High-speed Marching Cubes using Histogram Pyramids

Christopher Dyken[1,2], Gernot Ziegler[3], Christian Theobalt[4] and Hans-Peter Seidel[3]

[1] Centre of Mathematics for Applications, University of Oslo, Norway

[2] SINTEF ICT Applied Mathematics, Norway

[3] Max-Planck-Institut für Informatik, Germany

[4] Max Planck Center for Visual Computing and Communication, Stanford University, Palo Alto, CA, USA

EARLY DRAFT

Final version to appear in Computer Graphics Forum

Abstract

We present an implementation approach for Marching Cubes on graphics hardware for OpenGL 2.0 or comparable APIs. It currently outperforms all other known GPU-based iso-surface extraction algorithms in direct rendering for sparse or large volumes, even those using the recently introduced geometry shader capabilites. To achieve this, we outfit the HistoPyramid algorithm, previously only used in GPU data compaction, with the capability for arbitrary data expansion. After reformulation of Marching Cubes as a data compaction and expansion process, the HistoPyramid algorithm becomes the core of a highly efficient and interactive Marching Cube implementation. For graphics hardware lacking geometry shaders, such as mobile GPUs, the concept of HistoPyramid data expansion is easily generalized, opening new application domains in mobile visual computing. Further, to serve recent developments, we present how the HistoPyramid can be implemented in the parallel programming language CUDA, by using a novel 1D chunk/layer construction.

1 Introduction

Iso-surfaces of scalar fields defined over cubical grids are essential in a wide range of applications, e.g. medical imaging, geophysical surveying, physics, and computational geometry. A major challenge is that the number of elements grows to the power of three with respect to sample density, and the massive amounts of data puts tough requirements on processing power and memory bandwidth. This is particularly true for applications that require *interactive* visualization of scalar fields. In medical visualization, for example, iso-surface extraction, as depicted in Figure 1, is used on a daily basis. There, the user benefits greatly from immediate feedback in the delicate process of determining transfer functions and iso-levels. In other areas such as geophysical surveys, iso-surfaces are an invaluable tool for interpreting the enormous amounts of measurement data.

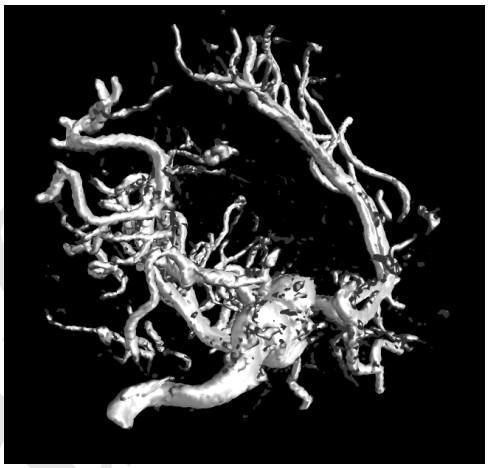


Figure 1: Determining transfer functions and iso-levels for medical data is a delicate process where the user benefits greatly from immediate feedback.

Therefore, and not unexpectedly, there has been a lot of research on volume data processing on Graphics Processing Units (GPUs), since GPUs are particularly designed for huge computational tasks with challenging memory bandwidth requirements, building on simple and massive parallelism instead of the CPU's more sophisticated serial processing. Volume ray-casting is one visualization technique for scalar fields that has been successfully implemented on GPUs. While the intense computation for every change in viewport can nowadays be handled, ray-casting can never produce an explicit representation of the iso-surface. Such an explicit iso-surface is essential for successive processing of the geometry, like volume or surface area calculations, freeform modeling, surface fairing, or surface-

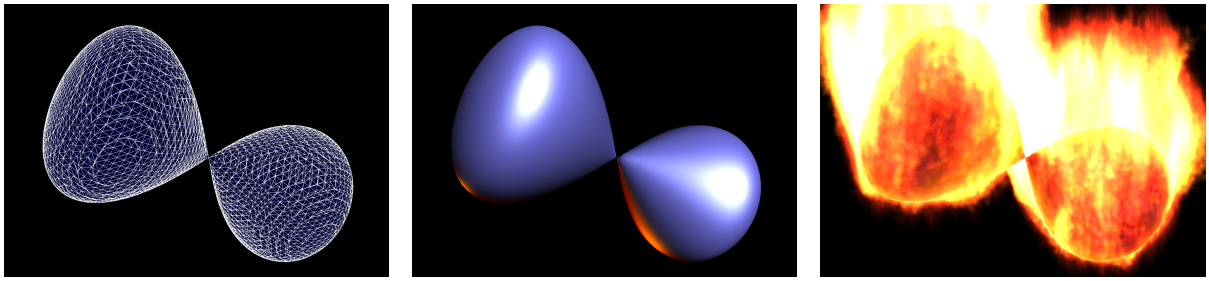


Figure 2: An iso-surface represented explicitly as a compact list of triangles (left) can be visualized from any viewpoint (middle) and even be directly post-processed. One example for such post-processing is the spawning of particles evenly over the surface (right). In all three images, the GPU has autonomously extracted the mesh from the scalar field, where it is kept in graphics memory.

related effects for movies and games, such as the one shown in Figure 2. In particular, two efficient algorithms for extracting explicit iso-surfaces, Marching Cubes (MC) and Marching Tetrahedra (MT), have been introduced. By now, substantial research effort has been spent on accelerating these algorithms on GPUs.

In this paper, we present a novel, though well-founded, formulation of the Marching Cubes algorithm, suitable for any graphics hardware with at least Shader Model 3 (SM3) capabilites. This allows the implementation to run on a wide range of graphics hardware. Our approach extract iso-surfaces directly from raw data without any pre-processing, and thus dynamic datasets, changes in the transfer function, or variations in the iso-level is handled directly. It is able to produce a compact sequence of iso-surface triangles in GPU memory without any transfer of geometry to or from the CPU. The method requires only a moderate implementation effort and can thus be easily integrated into existing applications, while currently outperforms all other known GPU-based iso-surface extraction approaches. For completeness, we also propose how this algorithm can be implemented in the general GPU programming language CUDA [14].

The main element of our approach is the *Histogram Pyramid* [19] (short: *HistoPyramid*), which has shown to be an efficient data structure for GPU stream compaction. In this paper, we have extended the HistoPyramid to handle *general GPU stream expansion*. This simple, yet fundamental modification, together with a *reformulation of the MC algorithm as a stream compaction and expansion process*, enables us to map the MC algorithm onto the GPU.

We begin with an overview of previous and related work in Section 2, followed by a description of HistoPyramids in Section 3. In Section 4, we describe the MC algorithm, its mapping to HistoPyramid stream processing, and implementation details for both the OpenGL and the CUDA implementations. After that, we provide a performance analysis in Section 5, before we conclude in the final section.

