LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-423218

Streaming Compression of Hexahedral Meshes

M. Isenburg, C. Courbet

February 5, 2010

Computer Graphics International
Singapore, Singapore
June 9, 2010 through June 11, 2010

Disclaimer

Noname manuscript No.
(will be inserted by the editor)

Streaming Compression of Hexahedral Meshes

paper 208

the date of receipt and acceptance should be inserted later

Abstract We describe a method for *streaming* compression of hexahedral meshes. Given an interleaved stream of vertices and hexahedra our coder incrementally compresses the mesh in the presented order. Our coder is extremely memory efficien when the input stream documents when vertices are referenced for the last time (i.e. when it contains topological finalizatio tags). Our coder then continuously releases and reuses data structures that no longer contribute to compressing the remainder of the stream. This means in practice that our coder has only a small fraction of the whole mesh in memory at any time. We can therefore compress very large meshes—even meshes that do not fi in memory.

Compared to traditional, non-streaming approaches that load the entire mesh and globally reorder it during compression, our algorithm trades a less compact compressed representation for significan gains in speed, memory, and I/O efficien y. For example, on the 456k hexahedra "blade" mesh, our coder is twice as fast and uses 88 times less memory (only 3.1 MB) with the compressed fil increasing about 3% in size. We also present the firs scheme for predictive compression of properties associated with hexahedral cells.

1 Introduction

Numerical simulations require the computation domain to be discretized. For some problems a simple uniformly structured grid is sufficient However, the use of unstructured meshes becomes necessary when the domain boundary is complex or when the size or shape of cells needs adaptation to the physics of the simulation. Although mixed element meshes are sometimes used, most meshes only have a single cell type: for 3D problems these are either tetrahedra or hexahedra. Recently hexahedral meshes have received a lot of attention because of several desirable properties: they usually enable to build meshes with fewer elements and exhibit better numerical behavior in various problems, e.g. for stress analysis [1]. Although the automatic generation of hexahedral meshes tends to be more difficul than that of tetrahedral meshes, there are now a number of algorithms [2, 16, 18] that can generate high-quality hexahedral meshes.

The improvements in technology over the last decade have allowed scientists to run more and more accurate numerical simulations using larger and larger meshes. However, the growth in computing power has far outpaced that of memory capacity and bandwidth. This poses several problems for the handling of large meshes. Tremendous amounts of disk space have to be used for raw data storage, and a lot of time is spent on data transfer, to the point that simulations are generally I/O bound. To alleviate this problem, specialized mesh compression algorithms are used to decrease the size of the mesh and attached quantities.

In this article, we present an algorithm for streaming compression of hexahedral meshes. Previous, non-streaming approaches for compressing hexahedral meshes aim solely at the lowest possible bit-rate. Due to their large memory requirements and their inefficien I/O behavior they are slow or even unable to compress very large meshes. Our compression and decompression methods on the other hand are specificall designed to keep memory footprint low and to interleave computation and I/O. We follow a strategy that has already proven successful for compressing triangle [12] and tetrahedral [13] meshes but adapt it for hexahedral elements. Furthermore, we also present the—to our knowledge—firs method for predictive compression of data associated with the hexahedral cells of an unstructured mesh.

Address(es) of author(s) should be given

2 Preliminaries

Unstructured hexahedral meshes consist of *n vertices* and *m cells* and typically *n* is just slightly higher than *m*. Each vertex has an *x*, *y*, and *z* coordinate that specify a discrete location. These coordinates are called the *geometry* of the mesh. Each cell references eight different vertices to form a hexahedral element. This connects the vertices into a graph-like structure that is called the *connectivity* of the mesh. Often there are also mesh *properties* (e.g. pressure, temperature or velocity values) that are associated with vertices or cells.

Standard indexed formats store the geometry as a floa array with three coordinates per vertex and the connectivity as an integer array with eight indices per hexahedron. Optional properties are stored in corresponding arrays. Vertices and hexahedra can appear in any permutation in the arrays. This convenience comes at the price of bloated connectivity storage costs at $8 \log_2 n$ bits per hexahedron. These costs can in practice be reduced to a constant number of bits per hexahedron by enforcing a more "canonical" order onto the mesh. This is the strategy of most mesh compressors.

2.1 Mesh Compression

Most compression schemes are specialized for meshes with either triangular [19, 12], polygonal [6], tetrahedral [3, 13], or hexahedral [7, 14] elements (with [17] being an exception). Connectivity and geometry are usually encoded with clearly distinct (but often interwoven) techniques as one is of combinatorial and the other of numerical nature. Most coders firs encode the connectivity and then use it to assist with geometry compression. Floating-point geometry and properties are often quantized to a user-specifie number of precision bits, but fully lossless schemes exists as well [11].

The fundamental assumption of all compression schemes (except [15]) is that the particular order in which mesh vertices and mesh elements are stored does not need to be preserved. Most schemes firs construct the mesh connectivity graph, traverse it in some deterministic manner, and encode vertices and elements as they are encountered—thereby reordering them. Hence, the original *layout* of the mesh is irrevocably lost. This has been acceptable as—historically—the original mesh layout was rarely intentional but merely an artifact of the application that generated the mesh [9].

