propose a novel parallel and scalable algorithm. We order vertices coherently to further improve our compression scheme. We use a variable bit-length code for additional compression benefits, for which we propose a scalable data-parallel decompression algorithm. For a set of standard benchmark models, we obtain (min: 3.7, med: 4.6, max: 7.6) bits per triangle. Our CUDA decompression requires only about 15 % of the time it takes to render the model even with a simple shader.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and techniques—Graphics data structures and data types I.3.1 [Computer Graphics]: Hardware architecture—Parallel processing

---

## 1. Introduction

Compression methods have proven to be an effective way to respond to the demand for even more detailed triangle meshes emanating from medical data, numerical simulations, and computer-aided design models. In order to quickly access triangles, efficient decompression is necessary. Yet, the better part of existing decompression algorithms relies on sequential techniques. These are unlikely to scale well, as hardware architecture shifts towards many-core systems. This trend helps in continuously increasing computing power, but at the same time, data access power cannot keep pace. Thus, compact representations remain an effective means to handle bandwidth limitations. Many-core chips increasingly find their way into mobile devices, which additionally benefit from data-parallel decompression, as they chronically suffer from memory space limitations.

While there is a whole body of sequential methods well suited for decompressing triangle meshes on single-core systems, far too little attention has been paid in efficient data-parallel decompression on many-core architectures. To unpack a triangle, most existing algorithms rely on previously unpacked triangles. As a consequence, they are inherently sequential. Though they achieve compression rates of 1 bit per triangle (bpt), they contain recursive dependencies. These dependencies are either hard to resolve or require far

too many synchronization points, which impedes efficient parallel implementations.

In this paper, we propose a codec that significantly reduces the memory needed for triangle connectivity. Vertex attributes, such as positions, normals, or colors, can be efficiently compressed and decompressed orthogonally to our approach [PBCK05, MSS\*10, MSGS11]. As our algorithm decompresses all triangles at once and not progressively, it belongs to the class of single-rate compression methods. Unlike previous approaches, our algorithm is fully parallel and therefore enables real-time decompression. Thus, it saves device memory and more triangle meshes fit in the same amount of GPU memory than non-compressed triangle meshes. This may additionally reduce the need for CPU-GPU data transfers. Sequential methods reach high compression ratios with data structures, such as stacks, queues, or trees. However, the use of these data structures in a data-parallel algorithm would harm performance significantly. Thus, we cannot simply port a sequential algorithm to a parallel system and expect close-to-optimal compression ratios at high-performance decompression times. Instead, we design a novel codec that prefers higher decompression speed at the cost of lower compression ratios. Yet, our algorithm reaches (min: 3.7, med: 4.6, max: 7.6) bpt for a series of test models. For the Welsh Dragon of Figure 1a with 2.2 M triangles, we obtain a compression ratio of 26:1 for the topology. While rendering the uncompressed Welsh Dragon with sim-

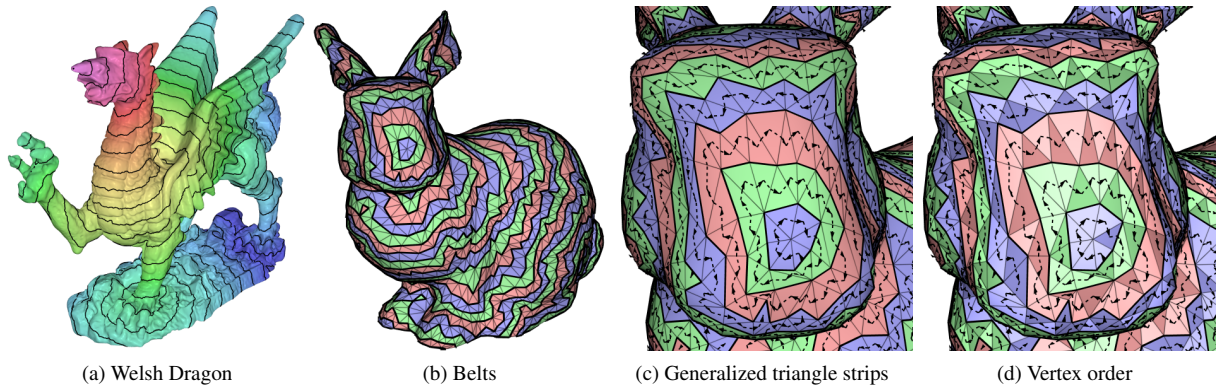


Figure 1: For compression, we reorder triangles and vertices coherently. (a, b) We traverse the model breadth-first and create a series of belts, shown in different colors. (c) Then, we create generalized triangle strips. (d) Vertices are ordered as guided by the strips. Triangles visiting a vertex for the first time are in light shades. Dark shaded triangles revisit a vertex.

ple Blinn-Phong shading takes 6.5 ms, rendering the same model including our decompression takes 8.1 ms, which is an overhead of only 1.6 ms. For more complex shading, the overhead amortizes quickly. We achieve these results by the following contributions:

- We traverse the mesh breadth-first. Triangles visited in the same breadth-level form a *belt*, indicated through different colors in Figure 1b. The triangles are ordered coherently in *generalized triangle strips* (Figure 1c).
- We present a novel data-parallel method for decoding generalized triangle strips efficiently. Its thread allocation scales linearly with the number of triangles.
- We coherently order vertices as guided by the triangle order: If a triangle references a vertex firstly (bright triangles in Figure 1d), one bit is enough for the vertex information. We present an efficient data-parallel algorithm for reconstituting these vertices.
- For triangles that refer to an already referenced vertex (dark triangles in Figure 1d), we compress its vertices using a variable bit-length code. For decompression, we present a fast data-parallel algorithm that scales well.

Compressing a model is fast enough to do it at load time. For example, the Welsh Dragon requires 2.3 seconds for compression. We alter triangle and vertex order (but not orientation) which is common practice for most compression methods. As a by-product, our coherent triangle order is also very beneficial for GPU rendering. Frame-times turned out to be very similar to those achieved by cache-oblivious orders.

## 2. Related Work

As the field of geometry compression is vast, we focus on single-rate, lossless connectivity coding of triangle meshes, and refer to overview reports [GGK02, AG03, PKJK05]. To reach the entropy for planar triangle graphs of

1.62 bpt [Tut62], connectivity coding methods conquer the mesh and thereby encode the mesh elements. Depending on the element type that guides the conquest, we distinguish between face-based [GS98, Ros99], edge-based [Ise00], and vertex-based [TG98, KPRW05] approaches. These methods achieve about 2 bpt and paired with arithmetic or Huffman coding obtain about 1 bpt. However, they unpack element after element and are therefore sequential. To our knowledge, no data-parallel implementation has been described yet.

As meshes grow faster than main memory, several formats have been proposed that allow out-of-core access [HLK01, IG03]. Meshes are divided into clusters that are compressed using existing sequential algorithms. Typically, each cluster is about  $10^3$  vertices, and each triangle uses 1 – 2 bpt. This immediately leads to a parallelization approach that unpacks each cluster on one core independently. However, every cluster comes with a certain memory overhead. Thus, the more cores we have to keep busy, the more clusters we require, and the worse the compression ratio becomes. In the limit case of one triangle per cluster, clustering approaches yield no compression at all. GPUs already have hundreds of cores and will have even more in the future; hence these approaches do not scale well. Moreover, as the underlying decompression algorithms are quite involved, parallel threads are unlikely to run in lockstep, which significantly degrades performance on single-instruction-multiple-data (SIMD) architectures.

So called random access mesh compression techniques divide the mesh into clusters. To access a vertex, the cluster it is located in has to be determined with an indexing structure. Then, the cluster is decompressed entirely. The methods are designed for interactive out-of-core applications and reach about 3 – 8 bpt. However, they all reuse sequential decompression methods [YL07, CKLL09, CH09].

Only recently, compact data structures were proposed [GLLR11a, GLLR11b] that allow true random access

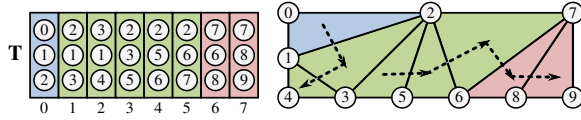


Figure 2: The array  $\mathbf{T}$  stores triangles in the explicit representation. Triangles of the same color belong to the same belt. The initial belt is shown in blue. Black arrows show the traversal order of the generalized triangle strip. The numbers below the elements of  $\mathbf{T}$  are the corresponding array indices.

and thus do not rely on sequential methods. They are primarily designed to access neighborhood information in order to enable efficient mesh operations. Thus, connectivity is compressed only moderately to about 26 bpt.