2 Previous and Related work

In recent years, iso-surface extraction on stream processors (like GPUs) has been a topic of intensive research. MC is particularly suited for parallelization, as each MC cell can be processed individually. Nevertheless, the number of MC cells is substantial, and some approaches employ pre-processing strategies to avoid processing of empty regions at render time. Unfortunately, this greatly reduces the applicability of the approach to dynamic data. Also, merging the outputs of the MC cells' triangles into one compact sequence is not trivial to parallelize.

Prior to the introduction of geometry shaders, GPUs completely lacked functionality to create primitives directly. Consequently, geometry had to be instantiated by the CPU or be prepared as a vertex buffer object (VBO). Therefore, a fixed number of triangles had to be assumed for each MC cell, and by making the GPU cull degenerate primitives, the superfluous triangles could be discarded. Some approaches used MT to reduce the amount of redundant triangle geometry, since MT never requires more than two triangles per MT tetrahedron. In addition, the configuration of a MT tetrahedron can be determined by inspecting only four corners, reducing the amount of inputs. How-

ever, for a cubical grid, each cube must be subdivided into at least five tetrahedra, which makes the total number of triangles usually larger in total than for an MC-generated mesh. Beyond that, tetrahedral subdivision of cubical grids introduces artifacts [2].

Pascucci et al. [15] represents each MT tetrahedron with a quad and let the vertex shader determine intersections. The input geometry is comprised of triangle strips arranged in a 3D space-filling curve, which minimizes the workload of the vertex shader. The MT approach of Klein et al. [10] renders the geometry into vertex arrays, moving the computations to the fragment shader. Kipfer et al. [9] improved upon this by letting edge intersections be shared. Buatois et al. [1] applied multiple stages and vertex texture lookups to reduce redundant calculations. Some approaches reduce the impact of fixed expansion by using spatial data-structures. Kipfer et al. [9], for example, identify empty regions in their MT approach using an interval tree. Goetz et al. [3] let the CPU classify MC cells, and only feed surface-relevant MC cells to the GPU, an approach also taken by Johansson et al. [7], where a kd-tree is used to cull empty regions. But they also note that this preprocessing on the CPU limits the speed of the algorithm. The geometry shader (GS) stage of SM4 hardware can produce and discard geometry on the fly. Uralsky [17] propose a GS-based MT approach for cubical grids, splitting each cube into six tetrahedra. An implementation is provided in the Nvidia OpenGL SDK-10, and has also been included in the performance analysis.

Most methods could provide a copy of the iso-surface in GPU memory, using either vertex buffers or the new transform feedback mechanism of SM4-hardware. However, except for GS-based approaches, the copy would be littered with degenerate geometry, so additional post-processing, such as stream compaction, would be needed to produce a *compact* sequence of triangles.

Horn's early approach [6] to GPU-based stream compaction uses a prefix sum method to generate output offsets for each input element. Then, for each output element, the corresponding input element is gathered using binary search. The approach has a complexity of O($n \log n$) and does not perform well on large datasets.

Prefix Sum (Scan) uses a pyramid-like up-sweep and down-sweep phase, where it creates, in parallel, a table that associates each input element with output offsets. Then, using scattering, the GPU can iterate over input elements and directly store the output using this offset table. Harris [4] designed an efficient implementation in CUDA. The Nvidia CUDA SDK 1.1 provides an MC implementation using Scan, and we have included a highly optimized version in the performance analysis.

Ziegler et al. [19] have proposed another approach to data compaction. With the introduction of HistoPyramids, data compaction can be run on the GPU of SM3 hardware. Despite a deep gather process for the output elements, the algorithm is surprisingly fast when extracting a small subset of the input data.

3 HistoPyramids

The core component of our MC implementation is the HistoPyramid data structure, introduced in [19] for GPU-based data compaction. We extend its definition here, and introduce the concept of local key indices (below) to provide for GPU-based 1:m expansion of data stream elements for all non-negative m. The input is a stream of data input elements, short: the *input stream*. Now, each input element may *allocate* a given number of elements in the output stream. If an input element allocates zero elements in the output stream, the input element is discarded and the output stream becomes smaller (data compaction). On the other hand, if the input element allocates more than one output element, the stream is expanded (data expansion). The input elements' individual allocation is determined by a user-supplied *predicate function* which determines the output multiplicity for each input element. As a sidenote, in [19], each element allocated exactly one output or none.

The HistoPyramid algorithm consists of two distinct phases. In the first phase, we create a HistoPyramid, a pyramid-like data structure very similar to a MipMap. In the second phase, we extract the output elements by traversing the HistoPyramid top-down to find the corresponding input elements. In the case of stream expansion, we also determine which numbered copy of the input element we are currently generating.

3.1 Construction

The first step is to build the HistoPyramid, a stack of 2D textures. At each level, the texture size is a quarter of the size of the level below, i.e. the same layout as the MipMap pyramid of a 2D texture. We call the largest texture, in the bottom of the stack, the *base texture*, and the single texel of the 1×1 texture in the top of the stack

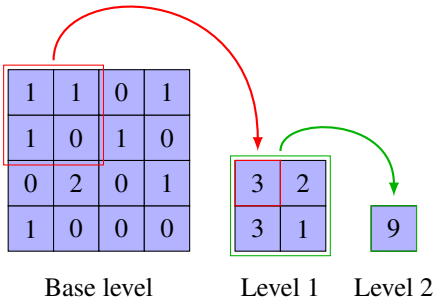


Figure 3: Bottom-up build process of the HistoPyramid, adding the values of four texels repeatedly. The top texel contains the total number of output elements in the pyramid.

the *top element*. Figure 3 shows the levels of a HistoPyramid, laid out from left to right. The texel count in the base texture is the maximum number of input elements the HistoPyramid can handle. For simplicity, we assume that the base texture is square and the side length a power of two (arbitrary sizes can be accommodated with suitable padding).

In the base level, each input element corresponds to one texel. This texel holds the number of allocated output elements. In Figure 3 we have an input stream of 16 elements, laid out from left to right and top to bottom. Thus, elements number 0,1,3,4,6,11, and 12 have allocated one output element each (stream pass-through). Element number 9 has allocated two output elements (stream expansion), while the rest of the elements have not allocated anything (stream compaction). These elements will be discarded. The number of elements to be allocated is determined by the predicate function at the base level. This predicate function may also map the dimension of the input stream to the 2D layout of the base level. In our MC application, the input stream is a 3D volume.