2.2 Hexahedral Mesh Compression

Three algorithms have been published on the subject of hexahedral mesh compression [7, 14, 15]. Isenburg and Alliez [7] extend the concept of degree coding (that was introduced by Touma and Gotsman for triangle meshes [19]) to compress the connectivity of hexahedral meshes. Their algorithm grows an *active hull* by traversing the connectivity graph one hexahedron at a time. They record the degree of all unseen edges (and a few special symbols) which allows the decoder to replay this traversal. Entropy coding of edge degrees results in connectivity compression rates that range from 1.55 bph (bits per hexahedron) down to 0.18 bph on our test set.

The algorithm of Krivograd et al. [14] is based on the same idea, but uses vertex instead of edge degrees. They firs compress the quadrilateral boundary surface of the volume mesh which becomes the initial hull that is then grown as in Isenburg and Alliez's method [7]. They obtain rates similar to [7] on regular grids but are worse on irregular models.

Lindstrom and Isenburg [15] propose a radically different compressor that neither reorders vertices nor hexahedra and is therefore *completely* lossless. It also handles non-manifold meshes or degenerate elements. They compress connectivity directly in its indexed form by predicting the eight indices of a hexahedron from preceding ones. This works because hexahedral meshes found in practice tend to have regular strides between indices of subsequent hexahedra. Their algorithm is an order of magnitude faster and has much lower memory consumption than [7] because it does not reconstruct and traverse the mesh connectivity. Its connectivity compression rates strongly depend on regularities in the indexing and are as high as 20.4 bph on our test set.

Prat *et al.* [17] have an algorithm to compress arbitrary manifolds—including hexahedral meshes. The genericity of their method negatively impacts compression rates (+400% on average compared to [7]) making it uncompetitive compared to a dedicated hexahedral mesh compressor.

2.3 Streaming Compression

Most mesh compression algorithms assume that the entire mesh fit in main memory. Three solutions have been proposed to deal with large meshes that defy this assumption: cut the mesh into smaller pieces and compress them one by one [4], employ external memory structures that page mesh parts from disk when needed [8], or represent the mesh as a *stream* of interleaved vertices, elements, and finalizatio tags and design a new compression scheme that can encode such a *streaming mesh* [9] on-the-fl as it streams in [12, 13].

Streaming compression is very memory efficien when the interleaved stream of vertices and elements contains *finalization tags* that document when a vertex was referenced for the last time. This information enables the compressor to continuously release and reuse data structures making it possible to compress gigantic meshes which cannot be handled by non-streaming algorithms. For example, the streaming triangle mesh compressor of Isenburg *et al.* [12] encodes the 6 GB "St. Matthew" model using less than 5 MB of main memory. This paper describes how to design a similar streaming compressor but for hexahedral meshes.

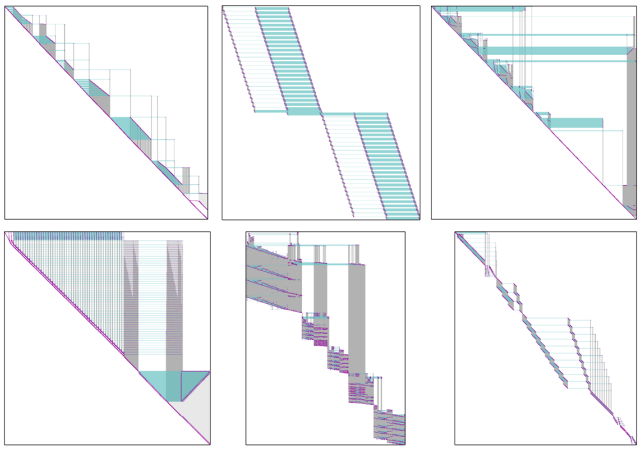

Fig. 1 The layout diagrams [9] for "blade", "block", "crank", "mdg", "shaft", and "steven" show that *vertex-compaction* [9] is sufficien to stream these meshes: any vertical line crosses only few green segments.

3 Streaming Hexahedral Mesh Compression

Our compressor requires streaming input: an interleaved sequence of vertices, hexahedra, and finalizatio tags. Mesh generators can easily be modifie to produce meshes in a streaming format. We can, in fact, directly compress their output and avoid having to store the uncompressed mesh altogether. Existing meshes in non-streaming formats need to be converted. For reasonably coherent meshes this can be done with *vertex-compaction* [9]. Isenburg and Lindstrom use *layout diagrams* to illustrate the coherence of a mesh.

The layout diagrams of several meshes are shown in Figure 1. Along the *y*-axis are the vertices (from top to bottom) and along the *x*-axis are the hexahedra (from left to right) in the original order they are stored in their array. The green segments (horizontal) connect all hexahedra that reference the same vertex. The width of the streaming mesh created with vertex-compaction equals the maximal number of green segments cut by a vertical line—reordering the vertices in order of firs reference merely permutes these segments vertically. Hexahedral meshes tend to have fairly coherent layouts (unlike tetrahedral meshes [13]). Their width after vertex-compaction is around $.5 - 5\%$ of their size.

Our compressor starts encoding the mesh as soon as the firs hexahedron and its eight vertices are received. It always encodes if and how the current hexahedron is adjacent to previously encoded hexahedra, compresses all new vertices that are referenced for the firs time, and then deallocates the data structure associated with all vertices that are referenced for the last time (i.e. that are *finalized*). As only the active vertices have to be stored in memory, the width of the streaming mesh determines maximum memory consumption. In the following section, we detail the compression of the three components of a mesh: *connectivity*, *geometry* (and vertex properties), and *cell properties*.