For fast decompression, special hardware implementations have been proposed [Dee95, Cho97, CK07] that achieve around 40, 20, and 8 bpt, respectively. However, all of them use FIFO caches which prevent fast and scalable data-parallel replications. Moreover, we are not aware of current hardware exposing these compression techniques. Variable bit-length codes were decompressed first on a GPU by Lindstrom and Cohen [LC10] in the context of terrain rendering: They group height maps in patches and then decompress the values sequentially in a geometry shader. Their approach requires carefully tweaking the chunk size to trade between compression ratio and decompression speed. Dick and co-workers [DSW09] unpack generalized triangle strips in a geometry shader for terrain rendering. They spawn one thread for each strip, and each strip consists of 16 triangles at most. However, for general meshes, efficiently finding strips of a fixed length is hard and time-consuming.

### 3. Creating Generalized Triangle Strips

Triangle topology is often stored in an *explicit representation*: triangles are arranged in an array  $\mathbf{T}$ , whose entries are triples of *vertices*, as shown in Figure 2.  $\mathbf{T}[i]$  refers to the  $i$ th triangle, and  $\mathbf{T}[i].v_0$ ,  $\mathbf{T}[i].v_1$ ,  $\mathbf{T}[i].v_2$  are its vertices. With the term *vertex*, we exclusively mean the topological entity: a vertex is an index that points into an array of positions or other attributes like normals, colors, or texture coordinates.

Similar to other compression schemes, ours exploits data coherency to reduce memory consumption. Therefore, we order triangles such that  $\mathbf{T}[i]$  and  $\mathbf{T}[i+1]$  share an edge. A *generalized triangle strip* is a sequence of triangles in which every triangle shares an edge with its preceding triangle. Of course, it is not always possible that all neighbors in  $\mathbf{T}$  share an edge. But the more neighboring entries share an edge the better the compression ratio becomes. Additionally, it is crucial for high compression ratios that strips are ordered coherently in memory. This will become apparent in Section 6.

Our algorithm for ordering triangles and strips consists of two steps: In step one – named *creating belts* – we conquer

the triangles of a mesh breadth-first and create an ordered sequence of *belts*. A belt is an unordered set of triangles visited in one breadth-first step. For example, in Figure 1b, the connected triangles of common color belong to the same belt. In the second step – called *stripification* – we order triangles of each belt into generalized triangle strips.

**CREATING BELTS:** To simplify explanations, we use the term *triangle one-ring of a vertex*, which is the set of triangles incident to that vertex. As initial belt, we pick the triangle one-ring of a *seed vertex*. In the example of Figure 2, the initial belt is shown in blue and originates from the seed vertex 0. We observe that the seed vertex has only minor influence on the compression ratio, so we choose it randomly. For each connected component of a mesh, we need one seed vertex. We iteratively add new belts by considering the union of all triangle one-rings of vertices of the previous belt. All triangles from that union which are not part of an existing belt are added to the next belt. In Figure 2, the belt following the blue belt contains all the green triangles. We continue adding belts until the last triangle is added to the final belt, shown in red in Figure 2. In Figure 1b, the connected triangles of the same color are in a common belt.

**STRIPIFICATION:** The belts allow to easily order the triangles into generalized triangle strips, shown with dashed arrows in Figure 2. We start with the first belt and add one of its triangles to the array of triangles  $\mathbf{T}$ . Then, we consider all its neighboring triangles in the same belt which are not yet in  $\mathbf{T}$ . We add one of them to  $\mathbf{T}$  and put all others onto a stack. In the same way, we iteratively consider the neighbors of the recently added triangle. If the current triangle has no neighbors left within the current belt, we start a new strip using a triangle from the stack. Once the stack is empty, we look for a triangle in the current belt that is not yet in  $\mathbf{T}$ . If all triangles of the belt are in  $\mathbf{T}$ , we proceed to the neighboring belt. To create strips across belt boundaries, we try to merge strips from neighboring belts by choosing the first triangle of a new belt such that it neighbors the most recently added triangle. Figures 1c and 2 show the result of our stripification.

The triangles in the array  $\mathbf{T}$  of Figure 2 are in generalized strip order. We use this ordering for compression in Section 4. At the same time, the strips are arranged according to the order guided by the breadth-first traversal. That means, strips which are close on the mesh are also close in  $\mathbf{T}$ . This is done intentionally, since it improves compression ratio, as shown in Section 6.

Further, we coherently order vertices. We index the vertices in the order they are visited when added to  $\mathbf{T}$ . If a vertex is visited multiple times, it keeps the index it was assigned first, as seen in Figure 2. By this, we order the vertices coherently which results in additional compression benefits, as outlined in Section 5.

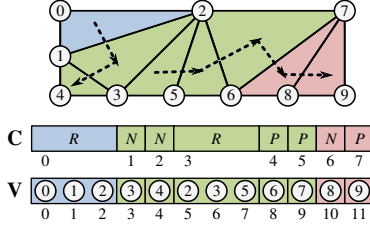
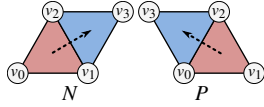


Figure 3: The generalized triangle strip of Figure 2 stored compactly in a strip-code array **C** and a vertex array **V**. The numbers below the elements are their array indices.

#### 4. Decompression of Generalized Triangle Strips

Obviously, generalized triangle strips can be stored compactly: If triangles  $T[i]$  and  $T[i+1]$  share an edge, we do not have to store three vertices explicitly to specify  $T[i+1]$ : it is sufficient to store which edge of the previous triangle  $T[i]$  is shared, and the vertex of  $T[i+1]$  that is not used in the previous triangle.



We can always consistently order the vertices of the triangles such that only the adjacency cases shown in the inset figure occur for oriented manifold meshes. The current triangle  $(v_0, v_1, v_2)$  is marked red. Its descending triangle, shown in blue, is either adjacent to the edge  $(v_1, v_2)$  or  $(v_2, v_0)$ . To distinguish the two cases, we introduce the *strip-codes*  $N$  and  $P$ , respectively.  $N$  is short for "next-edge", since  $(v_1, v_2)$  is the edge after  $(v_0, v_1)$  with respect to the current triangle. Likewise,  $P$  abbreviates "previous-edge". If  $T[i]$  and  $T[i+1]$  do not share an edge, we introduce the strip-code  $R$ , as for "restart". In that case, all three vertices have to be provided explicitly to specify the triangle. The restart-code enables us to handle non-manifold meshes as well. As depicted in Figure 3, it is sufficient to store one strip-code per triangle in an array **C** and – depending on the strip-code – one or three vertices per triangle in the vertex array **V**. This allows saving up to two third of the vertex information over an explicit representation. As we have three strip-code states –  $N$ ,  $P$ , and  $R$  – each triangle requires two bits for its strip-code.

However, current graphics hardware only supports *simple triangle strips*. They are a special case of generalized triangle strips: A strip-restart is encoded with a magic number in the vertex array **V**, and the array of strip-codes is given implicitly as  $C = (R, N, P, N, P, \dots)$ . If we used simple triangle strips, we could not choose between the two neighbors of a triangle and would be forced to take the one predefined by the implicit strip-codes. We could mimic the behavior with adding restart-codes or degenerate triangles, but compression ratios would suffer.

Thus, we have to account for the lacking hardware support of generalized triangle strips by converting them into the explicit representation **T** using CUDA. We only need the

arrays **C** and **V** on graphics hardware, that generally require less space than **T**. Yet, the conversion is a lot more involved than it is for simple triangle strips. To do so, we make use of *scan-operations* [Ble90]. We define an *exclusive scan* as

$$\bar{A}[i] := \begin{cases} 0, & i = 0 \\ op(\bar{A}[i-1], A[i-1]), & i > 0, \end{cases} \quad (1)$$

and an *inclusive scan* as

$$\bar{A}[i] := \begin{cases} A[0], & i = 0 \\ op(\bar{A}[i-1], A[i]), & i > 0, \end{cases} \quad (2)$$

where **A** is the input array,  $\bar{A}$  is the scanned output array, and  $op$  is a binary associative operation. Here, we use the maximum and the sum of two numbers. On many-core architectures, a scan operation is parallelized efficiently. Algorithms that make use of scans scale well, as their thread count is  $\mathcal{O}(|A|)$ , where  $|A|$  is the number of elements in **A**. Our algorithms make use of an optimized scan for CUDA [HOS\*11].