The next step is to build the rest of the levels from bottom-up, level by level. According to the MipMap principle, each texel in a level corresponds to four texels in the level below. In contrast to the averaging used in the construction of MipMaps, we *sum* the four elements. Thus, each texel receives the sum of the four corresponding elements in the level below. The example in Figure 3 illustrates this process. The sum of the texels in the 2×2 block in the upper left of the base level is three, and stored in the upper left texel of Level 1. The

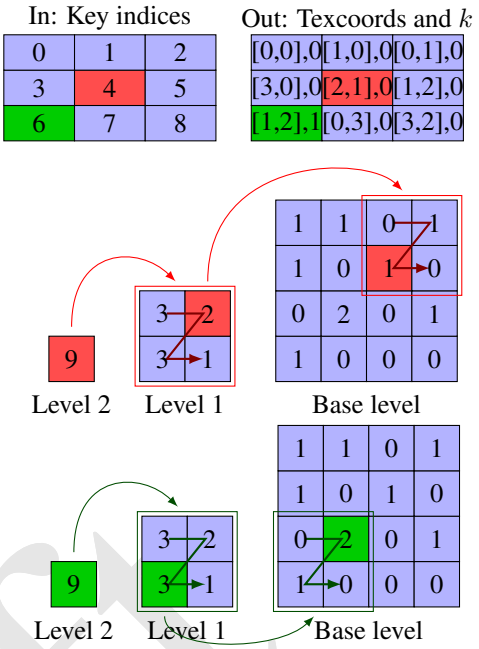


Figure 4: Element extraction, interpreting partial sums as interval in top-down traversal. Red traces the extraction of key index 4 and green traces key index 6.

sum of the texels in the single 2×2 block of Level 1 is nine, and stored in the single texel of Level 2, the top element of the HistoPyramid.

At each level, the computation of a texel depends only on four texels from the previous one. This allows us to compute all texels in one level in parallel, without any data inter-dependencies.

3.2 Traversal

In the second phase, we generate the output stream. The number of output elements is provided by the top element of the HistoPyramid. Now, to fill the output stream, we traverse the HistoPyramid once per output element. To do this, we linearly enumerate the output elements with a *key index* k, and *re-interpret HP values as key index intervals*. The traversal requires several variables: We let m denote the number of HP levels. The traversal maintains a texture coordinate $\mathbf{p}$ and a current level l, referring to one specific texel in the HistoPyramid. We further maintain a local key index k_l, which adapts to the local index range. It is initialized as $k_l = k$. The traversal starts from the top level $l = m$

and goes recursively down, terminating at the base level $l = 0$. During traversal, k_l and $\mathbf{p}$ are continuously updated, and when traversal terminates, $\mathbf{p}$ points to a texel in the base level. In the case of stream pass-through, k_l is always zero when traversal terminates. However, in the case of stream expansion, the value in k_l determines which numbered copy of the input element this particular output element is.

Initially, $l = m$ and $\mathbf{p}$ points to the center of the single texel in the top level. We subtract one from l, descending one step in the HistoPyramid, and now $\mathbf{p}$ refers to the center of the 2×2 block of texels in level $m - 1$ corresponding to the single texel $\mathbf{p}$ pointed to at level m. We label these four texels in the following manner,

$$\begin{matrix} a & b \\ c & d \end{matrix}$$

and use the values of these texels to form the four ranges A,B,C, and D, defined as

$$\begin{aligned} A &= [0\ ,\ a), \\ B &= [a\ ,\ a+b), \\ C &= [a+b\ ,\ a+b+c), \quad \text{and} \\ D &= [a+b+c\ ,\ a+b+c+d). \end{aligned}$$

Then, we examine which of the four ranges k_l falls into. If, for example, k_l falls into the range B, we adjust $\mathbf{p}$ to point to the center of b and subtract the start of the range, in this case a, from k_l, adapting k_l to the local index range. We recurse by subtracting one from l and repeating the process until $l = 0$, when the traversal terminates. Then, from $\mathbf{p}$ we can calculate the index of the corresponding input stream element, and the value in k_l enumerates the copy.

Figure 4 show two examples of HistoPyramid traversal. The first example, labeled red, is for the key index $k = 4$, a stream pass-through. We start at level 2 and descend to level 1. The four texels at level 1 form the ranges

$$A = [0\ ,\ 3), \quad B = [3\ ,\ 5), \quad C = [5\ ,\ 8), \quad D = [8\ ,\ 9).$$

We see that k_l is in the range B. Thus, we adjust the texture coordinate to point to the upper left texel and adjust k_l to the new index range by subtracting 3 from k_l, which leaves $k_l = 1$. Then, we descend again to the base level. The four texels in the base level corresponding to the upper left texel of level 1 form the ranges

$$A = [0\ ,\ 0), \quad B = [0\ ,\ 1), \quad C = [1\ ,\ 2), \quad D = [2\ ,\ 2).$$

The ranges A and D are empty. Here, $k_l = 1$ falls into C, and we adjust $\mathbf{p}$ and k_l accordingly. Since we're at the base level, the traversal terminates, $\mathbf{p} = [2, 1]$ and $k_l = 0$.

The second example of Figure 4, labeled green, is a case of stream expansion, with key index $k = 6$. We begin at the top of the HistoPyramid and descend to level 2. Again, the four texels form the ranges

$$A = [0\ ,\ 3), \quad B = [3\ ,\ 5), \quad C = [5\ ,\ 8), \quad D = [8\ ,\ 9),$$

and k_l falls into the range C. We adjust $\mathbf{p}$ to point to c and subtract the start of range C from k_l, resulting in the new local key index $k_l = 1$. Descending, we inspect the four texels in the lower left corner of the base level, which form the four ranges

$$A = [0\ ,\ 0), \quad B = [0\ ,\ 2), \quad C = [2\ ,\ 3), \quad D = [3\ ,\ 3),$$

where k_l now falls into range B, and we adjust $\mathbf{p}$ and k_l accordingly. Since we're at the base level, we terminate traversal, and have $\mathbf{p} = [1, 2]$. $k_l = 1$ implies that output element 6 is the second copy of the input element from position $[1, 2]$ in the base texture.

The traversal only reads from the HistoPyramid. There are no data dependencies between traversals. Therefore, the output elements can be extracted in any order — even in parallel.

3.3 Comments

The 2D texture layout of the HistoPyramid fits graphics hardware very well. It can intuitively be seen that in the domain of normalized texture coordinate calculations, the texture fetches overlap with fetches from the level below. This allows the 2D texture cache to assist HP traversal with memory prefetches, and thus increases its performance.

At each descent during traversal, we have to inspect the values of four texels, which amounts to four texture fetches. However, since we always fetch 2×2 blocks, we can use a four-channel texture and encode these four values as RGBA value. This halves the size of all textures along both dimensions, and thus let us build four times larger HistoPyramids within the same texture size limits. In addition, since we quarter the number of texture fetches, and graphics hardware is quite efficient at fetching four-channel RGBA values, this usually yields a speed-up. For more details, see *vec4-HistoPyramids* in [19].