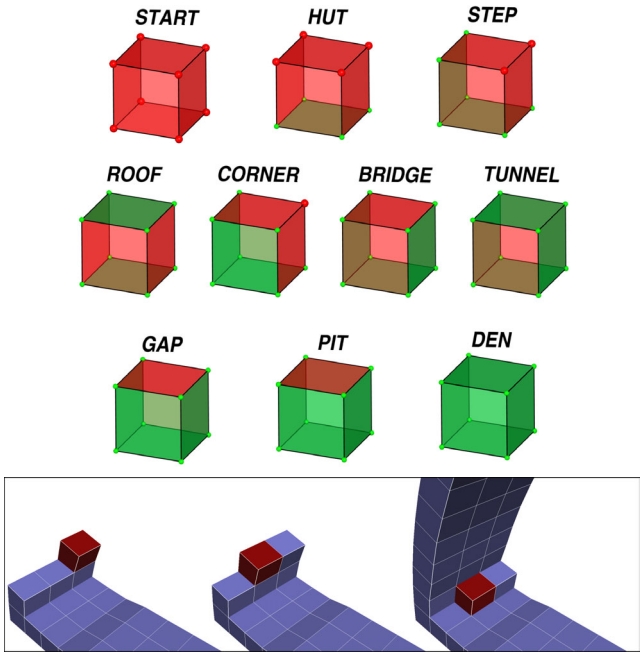

Fig. 2 The ten possible hexahedron configurations green faces and vertices are active, red faces are new, and red vertices are either new or joined. At the bottom, the three most frequent operations *HUT*, *STEP* and *CORNER* in context. Together with the *START* operation, they are sufficien to encode a regular grid in scanline order.

3.1 Connectivity Compression

Like Isenburg *et al.* [13], we maintain an *active surface*, a half-edge structure composed of active vertices and quadrilateral faces. A vertex of the current hexahedron is added to the active surface when it is referenced for the firs time and removed when it is finalized A face of the current hexahedron is added to the active surface when it was previously *not* part of it and removed otherwise. Faces are also removed after all their vertices were finalize (i.e. boundary faces).

The ten ways in which a hexahedron can be face-adjacent to the active surface are illustrated in Figure 2. The vertices shown in red for the *START*, *HUT*, *STEP* and *CORNER* configuratio are usually new and will be compressed (see Section 3.2). Occasionally, however, these vertices are already part of the active surface. Such *joined* vertices are specifie using dynamic indexing [12] with $log_2(width)$ bits.

For coding efficien y we rotate the current hexahedron into a *canonical* configuration For the *BRIDGE* configuration, for example, the active faces will always be f_0, f_1 and f_3 after rotating (see Figure 3). Then we only need to code the configuratio type and the following information:

- START: We code for all 8 vertices if they are new or joined. For coherent meshes there is usually only one *START* per component.
- HUT: We specify the face f_0 on the active surface the hexahedron is adjacent to and we code whether the 4 vertices are new or joined.
- ROOF: We specify f_0 on the active surface, code v_4 with dynamic indexing, and code which of v_4's adjacent faces is f_5.

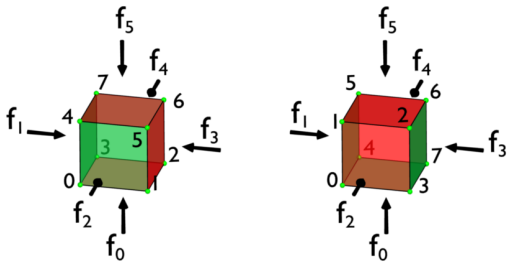

Fig. 3 Going from a randomly oriented *BRIDGE* configuratio (left) to a canonical *BRIDGE* configuratio (right). On the right, the vertices have been rotated such that the active faces are f_0, f_1 and f_3.

- STEP: After we specify f_0 we can fin face f_1 incident to the shared edge (which is known due to the canonical order). There is often only one candidate so that specifying face f_1 is in most cases free. For 2 vertices we code if they are new or joined.
- BRIDGE: We proceed as for *STEP* to specify f_1 and f_3.
- CORNER: We code a *STEP* and let the decoder deduce the following: if there exists an active face which contains vertices v_1,v_0,v_4 then the operation is a *CORNER* and this face is f_2. This works in all but one very special case illustrated in Figure 4. Here the face exists but is not f_2. To assure the decoder makes the correct decision, we output a confirmatio symbol. This special case is rare and happens only once in all our test models. Thus, the confirma tion symbol is nearly always the same and adds almost nothing to the bit budget. We also code if the last vertex is new or joined.
- TUNNEL: We code a *BRIDGE* and let the decoder deduce that this is in fact a *TUNNEL* operation whenever there exists an active face with vertices v_4,v_5,v_6,v_7. There is no ambiguity in this case.
- GAP: We code a *STEP* and firs let the decoder deduce that it is a *CORNER* and then—with the same reasoning—that it is a *GAP*.
- PIT: We code a *STEP* and let the decoder deduce a *GAP* and then—again with the same reasoning—that it is a *PIT*.
- DEN: We take the reasoning of *PIT* one step further.