##### 4.1. Data-Parallel Algorithm

At first sight, converting a generalized triangle strip to the explicit representation appears to be a sequential process: A triangle depends on its preceding triangle, which in turn depends on its predecessor, and so forth. In this section, we show how to express this recursiveness in terms of Equations (1) and (2). Thereby, the challenge is to find appropriate associative operations. Only this allows a parallel implementation of our algorithm using scans. The outline of the final algorithm is as follows: We use parallel scans to create two helper arrays  $\bar{M}$  and  $\bar{Q}$  from the strip-code array **C**, as shown in Figures 4a and 4b. We use the helper arrays to reconstruct the explicit representation **T** in Figure 4c.

Next, we derive how to create the helper arrays  $\bar{M}$  and  $\bar{Q}$ . To do so, we formalize the conversion from generalized triangle strips to the explicit representation. For simplicity, we neglect restart-codes  $R$ , first. We express the relation between current and previous triangle recursively:

$$T[i] = \begin{cases} (T[i-1].v_0, T[i-1].v_2, V[i+2]), & C[i] = P \\ (T[i-1].v_2, T[i-1].v_1, V[i+2]), & C[i] = N. \end{cases} \quad (3)$$

The parts depending on the previous triangles are colored red. By the end of this section, all red parts will have disappeared, allowing to unpack each triangle independently and thus in parallel with one thread per triangle.

We now rewrite Equation (3) to support restart codes. Without restart-codes,  $i+2$  always points to the third vertex of the  $i$ th triangle. In the presence of restart-codes, that is no longer the case, as each restart shifts the third vertex three more vertices back in **V**. In the other cases, i.e., the current triangle uses a  $P$ - or an  $N$ -code, the third vertex of the  $i$ th triangle is just one element further in **V**. We use these two properties to create a helper array  $\bar{M}$  that points to the third vertex of each triangle:

$$\bar{M}[i] = \begin{cases} \bar{M}[i-1] + 1, & C[i] \neq R \\ \bar{M}[i-1] + 3, & C[i] = R. \end{cases} \quad (4)$$

$$\bar{M}[i] = \begin{cases} \bar{M}[i-1] + 1, & C[i] \neq R \\ \bar{M}[i-1] + 3, & C[i] = R. \end{cases} \quad (5)$$



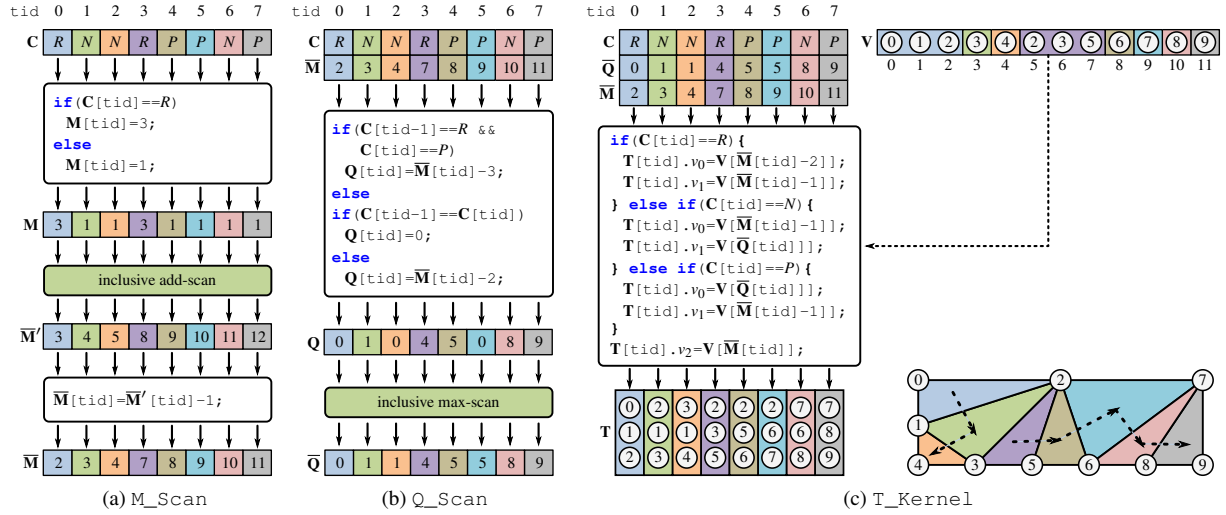


Figure 4: The mesh in (c) is converted from a generalized triangle strip stored compactly in  $C$  and  $V$  into the explicit representation  $T$ . The solid arrows indicate the data-parallel flow. Threads and array indices are enumerated consecutively by a thread identification number  $tid$ . The different colors in the arrays match the colors of the triangles. The code in the white boxes is executed for each triangle independently. The green boxes indicate data-parallel scans.

As the first triangle's strip-code is always  $R$ , we initialize  $\bar{M}[0] = 2$ . The recursion of Equations (4) and (5) match the definition of the inclusive scan using the plus-operation of Equation (2), which can be computed efficiently on GPUs. Figure 4a shows a flow chart including code for computing  $\bar{M}$ . The transformations before and after the scan require no separate kernel, as they can be fused into one kernel. Hence, the intermediate arrays  $M$  and  $\bar{M}'$  are of course not created explicitly and are depicted only for explanation purposes. The figure further demonstrates the computation of  $\bar{M}$  by continuing the example of the mesh in Figure 2 and 3.

With  $\bar{M}$ , we enrich Equation (3) to support restart-codes:

$$T[i] = \begin{cases} (T[\bar{M}[i-1]].v_0, T[\bar{M}[i-1]].v_2, V[\bar{M}[i]]), & C[i] = P \\ (T[\bar{M}[i-1]].v_2, T[\bar{M}[i-1]].v_1, V[\bar{M}[i]]), & C[i] = N \\ (V[\bar{M}[i]-2], V[\bar{M}[i]-1], V[\bar{M}[i]]), & C[i] = R. \end{cases}$$

While the  $R$ -case does not have any recursive dependency, the  $P$ - and  $N$ -cases require vertices of the previous triangle. Fortunately, both contain the third vertex of the previous triangle  $T[\bar{M}[i-1]].v_2$ . But we know that the third vertex of triangle  $i-1$  is  $V[\bar{M}[i-1]]$ . Using Equation (4), we simplify this further to  $V[\bar{M}[i]-1]$ :

$$T[i] = \begin{cases} (T[\bar{M}[i-1]].v_0, V[\bar{M}[i]-1], V[\bar{M}[i]]), & C[i] = P \\ (V[\bar{M}[i]-1], T[\bar{M}[i-1]].v_1, V[\bar{M}[i]]), & C[i] = N \\ (V[\bar{M}[i]-2], V[\bar{M}[i]-1], V[\bar{M}[i]]), & C[i] = R. \end{cases}$$

Now, only  $T[\bar{M}[i-1]].v_0$  and  $T[\bar{M}[i-1]].v_1$  depend on the previous triangle. To get rid of that recursion, we introduce the array  $Q$  whose entries point to those vertices in  $V$ , that

are already used in a triangle prior to the previous one:

$$T[i] = \begin{cases} (V[\bar{Q}[i]], V[\bar{M}[i]-1], V[\bar{M}[i]]), & C[i] = P \\ (V[\bar{M}[i]-1], V[\bar{Q}[i]], V[\bar{M}[i]]), & C[i] = N \\ (V[\bar{M}[i]-2], V[\bar{M}[i]-1], V[\bar{M}[i]]), & C[i] = R. \end{cases} \quad (6)$$

Next, we have to compute  $\bar{Q}$ , where it is not enough to consider each triangle and its strip-code  $C[i]$  isolated. Instead, we need to compare the indices into  $V$  as given by Equation (6) of two successive triangles  $T[i-1]$  and  $T[i]$ :

	$C[i] = P$	$C[i] = N$
$C[i-1] = P$	$\bar{Q}[i] = \bar{Q}[i-1]$	$\bar{Q}[i] = \bar{M}[i-1] - 1$
$C[i-1] = N$	$\bar{Q}[i] = \bar{M}[i-1] - 1$	$\bar{Q}[i] = \bar{Q}[i-1]$
$C[i-1] = R$	$\bar{Q}[i] = \bar{M}[i-1] - 2$	$\bar{Q}[i] = \bar{M}[i-1] - 1$