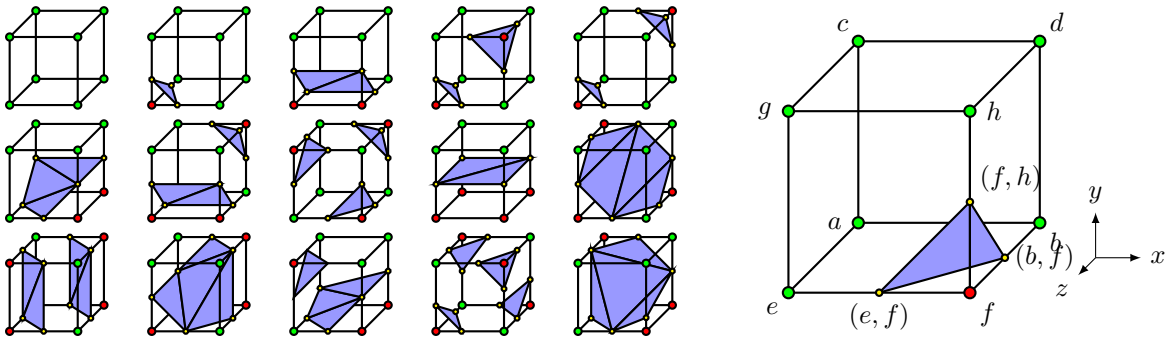


Figure 5: The 15 basic predefined triangulations [11] for edge intersections (left). By symmetry, they provide triangulations for all 256 MC cases. Ambiguous cases are handled by adding some extra triangulations [12]. A MC cell (right) where only f is inside the iso-surface, and thus the edges (e, f), (b, f), and (f, h) intersect the iso-surface.

Memory requirements of the HP are identical to a 2D MipMap-pyramid, i.e. 1/3 the size of the base level. Since the lower levels contain only small values, one could create a composite structure using UINT8 for the lowest levels, UINT16 for some levels before using UINT32 for the top levels.

4 Marching Cubes

The Marching Cubes (MC) algorithm [11] of Lorensen and Cline is probably the most commonly used algorithm for extracting iso-surfaces from scalar fields, which is why we chose it as basis for our GPU iso-surface extraction. From a 3D grid of $N \times M \times L$ scalar values, we form a grid of $(N-1) \times (M-1) \times (L-1)$ cube-shaped "MC cells" in-between the scalar values such that each corner of the cube corresponds to a scalar value. The basic idea is to "march" through all the cells one-by-one, and for each cell, produce a set of triangles that approximates the iso-surface locally in that particular cell.

It is assumed that the topology of the iso-surface inside a MC cell can be completely determined from classifying the eight corners of the MC cell as *inside* or *outside* the iso-surface. Thus, the topology of the local iso-surface can be encoded into an eight-bit integer, which we call the *MC case* of the MC cell. If any of the twelve edges of the MC cell have one endpoint inside and one outside, the edge is said to be *piercing* the iso-surface. The set of piercing edges is completely determined by the MC case of the cell. E.g., the MC cell right in Figure 5 has corner f inside and the rest of the corners outside. Encoding "inside" with 1 and "outside" with 0, we attain the MC case $\%00000100$ in binary notation, or 32 in decimal. The three piercing edges of the MC cell are (b, f), (e, f), and (f, h).

For each piercing edge we determine the *intersection point* where the edge intersects the iso-surface. By triangulating these intersection points we attain an approximation of the iso-surface inside the MC cell, and with some care, the triangles of two adjacent MC cells fit together. Since the intersection points only move along the piercing edges, there are essentially 256 possible triangulations, one for each MC case. From 15 basic predefined triangulations, depicted left in Figure 5, we can create triangulations for all 256 MC cases due to inherent symmetries [11]. However, some of the MC cases are ambiguous, which may result in a discontinuous surface. Luckily, this is easily remedied by modifying the triangulations for some of the MC cases [12]. On the downside, this also increases the maximum number of triangles emitted per MC cell from 4 to 5.

Where a piercing edge intersects the iso-surface is determined by the scalar field along the edge. However, the scalar field is only known at the end-points of the edge, so some assumptions must be made. A simple approach is to position the intersection at the midpoint of the edge, however, this choice leads to an excessively "blocky" appearance, see the left side of Figure 6. A better choice is to approximate the scalar field along the edge with an interpolating linear polynomial, and find the intersection using this approximation, as shown in the right half of Figure 6.

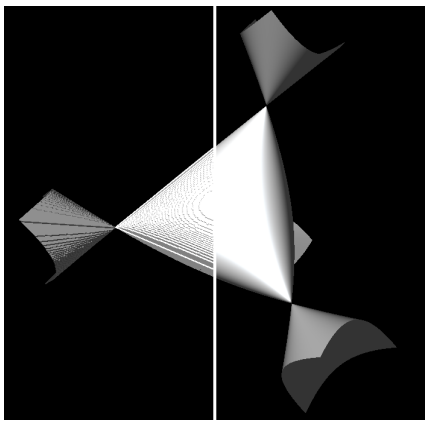


Figure 6: Assuming that edges pierce the iso-surface at the middle of an edge (left) and using an approximating linear polynomial to determine the intersection (right).

4.1 Mapping MC to Stream and HistoPyramid Processing

Our approach is to implement MC as a sequence of data stream operations, with the input data stream being the cells of the 3D scalar field, and the output stream being a set of vertices, forming the triangles of the iso-surface. The data stream operations are executed via the HistoPyramid or, in one variant, the geometry shader, which compact and expand the data stream as necessary.

Figure 7 shows a flowchart of our algorithm. We use a texture to represent the 3D scalar field, and the first step of our algorithm is to update this field. The 3D scalar field can stem from a variety of sources: it may e.g. originate from disk storage, CPU memory, or simply be the result of GPGPU computations. For static scalar fields, this update is of course only needed once.

The next step is to build the HistoPyramid. We start at the base level. Our predicate function corresponds the base level texels with one MC cell each, and calculates the corresponding 3D scalar field coordinates. Then, it samples the scalar field to classify the MC cell corners. By comparing against the iso-level, it can determine which MC cell corners are inside or outside the iso-surface. This determines the MC case of the cell, and thus the number of vertices needed to triangulate this case. We store this value in the base level, and can now proceed with HistoPyramid build-up for the rest of the levels, as described in Section 3.1.

After HistoPyramid buildup has been completed, we read back the single texel at its top level. This makes the CPU aware of the actual number of vertices required for a complete iso-surface mesh. Dividing this number by three yields the number of triangles.

As already mentioned, output elements can be extracted by traversing the HistoPyramid. Therefore, the render pass is fed with dummy vertices, enumerated with increasing key indices. For each input vertex, we use its key index to conduct a HistoPyramid traversal, as described in Section 3.2. After the traversal, we have a texel position in the base level and a key index remainder k_l. From the texel position in the base texture, we can determine the corresponding 3D coordinate, inverting the predicate function's 3D to 2D mapping. Using the MC case of the cell and the local key index k_l, we can perform a lookup in the *triangulation table texture*, a 16×256 table where entry (i, j) tells which edge vertex i in a cell of MC case j corresponds to. Then, we sample the scalar field at the two end-points of the edge, determine a linear interpolant of the scalar field along this edge, find the exact intersection, and emit the corresponding vertex.