To summarize: the coder will only distinguish between *START*, *HUT*, *ROOF*, *STEP* or *BRIDGE*. Everything else is deduced by the decoder. Each operation except *START* has to specify the face f_0 on the active surface. Doing this each time with dynamic indexing [12] would be very costly.

When consecutive hexahedra share a face we can specify face f_0 very efficientl by caching the 6 faces of the previous hexahedron. If we fin face f_0 in the cache we can code it with $log_2(6)$ instead of $log_2(width)$ bits. Using context-based entropy coding we can further decrease these costs as different configuration share faces with similar regularity. When face f_0 is not in the face cache we use the same idea with an edge cache and a finall a vertex cache. We only resort to dynamic indexing when this all fails.

In Table 2 we list the compression rates of our scheme on different models and compare them to those of [7], [14] and [15]. As expected, our rates are worse than those of degree-based methods since we compress the hexahedra in their original order. Unsurprisingly, our method outperforms the lossless coder that does not reorder vertices as we do and also preserves the original orientation of hexahedra. The penalty for streaming is highest for meshes with global regularity (e.g. "block" or "cylinder") that our compressor cannot exploit. Overall however, after also compressing geometry and properties, the connectivity accounts for a comparatively small amount of the total bit budget.

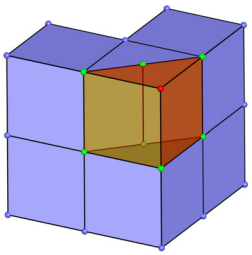

Fig. 4 Special *CORNER* case: the bottom face is active and shares three vertices with the current hexahedron. The confirmatio bit is set to 1, and the operation is a *CORNER*. However, the top face should no be shared. Thus, the second confirmatio bit set to 0, meaning that the operation is not a *GAP*.

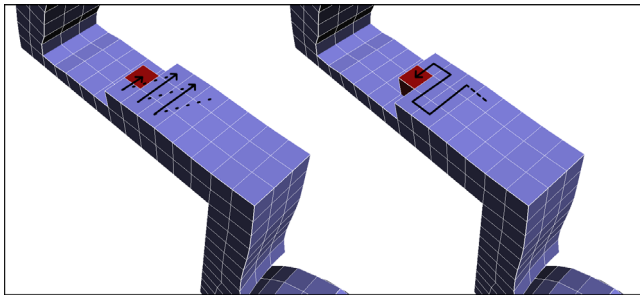

Fig. 5 A closeup on the *shaft* model, original order (left) and reordered using a buffer of three hexahedra (right). The original order causes a cache miss every three hexahedra, the new order has no cache misses.

3.1.1 Local Reordering

Locally, hexahedral meshes tend to have the connectivity of a grid whose cells are often specifie in scanline order (see Figure 5). There will be a cache miss for each scanline as soon as the scanlines are longer than two hexahedra. We can avoid this cache miss if we locally reorder the hexahedra.

Also on more irregular connectivities we get fewer cache misses and better compression rates when we buffer a number of hexahedra from which we then greedily pick "the best" hexahedron and feed it to the compressor. Using a fi ed-size *delay buffer* we tried a number of strategies with an emphasis on speed and simplicity. We briefl describe the simple *spiraling reorderer* that performed well in our experiments and that we use in the results that we report.

We label the active faces of the hexahedron in cache with *front*, *left*, *right*, *top*, and *bottom*. When we pick a new hexahedron among those waiting in the buffer, that has one of its faces in the cache, there are three possibilities for the label L of this face:

- L is *left* or *right*: We set $C(horizontal) = L$ and $D_{last} = horizontal$.

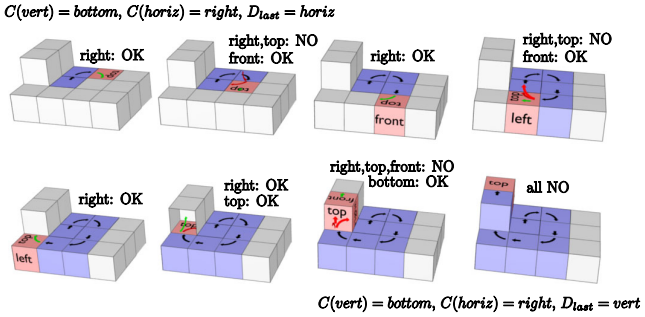

Fig. 6 The *spiralling* reorderer running on an example mesh. The blue, red and white hexahedra are respectively visited, in cache, and waiting in the buffer. A green arrow denotes the firs possible choice in order of priority, red arrows choices that would have had a higher priority but for which there was no corresponding hexahedron in the buffer. We suppose that the starting configuratio is $D_{last} = horizontal$, $C(horizontal) = right$, and $C(vertical) = top$.