Using these identities and  $\bar{M}[i-1] = \bar{M}[i] - 1$  (cf. Equation (4), as  $C[i] \neq R$ ), we find that

$$\bar{Q}[i] = \begin{cases} \bar{M}[i] - 3, & C[i-1] = R \text{ and } C[i] = P \\ \bar{Q}[i-1], & C[i-1] = C[i] \\ \bar{M}[i] - 2, & \text{otherwise.} \end{cases} \quad (7)$$

Note that  $\bar{Q}[i] = \bar{Q}[i-1]$  is the only recursive branch. We need to find a way to propagate previous entries across  $\bar{Q}$ : From its definition, we see that  $\bar{Q}$  is monotonic increasing. That means,  $\bar{Q}[i] \geq \bar{Q}[i-1]$  or  $\bar{Q}[i] = \max(\bar{Q}[i], \bar{Q}[i-1])$ . Thus, we compute the array  $Q$  first, i.e.,

$$Q[i] = \begin{cases} \bar{M}[i] - 3, & C[i-1] = R \text{ and } C[i] = P \\ 0, & C[i-1] = C[i] \\ \bar{M}[i] - 2, & \text{otherwise.} \end{cases} \quad (8)$$

Thereafter, we apply an inclusive max-scan to  $Q$ , which

yields  $\bar{\mathbf{Q}}$ , as shown in Figure 4b. The operations prior to the scan are fused into one kernel called  $\mathbf{Q\_Scan}$ , so no explicit memory has to be reserved for  $\mathbf{Q}$ .

With the temporary arrays  $\bar{\mathbf{M}}$  and  $\bar{\mathbf{Q}}$ , Figure 4c shows an example and the code for  $\mathbf{T\_Kernel}$  that computes the explicit representation  $\mathbf{T}$  using one thread per triangle. Also both  $\mathbf{M\_Scan}$  and  $\mathbf{Q\_Scan}$  scale with the number of triangles. Existing parallelization approaches [DSW09] subdivide the strips into chunks with equal number of triangles. Each chunk is assigned to one thread. However, they have to artificially insert restart codes which lowers compression efficiency. The more triangles that are in one chunk the better the compression, but at the same time the worse the degree of parallelism. It is hard to find a good trade-off, and, once found, it might only be suitable for a specific architecture. In contrast, our approach does not have this drawback, as each triangle is assigned to a different thread.

#### 4.2. Restart Emulation

Each strip-code  $\mathbf{C}[i]$  needs two bits to encode one of the three states  $R$ ,  $P$ , and  $N$ . We can save one bit on each code when we remove the  $R$ -state and simulate it by inserting four degenerate triangles. However, this increases the amount of triangles. In some cases, restart emulation has a positive effect on compression ratio, while for a mesh with many restart codes it degrades compression ratio (see Section 7.1).

In any case, restart emulation simplifies conversion to the explicit representation: We remove all  $R$ -branches from the kernels in Figure 4. Moreover,  $\bar{\mathbf{M}}[i] = i + 2$  always holds. This saves bandwidth, as no kernel needs to access  $\bar{\mathbf{M}}$ . Moreover, we omit  $\mathbf{M\_Scan}$  in Figure 4a. However, after decompression, degenerate triangles populate  $\mathbf{T}$ . This does not impose a major problem, as we feed the triangles into the GPU pipeline which removes degenerate triangles automatically.

#### 5. Incremental Vertices

With generalized triangle strips, it is possible to achieve a compression ratio of at most 3:1, as every triangle requires at least one vertex to be specified. We will now improve on this by compressing the vertex array  $\mathbf{V}$ . As explained in Section 3, vertices are numbered in the order they are visited when generating triangle strips. Thus, many neighboring entries of  $\mathbf{V}$  are consecutive (see, for example,  $\mathbf{V}$  of Figure 5). We use incremental vertices for further memory savings: we store one bit for every vertex  $k$  in an array  $\mathbf{INC}$ , which indicates if the vertex is incremented with respect to the previously incremented vertex ( $\mathbf{INC}[j] = 1$ ) or revisited ( $\mathbf{INC}[j] = 0$ ). Whenever a triangle needs to revisit a vertex, the vertex is stored in  $\mathbf{U}$ . In Figure 1d, all triangles in bright shades have incremental vertices.

Figure 5 demonstrates how to compute  $\mathbf{V}$  from  $\mathbf{U}$  and  $\mathbf{INC}$  efficiently in parallel. An exclusive add-scan over  $\mathbf{INC}$  directly yields the entries of  $\mathbf{V}$  in case the vertex is a simple

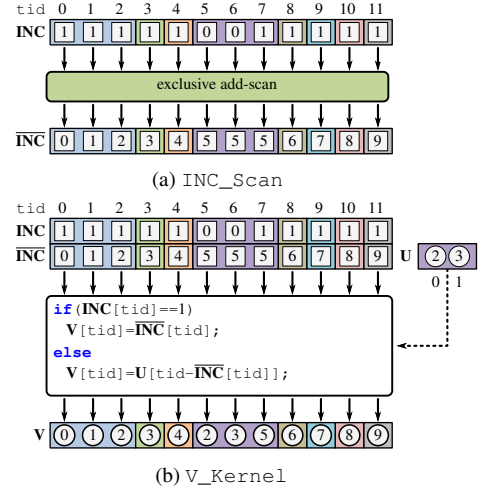


Figure 5: From the arrays  $\mathbf{INC}$  and  $\mathbf{U}$ , we compute the vertex array  $\mathbf{V}$  using a data-parallel scan (a) and a kernel (b).

increment. In the case of a vertex reuse, i.e.,  $\mathbf{INC}[j] = 0$ , we have to properly index into  $\mathbf{U}$  as shown in the code of Figure 5. Both kernels scale with the number of cores as they use  $\mathcal{O}(|\mathbf{V}|)$  threads.

Existing sequential mesh compression techniques use a similar approach (“add-operation” [TG98], “new-vertex operation” [GS98]), however, no parallelization is yet given. The number of elements in  $\mathbf{U}$  is approximately 50 % the size of  $\mathbf{V}$ : Leaving out restart triangles,  $|\mathbf{V}|$  is the number of triangles. From the Euler characteristics, we know that the number of triangles is about twice the number of vertices for low-genus triangle meshes. Hence, 50 % of the entries in  $\mathbf{V}$  reference revisited vertices. Those are added to  $\mathbf{U}$ , and therefore  $|\mathbf{U}| = 50\% \cdot |\mathbf{V}|$ . For our test meshes,  $\mathbf{U}$  is with 51 % – 55 % the size of  $\mathbf{V}$  slightly larger, which is due to restart codes.

#### 6. Data-Parallel Word-Aligned Code

The strip generation algorithm of Section 3 puts the strips into an order that conforms with the ordering of the belts. Hence, neighboring belts are close within the triangle array  $\mathbf{T}$ . If a vertex is not part of the previous triangle and not an incremental vertex, it must re-reference a vertex from the current or the previous belt. In Figure 1d, these are the triangles in dark shades. Thus, neighboring elements of  $\mathbf{U}$  do not differ much in value. So it is customary to use a *delta code* for further compression, i.e.,  $\mathbf{D}[k] = \mathbf{U}[k] - \mathbf{U}[k - 1]$ . We could store the values of  $\mathbf{D}$  using  $\lceil \log_2(\max \mathbf{D} - \min \mathbf{D} + 1) \rceil$  bits per element. While this would already give good compression ratios, we further compress  $\mathbf{D}[k]$  with a scheme similar to entropy encoding. We do not store the values of  $\mathbf{D}[k]$  in the two’s complement. Instead, we map the signed values to unsigned values using a zigzag pattern, i.e., 0, -1, 1, -2, 2, ..., as done by Lindstrom and Cohen [LC10]. In this repre-

Selector	a	b	c	d	e	f	g	h	i
bitsPerCode	1	2	3	4	5	7	9	14	28
numCodes	28	14	9	7	5	4	3	2	1

Table 1: A Simple-9 codeword holds  $\text{numCodes}[s]$  codes, where each code uses  $\text{bitsPerCode}[s]$  bits.

sensation, the smaller the absolute value of a number is the more leading zero bits it has.