In effect, the algorithm has transformed the stream of 3D scalar field values into a stream of vertices, generated on the fly while rendering iso-surface geometry. Still, the geometry can be stored in a buffer on the GPU if so needed, either by using transform feedback buffers or via a render-to-vertex-buffer pass.

4.2 Implementation Details

In detail, the actual implementation of our MC approach contains some noteworthy caveats described in this chapter.

We store the 3D scalar field using a large tiled 2D texture, know as a Flat 3D layout [5], which allows the scalar field to be updated using a single GPGPU-pass. Since the HistoPyramid algorithm performs better for large amounts of data, we use the same layout for the base level of the HistoPyramid, allowing the entire volume to be processed using one HistoPyramid.

We use a four-channel HistoPyramid, where the RGBA-values of each base level texel correspond to the analysis of a tiny $2 \times 2 \times 1$-chunk of MC cells. The analysis begins by fetching the scalar values at the common $3 \times 3 \times 2$ corners of the four MC cells. We compare these values to the iso-value to determine the inside/outside state of the corners, and from this determine the actual MC cases of the MC cells. The MC case corresponds to the MC template geometry set forth by the Marching

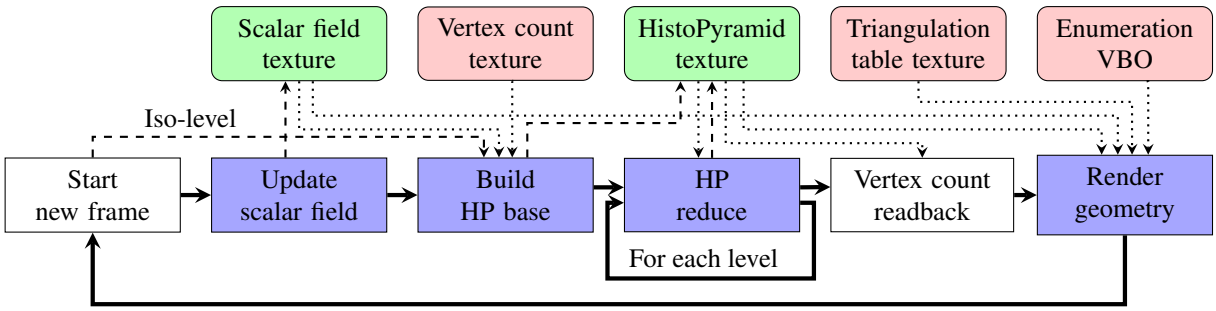


Figure 7: A schematic view of the implementation. Thick arrows designate control flow, with blue boxes executing on the GPU and white boxes on the CPU. The dotted and dashed arrows represent data flow, with dotted arrows for fetches and dashed arrows for writes. Green boxes stand for dynamic data. Red boxes for static data.

Cubes algorithm. As it is needed in the extraction process, we use some of the bits in the base level texels to cache it. To do this, we let the vertex count be the integer part and the MC case the fractional part of a single float32 value. This is sound, as the maximum number of vertices needed by an MC case is 15, and therefore the vertex count only needs 4 of the 32 bits in a float32 value. This data co-sharing is only of relevance in the base-level, and the fractional part is stripped away when building the first level of the HistoPyramid. HistoPyramid texture building is implemented as consecutive GPGPU-passes of reduction operations, as exemplified in "render-to-texture loop with custom MipMap generation" [8], but instead of using one single framebuffer object (FBO) for all MipMap levels, we use a separate FBO for each MipMap level, yielding a speedup on some hardware. We retrieve the RGBA-value of the top element of the HistoPyramid to the CPU as the sum of these four values is the number of vertices in the iso-surface.

Our SM3 variant uses the vertex shader to generate the iso-surface on the fly. Here, rendering is triggered by feeding the given number of (dummy) vertices to the vertex shader. The only vertex attribute provided by the CPU is a sequence of key indices, streamed off a static vertex buffer object (VBO). Even though SM4-hardware provides the `gl_VertexID`-attribute, OpenGL cannot initiate vertex processing without any attribute data, and hence a VBO is needed anyway. For each vertex, the vertex shader uses the provided key index to traverse the HistoPyramid, determining which MC cell and which of its edges this vertex is part of. It then samples the scalar field at both end-points of its edge, and uses its linear approximation to intersect with the edge. The shader can also find an approximate normal vector at this vertex, which it does by interpolating the forward differences of the 3D scalar field at the edge end-points.

Our SM4 variant of iso-surface extraction lets the geometry shader generate the vertices required for each MC cell. Here, the HistoPyramid is only used for data stream compaction, discarding MC cells that do not intersect with the iso-surface. To this purpose, we modified the predicate function to fill the HP base level with keep (1) or discard (0) values, since no output cloning is necessary for vertex generation. After retrieving the number of geometry-producing MC cells from the top level of the HistoPyramid, the CPU triggers the geometry shader by rendering one point primitive per geometry-producing MC cell. For each primitive, the geometry shader first traverses the HistoPyramid and determines which MC cell this invocation corresponds to. Then, based on the stored MC case, it emits the required vertices and, optionally, their normals by iterating through the triangulation table texture. This way, the SM4 variant reduces the number of HistoPyramid traversals from once for every vertex of each iso-surface triangle, to once for every geometry-relevant MC cell.

If the iso-surface is required for post-processing, the geometry can be recorded directly as a compact list of triangles in GPU memory using either the new transform feedback extension or a more traditional render-to-texture setup.

Algorithmically, there is no reason to handle the complete volume in one go, except for the moderate

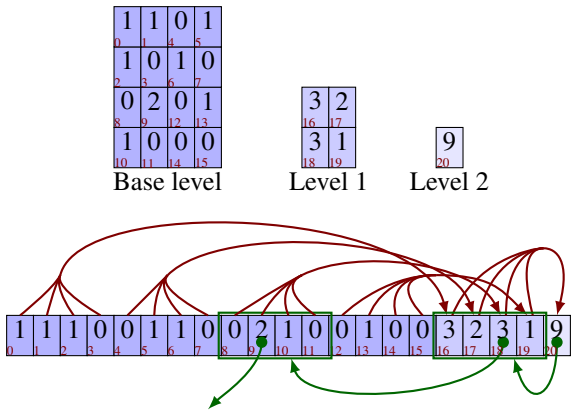


Figure 8: HistoPyramids in CUDA: A chunk. Serialization of the HistoPyramid cells, aka Morton code, is shown in the cells' lower left. If unknown, it can be constructed via the value one at each cell in the base level. Using this layout, four numbers that form intervals lie consecutively in memory. Red arrows (above) show construction of the linearized HistoPyramid, while the green arrows (below) show extraction of key index 6, as exemplified in Figure 4 and explained in Section 3.2.

performance increase at large volume sizes which is typical for HistoPyramids. Hence, the volume could also be tiled into suitable chunks, making the memory impact of the HP small.