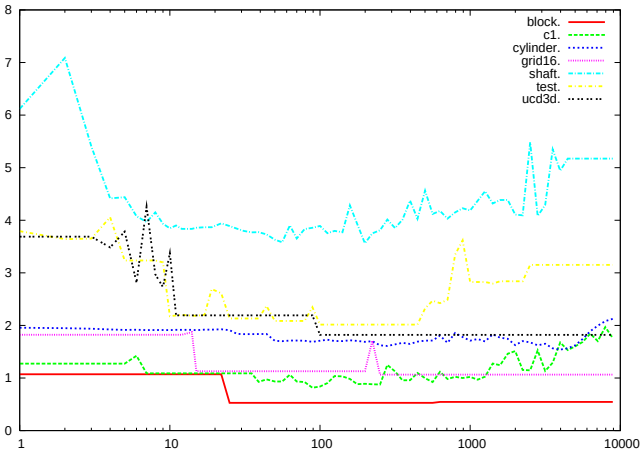

Fig. 7 Connectivity compression (bph) versus delay buffer size for several models. Local reordering is usually very efficient but further increasing the delay shows the limits of greedy strategies.

- L is *left* or *right*: We set $C(vertical) = L$ and $D_{last} = vertical$.
- L is *front*: Nothing changes.

To pick the next hexahedron we consider in decreasing priority among the hexahedra in the buffer:

- the hexahedron that shares the face $C(D_{last})$ with the currently cached hexahedron,
- the hexahedron that shares the face $C(\bar{D}_{last})$, where $\bar{D}_{last}$ is *vertical* if D_{last} is *horizontal*, and *horizontal* else.
- the hexahedron that shares the face *front*,
- the hexahedron that shares one of the the other faces,
- the oldest hexahedron in the buffer.

$C(horizontal)$, $C(vertical)$ and D_{last} are then updated according to this choice. We illustrate this process in Figure 6. This method has the advantage that the added hexahedra closely "stick" to the active surface.

The curves shown in Figure 7 plot the compression rate versus the delay buffer size for several models. Increasing the delay to more than a couple hundred hexahedra usually does not improve the compression rate any further, because the greedy strategy then shows its limits. A global strategy could potentially overcome this drawback, but it would also greatly decrease the speed of the compressor—eventually making it equivalent to a non-streaming compressor.

3.2 Geometry Compression

Each time a new vertex is added (during *START*, *HUT*, *STEP* and *CORNER* operations), we predict its position and code the difference between the predicted and actual value.

We use spectral prediction [5, 10] which assumes that the position of a vertex is determined by the low-frequency components of the Fourier Transform of the mesh element. A vertex is then predicted as a linear combination of the known vertices using weights that zero out the highest possible number of high frequencies. We list the spectral prediction rules we use in Table 1. They work extremely well on hexahedral meshes that tend to be geometrically smooth.

Table 1 Spectral prediction rules. The vertices appear in the order they are predicted, i.e. for any given vertex on line *i* of the table, the vertices of all previous lines are known.

Vertex	Prediction
v_0	last vertex
v_1	v_0
v_2	v_1
v_3	$v_0 + v_2 - v_3$
v_4	v_0
v_7	$v_3 + v_4 - v_0$
v_5	$v_1 + v_4 - v_0$
v_6	$v_0 - v_1 + v_2 - v_3 - v_4 + v_5 + v_7$

Our implementation has two modes of operation: it can either uniformly quantize the vertices prior to compression, perform all predictions in integer arithmetic, and entropy code the resulting integer residuals or it can avoid quantization, perform all predictions in floating-poin arithmetic, and compress the residuals with the method of Isenburg *et al.* [11]. In Table 2 we list our geometry compression rates across our set of test meshes side by side with those of [7] and [15]. For fair comparison we use a newer version of [7] that also uses spectral prediction. The implementation of [15] natively uses spectral prediction.

The degree coder [7] uses more information to predict v_4 in *HUT* configuration by mirroring the adjacent hexahedron. This gives slightly better compression ratios. However, most predictions (typically around 90-95%) are made using the efficien Lorenzo predictor (see Table 1, last row), so this will always remain a very small improvement.

3.3 Cell Data Compression

Ideally we would like to compress cell data with the same strategy that has already proven successful for vertex data. Hence, the prediction would be made using the already pro-

Table 2 Comparing compression performance of our coder with [7], [14], and [15]: We report two connectivity rates, one using a delay buffer of 50 hexahedra and one without delay buffer (in parentheses). For geometry compression, we give rates using 16-bits quantization (Q) and lossless compression (L). We denote with * the meshes that the software provided by Krivograd *et al.* did not handle.