We make use of this property and apply Anh’s and Mof-fat’s *Simple-9* technique [AM05]. Instead of storing all bits of a code  $\mathbf{D}[k]$ , we omit leading zero bits and pack the remaining bits into *codewords*. Every codeword has a fixed size of 32 bits: 4 bits for a *selector* and 28 bits for *data bits*. The data bits are partitioned uniformly into codes of equal bit length. The 4 selector bits encode what partitioning is used, i.e., the code length and the number of codes per codeword, as shown in Table 1. To create a codeword  $\mathbf{S}[l]$ , we use a fast greedy approach: we collect values from  $\mathbf{D}$ , remove leading zero bits, and tightly pack the trailing bits in the data bits of  $\mathbf{S}[l]$ . If no more space is left in  $\mathbf{S}[l]$ , we proceed to the next codeword. For example in Figure 6,  $\mathbf{D}[0]$  uses 9 bits and  $\mathbf{D}[1]$  uses 14 bits. So we allocate 14 bits for both, completing the 28 bits we can have per codeword. Thus, we store them in  $\mathbf{S}[0]$  together with the selector **h**. Next up is  $\mathbf{D}[2]$  through  $\mathbf{D}[5]$ , all encoded using 7 bits in  $\mathbf{S}[1]$ , and so forth. For our test models,  $\mathbf{S}$  compresses on average 2.7:1 with respect to  $\mathbf{D}$  that uses  $\lceil \log_2(\max \mathbf{D} - \min \mathbf{D} + 1) \rceil$  bits per element.

To decompress a Simple-9-compressed array, we propose an efficient data-parallel three-pass algorithm. We unpack every codeword  $\mathbf{D}[k]$  individually as it is independent from all other codewords. The only thing we need to know is where to output the unpacked codes  $\mathbf{D}[k]$ . Figure 7 shows the first two passes of our algorithm. First with  $\mathbf{S\_Scan}$ , we find to every code  $\mathbf{S}[l]$  the location  $\bar{\mathbf{S}}[l]$  of its first code in the array of unpacked codes  $\mathbf{D}$ . Second, in  $\mathbf{D\_Scatter}$ , we spawn one thread for each codeword and scatter all unpacked codes of the codeword into  $\mathbf{D}$ . In the final third pass,  $\mathbf{U\_Scan}$ , we undo the zigzag mapping and use an inclusive

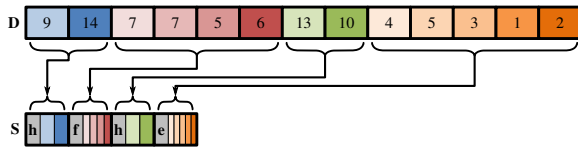


Figure 6: For Simple-9 compression, we remove leading zeros of the elements of  $\mathbf{D}$ . The numbers shown in  $\mathbf{D}$  indicate the number of the remaining bits. The elements of  $\mathbf{D}$  are packed into the data bits of the codeword array  $\mathbf{S}$ . Every codeword contains 28 data bits and a 4-bit selector, indicated by the lower-case letters.

add-scan over  $\mathbf{D}$  to invert the delta encoding. This gives us the array of reused vertices  $\mathbf{U}$ .

All passes of the algorithm offer a high degree of parallelism and scale well with the number of threads: The first two passes spawn  $\mathcal{O}(|\mathbf{S}|)$  threads and the third pass uses  $\mathcal{O}(|\mathbf{U}|)$  threads.

## 7. Results and Discussion

We test our algorithms on the models of Table 2. The heatmap colors on the images represent the belt traversal order, starting with blue for the first seed vertex. The compression algorithm compresses all models at a constant rate of about 922 thousand triangles per second ( $\pm 10\%$ , depending on the model). We provide the option to encode strip restarts either explicitly or by degenerate triangles, as explained in Section 4. This influences both compression ratio and decompression times. Therefore, in most tables we discriminate between explicit restart codes (columns  $R$ ) and emulation of restart codes (columns *deg.*).

We list bits per triangle (bpt) in column *compression rate*

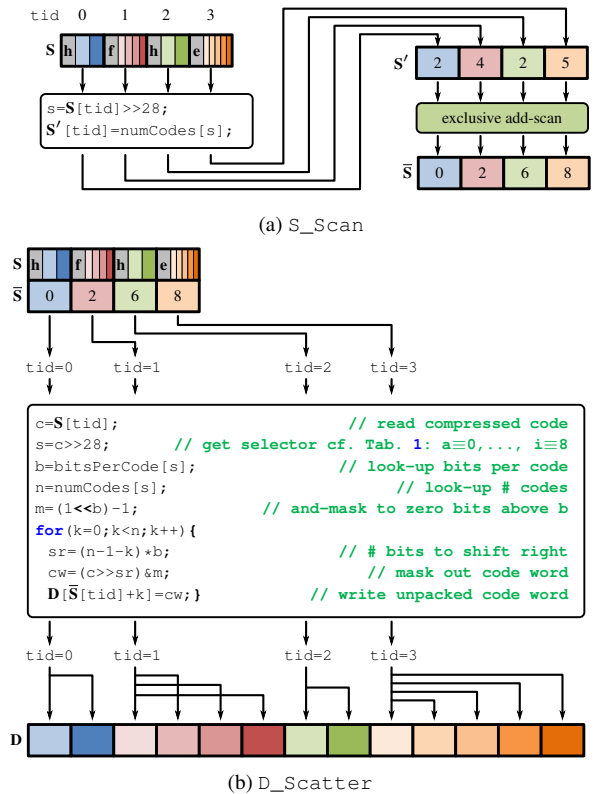


Figure 7: We unpack the Simple-9-compressed array  $\mathbf{S}$  to  $\mathbf{D}$ . (a) We apply a parallel add-scan over the number of codes in the selector bits of  $\mathbf{S}$  to create  $\bar{\mathbf{S}}$ . (b) To restore the unpacked data  $\mathbf{D}$ , we use one thread for each codeword  $\mathbf{S}[i]$ .

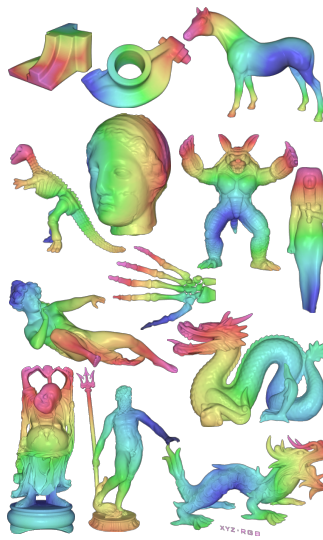
	models				compression rate		decompression speed			
	mesh	triangles	restart codes	val. 6 [%]	R [bpt]	deg. [bpt]	R [ms]	deg. [ms]	rate [Gtps]	render [ms]
Fan Disk	12,946	193	80	4.92	4.16	0.33	0.26	0.05	3.83	
Bunny	69,664	1,146	75	5.02	4.36	0.36	0.32	0.22	3.31	
Rocker Arm	80,354	1,637	65	5.27	4.61	0.38	0.35	0.23	4.71	
Horse	96,966	1,930	66	5.22	4.65	0.24	0.30	0.33	2.17	
Dinosaur	112,384	3,235	58	5.88	5.42	0.34	0.34	0.33	2.21	
Igea	268,686	4,429	66	5.03	4.35	0.53	0.38	0.71	4.18	
Armadillo	345,944	9,620	53	5.63	5.19	0.60	0.54	0.64	2.76	
Isis	374,309	3,877	64	4.66	3.84	0.47	0.42	0.89	2.25	
Angel	474,048	15,669	60	6.04	5.74	0.68	0.62	0.76	4.28	
Hand	654,666	24,704	53	6.23	6.00	0.78	0.64	1.03	3.07	
Dragon	869,928	49,744	33	7.40	7.59	0.98	0.93	0.94	6.07	
Buddha	1,087,436	61,873	32	7.42	7.61	1.12	0.98	1.11	3.24	
Welsh Dragon	2,210,673	19,823	87	4.53	3.72	1.81	1.56	1.42	6.49	
Neptun	3,316,916	40,800	84	4.76	4.01	2.52	2.13	1.56	4.86	
Asian Dragon	7,219,045	66,286	89	4.67	3.87	5.05	4.20	1.72	11.1	