4.3 CUDA Implementation

Even though our method can be implemented using standard OpenGL 2.0, we have noticed increased interest in performance behavior under the GPGPU programming language CUDA. In the following section, we describe some additional insights from porting our algorithm to CUDA. A thorough introduction to CUDA itself can be found in [14].

At the core of our algorithm lies the HistoPyramid, a data structure based on 2D textures. Unfortunately, in the current release of CUDA, kernels cannot output data to a 2D texture without an intermediate device-device copy. Instead, the kernels write output to linear memory, which can in its turn be bound directly to a 1D sampler. Therefore, we linearize the 2D structure using Morton-code, which preserves locality very well. The Morton-code of a point is determined by interleaving the bits of the coordinate values. Figure 8 shows a HistoPyramid with the Morton-code in the lower left corners of the elements.To improve the locality between MipMap-levels, we use *chunks*, HistoPyramids which are so small that all their levels remain close in memory. These chunks are then organized into *layers*, where the top level of the chunks in one layer forms the base of the chunks in the next layer. One example: using chunks with 64 base layer elements, we use one layer to handle 3 levels of a comparable 2D HistoPyramid.

Using this layout, we can link our data structures closely to the CUDA computation concepts *grids, blocks and threads*. Each layer of chunks is built by one grid. Inside the layer, each chunk is built with one block, using one thread per chunk base level element. The block's threads store the chunk base layer in global memory, but keep a copy in shared memory. Then, the first quarter of the threads continue, building the next level of the chunk from shared memory, again storing it in global mem, with a copy in shared mem and so on. Four consecutive elements are summed to form an element in the next level, as shown by the red arrows in Figure 8. HP Chunk/Layer Traversal is largely analogous to 2D texture-based traversal, as shown by the green arrows in Figure 8. In addition, for each chunk traversed, we must jump to the corresponding chunk in the layer below. Data extraction based on this traversal can be carried out in CUDA, by letting CUDA populate a VBO. Alternately, by letting CUDA store the layers of chunks in an OpenGL buffer object, HP Chunk/Layer structures can be traversed in an OpenGL vertex shader.

In effect, we have transformed the HistoPyramid 2D data structure into a novel 1D HistoPyramid layer/chunk-structure. In principle, the memory requirement of the layer/chunk-structure is one third of the base layer size, just like for 2D MipMaps. But since empty chunks are never traversed, they can even be completely omitted. This way, the size of the input data needs only be padded up to the number of base elements in a chunk, which further reduces memory requirements. Furthermore, all layers do not need chunks with full 32-bit values. MC produces maximally 15 vertices per cell, which allows us to use 8-bit chunks with 16 base level elements in the first layer, and a layer with 16-bit chunks with 256 base level elements, before we have to start using 32-bit chunks. Thus, the flexibility of the layer/chunk-structure makes it easier to handle large datasets, very probably even out-of-core data.

We had good results using 64 elements in the chunk base layer. With this chunk size, a set of 256^3 elements can be reduced using four layers. Since the chunks' cells are closely located in linear memory, we have im-

proved 1D cache assistance — both in theory and practice.

5 Performance Analysis

We have performed a series of performance benchmarks on six iso-surface extraction methods. Four are of our own design: the OpenGL-based HistoPyramid with extraction in the vertex shader (GLHP-VS) or extraction in the geometry shader (GLHP-GS), and the CUDA-based HistoPyramid with extraction into a VBO using CUDA (CUHP-CU), and extraction directly in the OpenGL vertex shader (CUHP-VS). In addition, we have benchmarked the MT-implementation [17] from the Nvidia OpenGL SDK-10 (NV-SDK10), where the geometry shader is used for compaction and expansion. For the purpose of this performance analysis, we obtained a highly optimized version of the Scan [4]-based MC-implementation provided in the Nvidia CUDA 1.1 SDK. This optimized version (CUDA1.1+) is up to three times faster than the original version from the SDK, which reinforces that intimate knowledge of the hardware is of advantage for CUDA application development.

To measure the performance of the algorithms under various loads, we extracted iso-surfaces out of six different datasets, at four different resolutions. The iso-surfaces are depicted in Figure 9. The first three volumes, "Bunny", "CThead", and "MRbrain", were obtained from the Stanford volume data archive [16], the "Bonsai" and "Aneurism" volumes were obtained from volvis.org [18]. The analytical "Cayley" surface is the zero set of the function $f(x, y, z) = 16xyz + 4(x + y + z) - 1$ sampled over $[-1, 1]^3$. While the algorithm is perfectly capable of handling dynamic volumes without modification, we have kept the scalar field and iso-level static to get consistent statistics, however, the full pipeline is run every frame.

Table 5 shows the performance of the algorithms, given in *million MC-cells processed per second*, capturing the throughput of each algorithm. In addition, the *frames per second*, given in parentheses, captures the interactiveness on current graphics hardware. Since the computations per MC cell vary, we recorded the percentage of MC cells that produce geometry. This is because processing of a MC cell that intersects the iso-surface is heavier than for MC cells that do not intersect. On average, each intersecting MC cell produces roughly two triangles.

All tests were carried out on a single workstation with an Intel Core2 2.13 GHz CPU and 1 GB RAM, with four different Nvidia GeForce graphics cards: A 128MB 6600GT, a 256MB 7800GT, a 512MB 8800GT-640, and a 768MB 8800GTX. Table 5 shows the results for the 7800GT and the 8800GTX, representing the SM3.0 and SM4.0 generations of graphics hardware. All tests were carried out under Linux, running the 169.04 Nvidia OpenGL display driver, except the test with NVSDK10, which was carried out on MS Windows under the 158.22 OpenGL display driver.

Evaluation shows that the HistoPyramid algorithms benefit considerably from increasing amounts of volume data. This meets our expectations, since the HistoPyramid is particularly suited for sparse input data, and in large volume datasets, large amounts of data can be culled early in the MC algorithm. However, some increase in throughput is also likely to be caused by the fact that larger chunks of data give increased possibility of data-parallelism, and require fewer GPU state-changes (shader setup, etc.) in relation to the data processed. This probably explains the (moderate) increase in performance for the NV-SDK10.