	vertices	hexas	Connectivity (bph)				Geometry (bpv)				Total (bph)			
			[7]	[14]	[15]	Ours	[7]	[15]	Ours		[7]	[15]	Ours	
							(Q)	(L)	(Q)	(L)	(Q)	(L)	(Q)	(L)
block	101,401	93,750	0.07	*	0.07	0.55 (1.00)	0.05	0.2	0.05	2.8	0.1	0.3	0.6	3.6
c1	78,618	71,572	0.59	0.56	1.50	0.94 (1.24)	3.4	14.8	3.2	9.3	4.3	17.8	4.5	10.2
cylinder	500,055	482,900	0.22	0.30	3.01	1.66 (1.91)	0.3	1.8	0.3	1.9	0.5	4.9	2.0	3.6
fru	5,124	4,360	0.97	0.98	3.06	2.20 (2.42)	16.5	55.7	17.4	35.8	20.3	68.5	22.6	44.3
grid	4,096	3,375	0.29	0.4	0.21	1.35 (1.93)	0.4	0.3	0.4	19.5	0.8	0.6	1.8	25.0
hutch	8,790	8,172	0.30	0.48	8.68	2.56 (2.91)	8.5	26.8	7.8	24.9	9.4	37.5	10.9	29.3
mdg-1b	4,510	3,710	0.77	0.93	7.81	2.98 (3.35)	2.8	9.3	2.8	28.7	4.2	19.1	6.4	37.9
shaft	9,218	6,883	1.70	*	18.55	4.04 (5.63)	16.5	42.3	17.4	30.0	23.8	75.2	27.3	44.2
steven	96,030	81,832	0.05	*	1.96	1.04 (1.00)	3.3	14.7	3.7	9.7	3.9	19.2	5.4	12.4
test	3,198	2,386	0.87	1.09	20.40	2.53 (4.02)	4.5	10.8	5.7	34.0	6.9	34.9	10.2	48.1
ucd3d	2,646	2,000	0.47	*	0.30	2.52 (4.50)	1.9	4.67	2.0	29.9	3.0	6.4	5.2	42.1

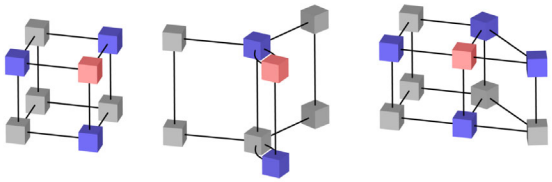

Fig. 8 Examples of possible cell neighborhoods for prediction: (left) a *CORNER* on a grid, (center) the *CORNER* configuratio of Figure 4, and (right) a *GAP* configuratio with irregular neighborhood. For readability, we show the dual of the mesh. Hexahedra are represented as colored cubes (red: new hexahedron, blue: active hexahedra, gray: hexahedra that are in the neighborhood but may no longer be active).

cessed cells of the neighborhood with a spectral predictor on the dual of the mesh. However, two problems arise:

First, the dual of a hexahedral mesh is generally not a hexahedral mesh itself (except in the case of a grid). Thus, the configuratio of the neighborhood can vary a lot between hexahedra as shown in Figure 8. It would be expensive, if not impossible to store the prediction weights for every possible configuratio of the neighborhood. Alternatively the compressor and decompressor could determine the configuratio of the neighborhood and compute the weights with respect to this configuratio on-the-fl . However, computing spectral weights for a typical neighborhood is a costly operation and requires the inversion of a $2n \times 2n$ matrix.

Second, using the complete hexahedron neighborhood requires additional storage: We would have to store the cell data for all processed hexahedra that still have one or more unfinalize vertices (shown as grey and blue cells in Figure 8). This goes against our objective to keep the memory footprint of our algorithm as small as possible.

For these reasons we use a simpler prediction scheme that is faster and more memory efficient We predict cell data using only the hexahedra that are face-adjacent to the current hexahedron (illustrated by blue cells in Figure 8). These hexahedra have at least the corresponding active face (but usually a few more) on the active surface. We simply

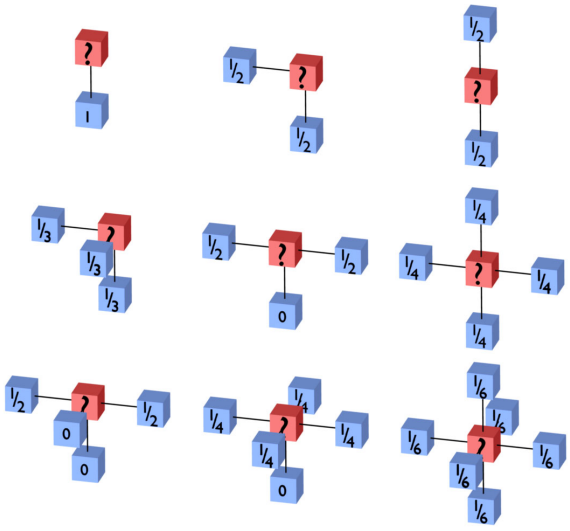

Fig. 9 Cell data prediction weights for the different configurations The red hexahedron is the new cell whose data is to be predicted. Blue hexahedra are the active ones that are used for prediction.

store copies of the cell data of a hexahedron with each of its active face on the active surface. When these faces become inactive and are removed from the active surface the cell data is released along with the face.

Another benefi of this scheme is that there are only the 10 possible neighborhood configuration we already distinguish for connectivity coding. This enables us to precompute the weights and store them in a lookup table (see Figure 9). The weights are obvious for symmetric configura tions: they are all the same and add up to 1. We tried two different approaches to determine the weights for the non-symmetric configuration *BRIDGE*, *GAP* and *PIT*. We computed spectral weights from a $3 \times 3 \times 3$ grid neighborhood, the 3D equivalent of the method presented in [5]. We also computed least squares weights on the "blade" model—they were the always same to a precision of at least 10^{-3}.

Table 3 Cell data compression rates in bits per hexahedron for each of the twelve properties associated with hexahedra in the "cedre" model.