Table 2: We use a set of standard *models* to benchmark our algorithms. We list the number of *triangles*, *restart codes*, and the percentage of valence-six vertices (*val. 6*). We list *compression rate* in bits per triangle (bpt) for the algorithm using explicit restart codes (*R*) and the one using degenerate triangles (*deg.*). *Decompression speed* is given in milliseconds (ms) for both *R*- and *deg.*-case. Additionally, we provide the *decompression rate* in billion triangles per second (Gtps) for the *deg.*-case. Timings to *render* the models without compression but with our triangle and vertex order are shown in the last column.

of Table 2. Our compression algorithms achieve compression ratios of as low as 3.7 bpt. When using degenerate triangles, we observe a median compression ratio of 4.6 bpt. Only the models *Dragon* and *Buddha* need more than 7 bpt. Similar to other compact representations [GLLR11a, GLLR11b], we observe that the more valence-six vertices a mesh possesses the better the compression ratio becomes. For example, 33 % of the vertices of the *Dragon* have valence six, whereas for the *Welsh Dragon*, 87 % of the vertices have valence six. The number of valence-six vertices indicates how regular a surface is meshed. Thus, the more regular a mesh is the better it compresses. When stored in the commonly used explicit representation (i.e., with three indices per triangle, as in the array **T**) on the GPU, most of our models consume 96 bpt. Small models can be stored with 48 bpt, since OpenGL supports 16-bit index buffers. Hence, we achieve compression ratios from 8:1 up to 21:1, and a median of 14:1. Note that for the models *Dragon* and *Buddha* the use of explicit restart codes pays off, since they require relatively many restart-codes.

### 7.1. Compression Rate

In Table 3, we representatively investigate the model with the lowest (*Welsh Dragon*) and highest (*Buddha*) bpt, and a model with a medium number of triangles and an average bpt (*Armadillo*). We analyze the compression rate achieved after each stage of our algorithm.

The row *explicit* shows the bpt achieved with the explicit

representation using  $3 \cdot \lceil \log_2(\text{number of vertices}) \rceil$  bpt. We use the input number of triangles as reference for computing bpt values. Therefore, the use of degenerate triangles bloats the storage per triangle for the explicit representation.

The row *strips* shows the benefits of using generalized triangle strips, as explained in Section 4. It cuts the cost by a factor of 2.5 – 2.8 which comes close to the strict upper bound of 3. However, this is an optimistic upper bound, as it ignores the overhead of one bit (*deg.*) or two bits (*R*) for each strip-code. Note that only the *Welsh Dragon* benefits from the use of degenerate triangles. This is because less than 1 % of all strip-codes are restarts. The *Armadillo* (3 %) and the *Buddha* (6 %) have a higher restart code frequency. Thus, they do not profit from degenerate triangles.

	Welsh Dragon		Armadillo		Buddha	
	R	deg.	R	deg.	R	deg.
explicit	66	68.4	54	60	60	73.7
strips	24.4	23.8	21.0	21.1	24.3	25.8
inc.	14.4	13.9	13.1	13.2	15.4	17.0
S-9	4.5	3.7	5.6	5.2	7.4	7.6

Table 3: Compression rates in bpt for three models achieved for the explicit representation (*explicit*), after creating generalized triangle strips (*strips*), after using incremental vertices (*inc.*), and after Simple-9 compression (*S-9*).



	kernel/scan	R		deg.	
		time	pct.	time	pct.
strips	M_Scan	0.27 ms	14 %	0.00 ms	0 %
	Q_Scan	0.31 ms	16 %	0.28 ms	14 %
	T_Kernel	0.45 ms	23 %	0.41 ms	21 %
inc.	INC_Scan	0.26 ms	13 %	0.26 ms	13 %
	V_Kernel	0.20 ms	10 %	0.20 ms	10 %
S-9	S_Scan	0.07 ms	4 %	0.07 ms	4 %
	D_Scatter	0.20 ms	10 %	0.20 ms	10 %
	U_Scan	0.18 ms	9 %	0.18 ms	9 %
total		1.94 ms	100 %	1.61 ms	83 %

Table 4: We break down the decompression times of the Welsh Dragon into sub-timings of the kernels and scans.

The row *inc.* shows the result achieved by incremental vertices, as explained in Section 5: A triangle that references a vertex firstly can infer the vertex by incrementing a counter. If a triangle references an already visited vertex, it has to store the vertex explicitly. This further improves the compression by a factor of about 1.6. When using degenerate triangles, the compression benefit is a little lower: In that case, each restart code is emulated by four extra degenerate triangles. At least two of them re-reference a vertex and thus we have to store them explicitly.

Row S-9 shows the additional benefits when compressing the re-referenced vertices using delta codes and Simple-9, as explained in Section 6. It improves compression by roughly 8 – 10 bpt. In contrast to the previous rows, *Armadillo* compresses better when using degenerate triangles: As the values of **D** become small, they require only few bits per vertex. Thus, emulating restart codes with four degenerate triangles becomes less expensive and this outweighs the initial benefit of using explicit restart codes.

## 7.2. Decompression Speed

We implement our decompression algorithm using CUDA 4.0 and use CUDPP [HOS\*11] for scan operations. The entries in the column *decompression speed* of Table 2 are measured on an Nvidia GeForce 580 GTX including CUDA-OpenGL context switches and buffer mapping times. We further provide the triangle *rate* in billion triangles per second (Gtps) and observe up to 1.72 Gtps. Rendering timings, shown in column *render*, exclude decompression timings, but are measured using the vertex and triangle order of Section 3. During rendering, we did not observe any difference between the versions with explicit restart codes and degenerate triangles, as degenerate triangles are removed by the pipeline at no extra cost. We used a resolution of 1920x1200 using OpenGL 4.2 and a simple Blinn-Phong shading model. Under these circumstances, our decompression for the deg.-case makes only (min: 6 %, avg: 15 %, max: 30 %) of the

total rendering cost. Hence, our algorithm is well suited for decompressing models every frame. When using more sophisticated shaders, the ratio between rendering and decompression times increases. This makes the use of triangle decompression even more attractive. In almost all cases, we observe that decompression is faster with degenerate triangles than with restart codes.

Table 4 lists detailed timings spent on the different parts of our algorithm at the example of the *Welsh Dragon* model. Note that the *total* timings are slightly higher than the one listed in Table 2, as the additional timing code comes with some overhead. The row *strips* details the timings for unpacking the strips (Section 4). The most noteworthy difference between the two methods is that M\_Scan is not required when using degenerate triangles. With the array  $\bar{\mathbf{M}}$  given implicitly, Q\_Scan and T\_Kernel are less complex, and thus take less time to compute. However, most of the speed-up is attributed to the reduced bandwidth consumption, as the kernels and scans do not have to load  $\bar{\mathbf{M}}$ . The timings for the use of incremental vertices (row *inc.*, cf. Section 5) and Simple-9 (row S-9, cf. Section 6) take equally long for both methods. This is not surprising, as degenerate triangles do not significantly enlarge their workloads. As the computational density is low for all our kernels and scans, our algorithm is bandwidth bound. To hide memory latency, we need as many CUDA threads as possible. This is the reason why large meshes decompress faster than small meshes, as shown in Table 2: the larger the mesh is the more threads we can use, and thus, the better the decompression speed is.

## 7.3. Impact of Vertex and Triangle Order

We achieve the reported compression ratios by ordering triangles and vertices coherently. In most cases, applications do not rely on a particular triangle order. If they do, our decompression also works; however, the compression ratio may be worse. The rendering speed of graphics hardware depends on both triangle and vertex order. For good performance, it is recommended to use cache oblivious layouts [Kil08]. Table 5 shows that frame-times achieved with our order are similar to the order computed with OpenCCL, a library that creates cache oblivious mesh layouts [YLP05]. This comes not unexpectedly, as performance of GPUs with unified shader architecture increases if triangles and vertices are ordered coherently (see [Kil08], Section 7.2).

	Welsh Dragon	Armadillo	Buddha
Our order	6.48 ms	2.73 ms	3.24 ms
OpenCCL	6.84 ms	2.96 ms	3.14 ms

Table 5: We compare the pure rendering times without decompression achieved with our triangle and vertex order against the one of OpenCCL.