The 6600GT performs quite consistently at half the speed of the 7800GT, indicating that HistoPyramid buildup speeds are highly dependent on memory bandwidth, as the 7800GT has twice the memory bandwidth of the 6600GT. The 8800GT performs at around 90–100% the speed of the 8800GTX, which is slightly faster than expected, given it only has 70% of the memory bandwidth. This might be explained by the architecture improvements carried out along with the improved fabrication process that differentiates the GT from the GTX. However, the HP-VS algorithm on the 8800GTX is 10–30 times faster than on the 7800GT, peaking at over 1000 million MC cells processed per second. This difference cannot be explained by larger caches and improved memory bandwidth alone, and shows the benefits of the unified Nvidia 8 architecture, enabling radically higher performance in the vertex shader-intensive extraction phase.

The CUDA implementations are not quite as efficient as the GLHP-VS, running at only 70–80% of its speed. However, if geometry must not only be rendered but also stored (GLHP-VS uses transform feedback in this case), the picture changes. There, CUHP-CU is at least as fast as GLHP-VS, and up to 60% faster for dense datasets than our reference. CUHP-VS using transform feedback, however, is consistently slower than the GLHP-VS with transform feedback, indicating that the

	Model	MC cells	Density	7800GT GLHP-VS	8800GTX GLHP-VS	8800GTX GLHP-GS	8800GTX CUHP-CU	8800GTX CUHP-VS	8800GTX NVSDK10	8800GTX CUDA1.1+
Bunny	255x255x255	16581375	3.18%	—	540 (33)	82 (5.0)	414 (25)	420 (25)	—	400 (24)
	127x127x127	2048383	5.64%	12 (5.7)	295 (144)	45 (22)	250 (122)	248 (121)	—	246 (120)
	63x63x63	250047	9.07%	8.5 (34)	133 (530)	27 (108)	94 (378)	119 (474)	28 (113)	109 (436)
	31x31x31	29791	13.57%	5.0 (167)	22 (722)	12 (399)	17 (569)	24 (805)	22 (734)	22 (739)
CTHead	255x255x127	8258175	3.73%	16 (2.0)	434 (53)	68 (8.2)	372 (45)	366 (44)	—	358 (43)
	127x127x63	1016127	6.25%	12 (11)	265 (260)	40 (40)	200 (197)	200 (196)	—	217 (213)
	63x63x31	123039	9.62%	7.6 (62)	82 (669)	23 (189)	58 (473)	76 (615)	25 (206)	70 (571)
	31x31x15	14415	14.46%	4.5 (310)	11 (768)	8 (566)	8.6 (599)	12 (857)	17 (1187)	12 (802)
MRBrain	255x255x127	8258175	5.87%	10 (1.3)	305 (37)	38 (4.6)	269 (33)	274 (33)	—	279 (34)
	127x127x63	1016127	7.35%	9.8 (9.7)	239 (235)	32 (31)	184 (181)	183 (180)	—	112 (194)
	63x63x31	123039	9.96%	7.4 (60)	82 (663)	21 (169)	57 (466)	75 (611)	26 (215)	70 (566)
	31x31x15	14415	14.91%	4.3 (302)	11 (771)	8 (546)	8.5 (589)	12 (837)	18 (1257)	12 (795)
Bonsai	255x255x255	16581375	3.04%	—	562 (34)	82 (4.9)	427 (26)	433 (26)	—	407 (25)
	127x127x127	2048383	5.07%	13 (6.3)	314 (153)	45 (22)	264 (129)	262 (128)	—	269 (131)
	63x63x63	250047	6.69%	11 (45)	148 (590)	32 (127)	103 (413)	132 (526)	29 (116)	119 (476)
	31x31x31	29791	8.17%	8.0 (268)	21 (717)	16 (529)	17 (578)	25 (827)	24 (805)	23 (783)
Aneurism	255x255x255	16581375	1.60%	—	905 (55)	134 (8.1)	605 (37)	598 (36)	—	510 (31)
	127x127x127	2048383	2.11%	29 (14)	520 (254)	98 (48)	396 (193)	427 (209)	—	372 (182)
	63x63x63	250047	3.70%	19 (77)	169 (676)	50 (199)	116 (464)	152 (607)	33 (132)	136 (544)
	31x31x31	29791	6.80%	8.7 (292)	21 (715)	16 (545)	17 (584)	25 (830)	26 (857)	24 (789)
Cayley	255x255x255	16581375	0.93%	—	1135 (68)	245 (15)	695 (42)	700 (42)	—	563 (34)
	127x127x127	2048383	1.89%	31 (15)	534 (261)	118 (58)	405 (198)	438 (214)	—	377 (184)
	63x63x63	250047	3.87%	18 (72)	174 (695)	52 (206)	116 (465)	151 (606)	32 (129)	133 (530)
	31x31x31	29791	8.10%	7.3 (246)	22 (736)	17 (574)	18 (589)	25 (828)	25 (828)	23 (774)

Table 1: The performance of extraction and rendering of iso-surfaces, measured in million MC cells processed per second, with frames per second given in parentheses. The implementations are described in Section 5.

1D chunk/layer-layout is not as optimal as the MipMap-like 2D layout.

The geometry shader approach, GLHP-GS, has the theoretical advantage of reducing the number of HistoPyramid traversals to roughly one sixth of the vertex shader traversal in GLHP-VS. Surprisingly, in practice we observed a throughput that is four to eight times less than for GLHP-VS, implying that the data amplification rate of the geometry shader cannot compete with the HistoPyramid, at least not in this application. It seems as if the overhead of this additional GPU pipeline stage is still considerably larger than the partially redundant HistoPyramid traversals. Similarly, the NVSDK10-approach shows a relatively mediocre performance compared to GLPHP-VS. But this picture is likely to change with improved geometry shaders of future hardware generations.

The CUDA1.1 approach uses two successive passes of scan. The first pass is a pure stream compaction pass, culling all MC cells that will not produce geometry. The second pass expands the stream of remaining MC cells. The advantage of this two-pass approach is that it enables direct iteration over the geometry-producing voxels, and this avoids a lot of redundant fetches from the scalar field and calculations of edge intersections. The geometry-producing voxels are processed homogenously until the final step where the output geometry is built using scatter write. This approach has approximately the same performance as our CUDA implementation for moderatly dense datasets, and slightly worse for the Aneurism and Cayley datasets, which are sparse datasets on which the HistoPyramid excels.

We also experimented with various detail changes in the algorithm. One was to position the vertices at the edge midpoints, removing the need for sampling the scalar field in the extraction pass, as mentioned in Section 4. In theory, this should increase performance, but experiments show that the speedup is marginal and visual quality drops drastically, see Figure 6. In addition, we benchmarked performance with different texture storage formats, including the new integer storage format of SM4. However, it turned out that the stor-

age type still has relatively little impact in this hardware generation. We therefore omitted these results to improve readability.