Quantity	12 bits	16 bits	Lossless
K	1.83	4.44	18.1
L	1.99	5.17	11.6
P	2.39	5.35	10.8
T	2.32	5.20	16.0
V_x	2.74	6.05	23.3
V_y	2.68	5.94	23.2
V_z	1.97	4.66	16.4
Y_{CH_4}	1.39	3.47	14.4
Y_{CO_2}	2.28	5.17	15.1
Y_{H_2O}	2.28	5.18	14.9
Y_{N_2}	1.48	3.67	12.2
Y_{O_2}	1.94	4.19	14.7

It may be noted that all the weights are positive. That means that the result of the prediction will always lie within the convex hull of the values of all the cells used for prediction. That usually results in a systematic prediction error, because the predictor is unable to accurately extrapolate, for example in the typical case of a locally monotonous scalar field Because of this, we use a second-order predictor.

Let $c_1,\ldots,c_k$ be the data at known cells of the neighborhood, and $w_1,\ldots,w_k$ the weights used to predict the unknown value c_u. The first-orde prediction of c_u and the prediction residual are:

$$\begin{cases} \bar{c_u} = \sum_{i=1}^{k} w_k.c_k \\ r_u = c_u - \bar{c_u} \end{cases} \quad (1)$$

We simply use the same prediction rule to predict the residual from the residuals of neighbor cells $r_1,\ldots,r_k$, and code the difference r_u^2 between the real and expected residuals:

$$\begin{cases} \bar{r_u} = \sum_{i=1}^{k} w_k.r_k \\ r_u^2 = r_u - \bar{r_u} \end{cases} \quad (2)$$

The decoder can then retrieve the original value as $\bar{c_u} + \bar{r_u} + r_u^2$. Coding the second-order residual r_u^2 instead of the first-orde residual r_u drastically reduces the entropy. For our test models, the second-order scheme halved the entropy of residuals on average for 12 bits quantization.

Table 3 gives the bitrates achieved using our algorithm on the "cedre" dataset. This model comes from a computational flui dynamics combustion simulation with twelve properties attached to the cells (see Figure 10, right).

3.4 Compressing large models

The advantage of streaming over non-streaming compression becomes more and more apparent as models are growing in size. The memory footprint of our compressor remains low even when compressing very large meshes. In Table 4 we compare our memory consumption with that of [7]. For fair comparison we use a version of [7] that has been optimized for speed and reduced memory consumption. The tests were run on an Intel core 2 duo running at 2.66GHz (our implementation uses only one core). We do not compare with [14] as their unoptimized proof-of-concept software is neither speed nor memory efficient

Our streaming compressor greatly outperforms the degree coder [7] with a memory footprint that is orders of magnitude smaller. The non-streaming degree coder is in fact not able to compress the "crank" dataset on our 32-bit operating system because there is not enough RAM to accommodate the compressor's memory needs.

The lossless compressor of Lindstrom and Isenburg [15] was not originally designed for streaming compression. We modifie their source code to enable streaming operation at *compression* (!) time. However, since their format has no mechanism to store finalizatio information in the compressed fil *decompression* will still not be streaming (and have a much much larger memory footprint). The resulting coder is slower than the original one, but has a drastically reduced memory footprint because it no longer accumulates the entire vertex data in memory during compression (but still during decompression). As their connectivity coding scheme is much simpler than our scheme their modifie coder is still faster than ours. It also has a smaller memory footprint (approximately 4 times more compact) since it only maintains the active vertices whereas our coder also keeps track of active faces. However, our compression rate is lower and—more importantly—we can compress cell data.

4 Conclusion

We have described a scheme for streaming compression of hexahedral meshes that scales to very large meshes. Our algorithm is much faster than non-streaming approaches because it interleaves computation and I/O. It also has an orders of magnitude smaller memory footprint because it stores only a small fraction of the whole mesh at any time. Using local reordering in a delay buffer we are able to somewhat mitigate the overhead for compressing the mesh in "stream order" although our connectivity compression rates can be noticable worse on very regular meshes. For most practical applications, however, this has only a minor effect on the total bitrate which is typicaly dominated by the costs for compressing geometry and properties. A representative result is the performance on the 456k hexahedra "blade" mesh: compared to [7] our coder is twice as fast and uses 88 times less memory (only 3.1 MB) while the compressed fil increasing only by about 3% in total size.

We also have shown how to compress cell data with spectral predictions from neighboring cell data maintained on the active surface. Numerical simulations often store a fair number of physical properties with hexahedral cells that can account for a large portion of the storage costs. For the

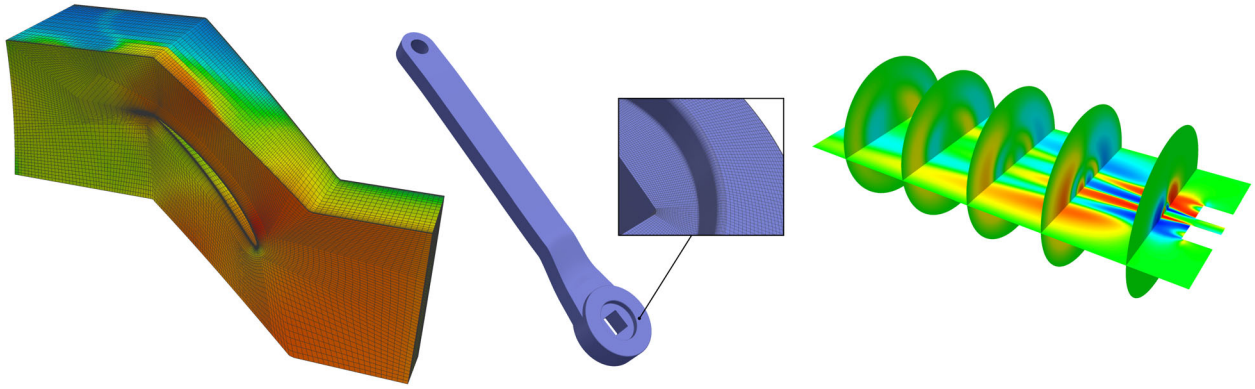

Fig. 10 The “blade” (left), “crank” (center) and “cedre” (right) models. The “blade” and “cedre” models have properties attached to cells.