#### 7.4. Runtime Memory Consumption

Besides the memory space for the input arrays  $\mathbf{S}$ ,  $\mathbf{INC}$ , and  $\mathbf{C}$ , our algorithm needs space for  $3 \cdot |\mathbf{T}|$  vertices to store the outputted explicit representation. While our prototype implementation explicitly reserves memory for all temporary arrays for debugging purposes, a production application would need temporary memory of only  $|\mathbf{V}|$ : There is sufficient space in the output array  $\mathbf{T}$  for the temporary arrays  $\bar{\mathbf{S}}$ ,  $\mathbf{D}$ , and  $\mathbf{U}$ , used for unpacking word-aligned codes. After computing  $\mathbf{V}$ , the temporary space from word-aligned-code decompression can be reused for  $\bar{\mathbf{M}}$  and  $\bar{\mathbf{Q}}$ .

#### 7.5. Application Fields

Our approach works well with out-of-core algorithms: Remember our median compression ratio is at 4.6 bpt. This is almost 21 times smaller than required for the uncompressed explicit representation, which has 96 bpt. Thus, 21 times more triangle topology fits in GPU memory, which dramatically decreases the necessity for expensive CPU-to-GPU memory transfers.

We compare our compression scheme with degenerate triangles against the explicit representation using parts of the David model from the Digital Michelangelo Project [LPC\*00]. The mesh consists of 28 sub-meshes shown in different colors in Table 6. In total, there are 191 million triangles. In the explicit representation, we need 32 bits per index. This amounts for 2.13 GiB of data for triangle topology. Using our approach with degenerate triangles, we obtain 5.91 bpt, resulting in 0.131 GiB for the topology, which is 6 % the size of the explicit representation. For both compressed and non-compressed versions of the David mesh, we quantize vertex positions with 16 bits per component and we use 16-bit octahedron normals [MSS\*10] to fit all geometry into a single 64-bit word per-vertex, instead of  $6 \cdot 32\text{bits} = 192\text{bits}$ . We convert the per-vertex attributes to floating-point numbers in the vertex-shader with only a few instructions, so the overhead is negligible. Thus, the vertex attributes consume 0.715 GiB instead of 2.15 GiB. When using our topology compression scheme, all data fits in the GPU memory of our test-system. During rendering, we de-

compress one sub-mesh at a time and draw it directly after decompression. Then, we proceed with the next sub-mesh, until all sub-meshes are processed. This way, the memory for the temporary arrays (see Section 7.4) has to fit the largest sub-mesh only. Table 6 shows that using our compression scheme is 20 % faster than using the non-compressed explicit representation. This is because the compressed version requires a minimum amount from the off-GPU system memory. As opposed to the non-compressed version, no data needs to be swapped between CPU and GPU. Unlike our CUDA decompression, rendering is asynchronous to transferring data from CPU to GPU. If we were able to use some of the CUDA cores for decompression and others for rendering, our algorithm would benefit even more in terms of rendering speed.

#### 7.6. Comparison

Clearly, our algorithms do not achieve bit rates of sequential algorithms that reach around 1 bpt. We need about five times more space for topology. However, our algorithms target many-core systems and thus scale well with future hardware as opposed to sequential methods. Our algorithms are designed for on-the-fly unpacking of triangle topology at render time or for faster GPU upload of topology data. The performance that is needed in this setup is achieved with an unpacking scheme that exploits parallelism even on highly parallel processors. Our algorithms unpack triangle topology at roughly the same rate as a GPU can render it. Compression times, even for complex meshes, are in the range of seconds, so compression can be done at loading time. While our algorithms do not achieve bit rates of sequential algorithms, they still compress topology data to less than 10 %.

We compare our timings and memory consumptions with two recent papers [CH09, GLLR11b]. Note that both of these papers have a very different focus. They compress more data (e.g., also neighboring information or vertex data) and allow random access to single triangles, features that are not supported by our algorithm. For our applications, these types of functionality are not necessary. Instead, we need high decompression speed and good scalability behavior.

In 2009, Courbet and Hudelot [CH09] described a method to compress triangles down to 3 bpt using a sequential algorithm. They report access times of  $1\mu\text{s}$  per vertex. Our maximum of 1.7 billion triangles per second is equal to an access time of 0.2 ns per vertex. Though hardware has become faster since then and the reported timings include fetching 3d positions, we do not anticipate that this may compensate for four orders of magnitudes in speed. Most recent data structures [GLLR11b] are designed for compactness and to quickly access neighborhood information, a feature we do not support. Hence, they require about 26 bpt, i.e., five times more memory than we do. They report CPU access times from their non-optimized compact version of about 20 ns, i.e., our algorithm is two orders of magnitudes faster. The

	compressed	non-compressed
compression ratio	5.91 bpt	96 bpt
frame-time	248 ms	298 ms
GPU memory	1.41 GiB	1.39 GiB
system memory	0.0315 GiB	1.53 GiB



Table 6: The 28 sub-meshes of David (right) are shown in different colors. It consists of 191 million triangles. We compare our topology compression scheme with degenerate triangles against the non-compressed explicit representation.

authors briefly describe a geometry-shader-based GPU implementation, yet they give no timings. So it remains unclear how their GPU implementation compares with ours.

## 8. Conclusion and Future Work

In this paper, we presented a data-parallel algorithm for fast decompression of triangle topology. We decompress at a rate of 1.7 billion triangles per second and we compress to about 5 bpt. To achieve these results, we proposed an algorithm to order triangles and vertices coherently. We contributed a method that decompresses generalized triangle strips in a data-parallel fashion. Further, we proposed a data-parallel algorithm for decompressing word-aligned codes. To our knowledge, no prior triangle decompression algorithm runs with a comparable high degree of parallelism as the algorithm presented here.

In the future, we want to extend our approach to compress vertex attributes, such as positions, normals, and texture coordinates. While our belt traverser is simple, fast, and already yields low bit-rates, we believe that we can still improve bit-rates without changing the GPU decompression algorithms. For example, an order guided by the Fiedler vector [IL05], or generated by stripification algorithms [GE04] might produce better compression results, however, at a higher effort than our approach. Moreover, we need to see how that order interplays with the incremental vertices and our word-aligned code.

## Appendix: CUDA Implementation

In this section, we detail how our approach works on CUDA, which is crucial for obtaining the reported decompression performance. Many of our kernels (M\_Scan, Q\_Scan, INC\_Scan, S\_Scan U\_Scan) involve scans, so the performance of the overall system heavily depends on a fast implementation of a data-parallel scan. Table 4 reveals that over 50 % of the overall decompression time is due to a scan. Luckily, scans are a very common operation in Computer Graphics (e.g., [PO08, HZG08, LGS\*09, LHLK10]) and many (open-source) libraries provide scan implementations [HOS\*11, Mic10, HB11]. Most important for the scalability of our algorithm is that a scan scales linearly with the available bandwidth on current GPUs [MG09]. Scan implementations are based on algorithms proposed by Blelloch [Ble90], and Hillis and Steele [HS86]. For a detailed description of a CUDA scan implementation, we have to refer to the relevant articles [HS007, SHG08, BOA09].

We did not implement a scan ourselves and used the highly optimized CUDPP library [HOS\*11]. We follow the suggested thread allocation and the distribution of array elements to threads. However, we carried out two minor modifications. First, we added support for reading from bit-streams. We tightly pack the elements without fragmentation

in a *uint4* and use logical bit operations to extract the individual elements. This applies to **C** (one bit without, two bits with *R*-codes per element) and **INC** (one bit per element). The second modification is to support *transform scans*. It is defined similar to a regular scan of Equation 1 with a little modification, i.e., we transform  $\mathbf{A}[i-1]$  using a function  $f$ :

$$\bar{\mathbf{A}}[i] := \begin{cases} 0, & i = 0 \\ op(\bar{\mathbf{A}}[i-1], f(\mathbf{A}[i-1])), & i > 0. \end{cases} \quad (9)$$

A similar definition applies for the inclusive scan of Equation 2. The code boxes in Figure 4a, 4b, and 7a show the respective definitions of the functions  $f$ . Instead of modifying CUDPP to support transform scans, we could have launched a separate kernel call prior to every scan. However, this would dramatically increase the bandwidth pressure and thus degrade performance significantly, as every array element would require an extra read and write to global memory. The remaining scan-kernels require no transformations.

Each input and output element in **T\_Kernel** (Figure 4c) and **V\_Kernel** (Figure 5) is assigned to one thread, and so memory access is coalesced for **C** and **INC**. Also **T\_Kernel** accesses **V** in a linear manner, resulting in high data throughput. We got the best results with 512 threads per block. **D\_Scatter** is the only kernel that has to access data sequentially, however, only for writing. Each thread reads one 32-bit words of **S** and writes up to 28 words to **D**. In experiments, 64 threads per block give the best results.

## Acknowledgments

We thank Stanford University (Bunny, Buddha, Armadillo, Dragon, Asian Dragon), The Digital Michelangelo Project (David), Georgia Institute of Technology (Angel, Horse, Hand), Cyberware Inc. (Dinosaur, Igea, Isis, Rocker Arm), AIM@SHAPE (Neptun), and Bangor University, UK (Welsh Dragon) for providing test data sets.

## References

- [AG03] ALLIEZ P., GOTSMAN C.: Recent advances in compression of 3d meshes. In *Proceedings of the Symposium on Multiresolution in Geometric Modeling* (2003), Springer, pp. 3–26. 2
- [AM05] ANH V. N., MOFFAT A.: Inverted index compression using word-aligned binary codes. *Information Retrieval* 8 (2005), 151–166. 7
- [Ble90] BLELLOCH G. E.: *Vector models for data-parallel computing*. MIT Press, 1990. 4, 11
- [BOA09] BILLETER M., OLSSON O., ASSARSSON U.: Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), pp. 159–166. 11
- [CH09] COURBET C., HUDELLOT C.: Random accessible hierarchical mesh compression for interactive visualization. *Computer Graphics Forum* 28, 5 (2009), 1311–1318. 2, 10
- [Cho97] CHOW M. M.: Optimized geometry compression for real-time rendering. In *IEEE Visualization '97* (1997), pp. 346–354. 3

- [CK07] CHHUGANI J., KUMAR S.: Geometry engine optimization: Cache friendly compressed representation of geometry. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (2007), pp. 9–16. [3](#)
- [CKLL09] CHOE S., KIM J., LEE H., LEE S.: Random accessible mesh compression using mesh chartification. *IEEE Transactions on Visualization and Computer Graphics* 15, 1 (2009), 160–173. [2](#)
- [Dee95] DEERING M. F.: Geometry compression. In *SIGGRAPH '95* (1995). [3](#)
- [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. [3, 6](#)
- [GE04] GOPI M., EPPSTEIN D.: Single-strip triangulation of manifolds with arbitrary topology. *Computer Graphics Forum* 23, 3 (2004), 371–380. [11](#)
- [GGK02] GOTSMAN C., GUMHOLD S., KOBELT L.: Tutorials on multiresolution in geometric modelling. pp. 319–362. [2](#)
- [GLLR11a] GURUNG T., LANEY D. E., LINDSTROM P., ROSSIGNAC J.: SQuad: Compact representation for triangle meshes. *Computer Graphics Forum* 30, 2 (2011), 355–364. [2, 8](#)
- [GLLR11b] GURUNG T., LUFFEL M., LINDSTROM P., ROSSIGNAC J.: LR: Compact connectivity representation for triangle meshes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)* 30, 4 (2011), 67:1–67:8. [2, 8, 10](#)
- [GS98] GUMHOLD S., STRASSER W.: Real time compression of triangle mesh connectivity. In *SIGGRAPH '98* (1998), pp. 133–140. [2, 6](#)
- [HB11] HOBEROCK J., BELL N.: *Thrust Library*, 2011. <http://code.google.com/p/thrust/>. [11](#)
- [HLK01] HO J., LEE K.-C., KRIEGMAN D.: Compressing large polygonal models. In *IEEE Visualization 2001* (2001), pp. 357–362. [2](#)
- [HOS\*11] HARRIS M., OWENS J. D., SENGUPTA S., TSENG S., ZHANG Y., DAVIDSON A.: *CUDPP*, 2011. <http://code.google.com/p/cudpp/>. [4, 9, 11](#)
- [HS86] HILLIS W. D., STEELE JR. G. L.: Data parallel algorithms. *Communications of the ACM* 29, 12 (1986), 1170–1183. [11](#)
- [HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). Addison Wesley, 2007. [11](#)
- [HZG08] HOU Q., ZHOU K., GUO B.: BSGP: Bulk-synchronous GPU programming. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)* 27, 3 (2008), 19:1–19:12. [11](#)
- [IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. *ACM Transactions Graphics (Proceedings of SIGGRAPH 2003)* 22, 3 (2003), 935–942. [2](#)
- [IL05] ISENBURG M., LINDSTROM P.: Streaming meshes. In *16th IEEE Visualization Conference* (2005). [11](#)
- [Ise00] ISENBURG M.: Triangle fixer: Edge-based connectivity compression. In *EuroCG* (2000). [2](#)
- [Kil08] KILGARD M. J.: Modern OpenGL usage: Using vertex buffer objects well. In *SIGGRAPH ASIA 2008 Course Notes* (2008), pp. 49:1–49:19. [9](#)
- [KPRW05] KÄLBERER F., POLTHIER K., REITEBUCH U., WARDETSKY M.: Freelence – coding with free valences. *Computer Graphics Forum* 24, 3 (2005), 469–478. [2](#)
- [LC10] LINDSTROM P., COHEN J. D.: On-the-fly decompression and rendering of multiresolution terrain. In *Proceedings of the 2010 Symposium on Interactive 3D Graphics* (2010), pp. 65–73. [3, 6](#)
- [LGS\*09] LAUTERBACH C., GARL M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast BVH construction on GPUs. *Computer Graphics Forum* 28, 2 (2009), 375–384. [11](#)
- [LHLK10] LIU F., HARADA T., LEE Y., KIM Y. J.: Real-time collision culling of a million bodies on graphics processing units. *ACM Transactions on Graphics* 29, 6 (2010), 154:1–154:8. [11](#)
- [LPC\*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The Digital Michelangelo Project: 3D scanning of large statues. In *SIGGRAPH '00* (2000), pp. 131–144. [10](#)
- [MG09] MERRILL D., GRIMSHAW A.: *Parallel Scan for Stream Architectures*. Tech. Rep. CS2009-14, Department of Computer Science, University of Virginia, 2009. [11](#)
- [Mic10] MICROSOFT: *Direct3D 11 Graphics*. Microsoft, 2010. Available online at <http://msdn.microsoft.com/en-us/library/windows/desktop/ff476080.aspx>. [11](#)
- [MSGs11] MEYER Q., SUSSNER G., GREINER G., STAMMINGER M.: Adaptive level-of-precision for GPU-rendering. In *Proceedings of the Vision, Modeling, and Visualization Workshop 2011* (2011), pp. 169–176. [1](#)
- [MSS\*10] MEYER Q., SÜSSMUTH J., SUSSNER G., STAMMINGER M., GREINER G.: On floating-point normal vectors. *Computer Graphics Forum* 29, 4 (2010), 1405–1409. [1, 10](#)
- [PBCK05] PURNOMO B., BILODEAU J., COHEN J. D., KUMAR S.: Hardware-compatible vertex compression using quantization and simplification. In *Graphics Hardware 2005* (2005), pp. 53–62. [1](#)
- [PKJK05] PENG J., KIM C.-S., JAY KUO C. C.: Technologies for 3D mesh compression: A survey. *Journal of Visual Communication and Image Representation* 16, 6 (2005), 688–733. [2](#)
- [PO08] PATNEY A., OWENS J. D.: Real-time Reyes-style adaptive surface subdivision. *ACM Transactions on Graphics* 27, 5 (2008), 143:1–143:8. [11](#)
- [Ros99] ROSSIGNAC J.: Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization and Computer Graphics* 5, 1 (1999), 47–61. [2](#)
- [SHG08] SENGUPTA S., HARRIS M., GARLAND M.: *Efficient Parallel Scan Algorithms for GPUs*. Tech. rep., Nvidia Cooperation, 2008. [11](#)
- [TG98] TOUMA C., GOTSMAN C.: Triangle mesh compression. In *Proceedings of the Graphics Interface 1998 Conference* (1998). [2, 6](#)
- [Tut62] TUTTE W. T.: A census of planar triangulations. *Canadian Journal of Mathematics* 14 (1962), 21–38. [2](#)
- [YL07] YOON S.-E., LINDSTROM P.: Random-accessible compressed triangle meshes. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (2007), 1536–1543. [2](#)
- [YLPm05] YOON S.-E., LINDSTROM P., PASCUCCHI V., MANOCHA D.: Cache-oblivious mesh layouts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)* 24 (2005), 886–893. [9](#)