Table 4 Comparing the performance of our coder with [7] and our modificatio of [15] on large models. For our algorithm we report two results: one without delay buffer and one with a delay of 50 hexahedra (in parentheses). The reported geometry rates are for lossless compression. The given memory footprint is the peak heap usage reported by GNU `memusage`. The timings for [7] do not include reading the fil into memory. The rates for the modifie version of [15] do not accommodate finalizatio (this means that the memory footprint at decompression time will be large).

	Vertices	Hexas	Width	Connectivity (bph)			Geometry (bpv)			Memory (MB)			Time (s)		
			(%)	[7]	[15]	Ours	[7]	[15]	Ours	[7]	[15]	Ours	[7]	[15]	Ours
blade	479k	456k	1.17	0.02	0.85	0.78 (0.45)	16.8	20.8	16.7 (16.8)	273.8	1.8	1.9 (3.1)	2.0	0.6	0.7 (1.0)
crank	2M	2M	0.90		0.98	0.82 (0.46)		12.3	9.1 (9.1)		3.2	7.6 (11.9)		3.0	3.4 (5.3)
crank	49M	48M	0.35		0.36	0.48 (0.28)		10.7	6.3 (6.3)		16.5	62.5 (97.9)		72	95 (142)

“cedre” model, for example, over 50% of the fil stores cell data. It is thus important to be able to efficientl compress these properties and to best of our knowledge the method we present here is the firs to support it. Our cell data prediction scheme can easily be integrated into other compressors [7, 14]. However, it would be difficul to do this for [15] as this algorithm does not maintain a sufficientl large hexahedral neighborhood for efficien spectral predictions. It only provides access to the last hexahedron that (if it happens to be a neighbour) would limit us to simplest spectral prediction rule that is equivalent to delta-coding.

Demo Software: This paper is accompanied by software containing a fully functional streaming compressor (a Windows executable) and a few of our models. You also fin a viewer that can read the compressed format and as a tool to plot layout diagrams. The README.txt fil contains detailed instructions how to run the software.

Prepared by LLNL under Contract DE-AC52-07NA27344.

References

1. Benzley, Perry, Merkley, Clark, Sjaardema (1995) A comparison of all-hexahedral and all-tetrahedral finit element meshes for elastic and elasto-plastic analysis. In: International Meshing Roundtable
2. Blacker (1996) The cooper tool. In: International Meshing Roundtable
3. Guthe, Gumhold, Strasser (1999) Tetrahedral mesh compression with the cut-border machine. In: Visualization
4. Ho, Lee, Kriegman (2001) Compressing large polygonal models. In: Visualization
5. Ibarria, Lindstrom, Rossignac (2007) Spectral interpolation on 3x3 stencils for prediction and compression. Journal of Computers
6. Isenburg (2002) Compressing polygon mesh connectivity with degree duality prediction. In: Graphics Interface
7. Isenburg, Alliez (2002) Compressing hexahedral volume meshes. In: Graphical Models
8. Isenburg, Gumhold (2003) Out-of-core compression for gigantic polygon meshes. In: SIGGRAPH
9. Isenburg, Lindstrom (2005) Streaming meshes. In: Visualization
10. Isenburg, Ivrissimtzis, Gumhold, Seidel (2005) Geometry prediction for high degree polygons. In: Spring Conference on Computer Graphics
11. Isenburg, Lindstrom, Snoeyink (2005) Lossless compression of predicted floating-poin geometry. Computer-Aided Design
12. Isenburg, Lindstrom, Snoeyink (2005) Streaming compression of tetrahedral volume meshes. In: Eurographics Symposium on Geometry Processing
13. Isenburg, Lindstrom, Gumhold, Shewchuk (2006) Streaming compression of tetrahedral volume meshes. In: Graphics Interface
14. Krivograd, Trlep, Zalik (2008) A hexahedral mesh connectivity compression with vertex degrees. Computer-Aided Design
15. Lindstrom, Isenburg (2008) Lossless compression of hexahedral meshes. In: IEEE Data Compression Conference
16. Muller-Hannemann (2001) Shelling hexahedral complexes for mesh generation. Journal of Graph Algorithms and Applications
17. Prat, Gioia, Bertrand, Meneveaux (2005) Connectivity compression in an arbitrary dimension. The Visual Computer
18. Staten, Owen, Blacker (2005) Unconstrained paving and plastering: A new idea for all hexahedral mesh generation. In: International Meshing Roundtable
19. Touma, Gotsman (1998) Triangle mesh compression. In: Graphics Interface