6 Conclusion and Future work

We have presented a fast and general method to extract iso-surfaces from volume data running completely on the GPU of OpenGL 2.0 graphics hardware. It combines the well-known MC algorithm with novel HistoPyramid algorithms to handle geometry generation. We have described a basic variant using the HistoPyramid for stream compaction and expansion, which works on almost any graphics hardware. In addition, we have described a version using the geometry shader for stream expansion, and one version implemented fully in CUDA. Since our approach does not require any pre-processing and simply re-generates the mesh constantly from the raw data source, it can be applied to all kinds of dynamic data sets, be it analytical, volume streaming, or a mixture of both (e.g. an analytical transfer function applied to a static volume, or volume processing output).

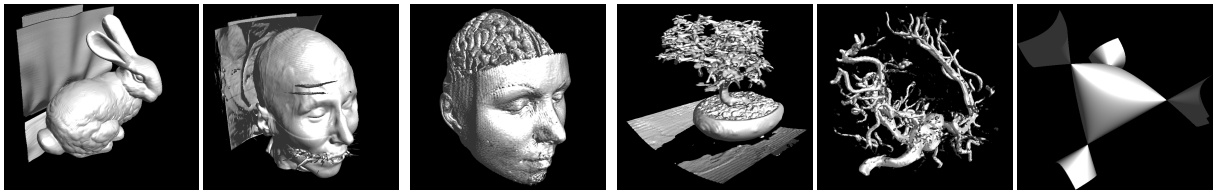
We have conducted an extensive performance analysis on both versions, and set them in contrast to the MT implementation of the Nvidia SDK-10, and an optimized version of the CUDA-based MC implementation provided in the CUDA SDK 1.1. At increasing data sizes, our algorithms outperform all other known GPU-based iso-surface extraction algorithms. Surprisingly, the vertex-shader based variant of the algorithm (GLHP-VS) is also the fastest on recent DX-10-class hardware, even though it does *not* use any geometry shader capabilities.

In direct comparison, Scan and HistoPyramids have some similarities (the Scan up-sweep phase and the HistoPyramid construction are closely related), while the difference lies in the extraction process. Scan has the advantage that only one table lookup is needed, as long as scatter-write is available. For HistoPyramids, each output element extraction requires a $\log(n)$-traversal of the HistoPyramid. Despite that algorithmic complexity, the HistoPyramid algorithm can utilize the texture cache very efficiently, reducing the performance hit of the deeper traversal. A second difference is that Scan's output extraction iterates over *all input elements* and scatters the relevant ones to output, while HistoPyramid iterates on the *output elements* instead. Scan uses two passes to minimize the impact of this disadvantage for the MC application, and often succeeds. However, if a lot of the input elements are to be culled, which is the case with MC for larger and sparse volumes, the HistoPyramid algorithms can play out their strengths, despite the deep gathering traversal. This is shown by the performance of GLHP-VS on the Aneurism and Cayley datasets.

While the CUDA API is an excellent GPGPU tool, seemingly more fitting to this issue, we still feel that a pure OpenGL implementation is of considerable interest. First of all, the OpenGL implementation still outperforms all other implementations. Further, it is completely *platform-independent*, and can thus be implemented on AMD hardware or even mobile graphics platforms, requiring only minor changes. This aside, we still see more future potential for our CUDA implementation, which we believe is not yet fully mature. The chunk/layer structure does remedy CUDA's lack of render-to-2D texture, which brings the CUDA implementation up to speed with the OpenGL approach and introduces a flexible data structure that requires a minimal amount of padding. But we believe that our CUDA approach would benefit significantly from a traversal algorithm that iterates over every *output-producing input element*, which would allow to calculate edge intersections once per geometry producing voxel and triangles to be emitted using scattered write, similar to the Scan-based approach (CUDA1.1+). We have begun investigating approaches to this problem, and preliminary results look promising.

Further, our algorithm can be applied to any MC variant that can be factored into two distinct phases, with the first phase determining the local number of triangles in the case triangulation, and a second phase extracting the vertices one-by-one. Under these preconditions, the approach of Nielson [13], for example, would be a potential candidate for our algorithmic framework. Since HP algorithms often are memory bandwidth restricted, the extra computations and evaluations needed for Nielson's approach could virtually be introduced for free.

A port of Marching Cubes to OpenGL ES would be a good reference for making general data compaction and expansion available on mobile graphics hardware. As already mentioned, our geometry generation approach is *not* specific to MC; its data expansion principle is general enough to be used in totally different areas, such as providing geometry generation for games, or advanced mobile image processing. For geometry shader capable hardware, we are curious if the gap between geometry shaders and HistoPyramids will actu-



Bunny, iso=512 CThead, iso=512 MRbrain, iso=1539 Bonsai, iso=36 Aneurism, iso=11 Cayley, iso=0.0

Figure 9: The iso-surfaces used in the performance analysis, along with the actual iso-values used in the extraction process.

ally close. It is fully possible that general GPU improvements will benefit HistoPyramid performance accordingly, and thus keep this HP-based algorithm useful even there.

Future work might concentrate on out-of-core applications, which also benefit greatly from high-speed MC implementations. Multiple Rendering Targets will allow us to generate multiple iso-surfaces or to accelerate HistoPyramid processing (and thus geometry generation) even further. For example, a view-dependent layering of volume data could allow for immediate output of transparency sorted iso-surface geometry. We also consider introducing indexed triangle mesh output in our framework, as they preserve mesh connectivity. For that purpose, we would experiment with algorithmic approaches that avoid the two passes and scattering that the straight-forward solution would require. We have further received notice that this approach should fit to iso-surface extraction from unstructured grids, since the whole approach is independent of the input's data structure: It only requires a stream of MC cells.

Acknowledgment. We thank Simon Green and Mark Harris for helpful discussions and comments in this work, and for providing the optimized version of the Marching Cubes implementation from the CUDA SDK 1.1.

References

[1] L. Buatois, G. Caumon, and B. Levy. GPU accelerated isosurface extraction on tetrahedral grids. In *International Symposium on Visual Computing*, 2006.

[2] H. Carr, T. Moller, and J. Snoeyink. Artifacts caused by simplicial subdivision. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):231–242, Mar. 2006.

[3] F. Goetz, T. Junklewitz, and G. Domik. Real-time marching cubes on the vertex shader. *Eurographics 2005 Short Presentations*, Aug. 2005.

[4] M. Harris. Parallel prefix sum (scan) with CUDA. Nvidia CUDA SDK 1.1, 2007.

[5] M. J. Harris, W. V. B. III, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. *Proceedings of Graphics Hardware*, 2003.

[6] D. Horn. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Stream Reduction Operations for GPGPU Applications, pages 573–589. Addison-Wesley, 2005.

[7] G. Johansson and H. Carr. Accelerating marching cubes with graphics hardware. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*. ACM Press, 2006.

[8] J. Juliano and J. Sandmel. GL_EXT_framebuffer_object. OpenGL extension registry, 2005.

[9] P. Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surface. In G. Greiner, J. Hornegger, H. Niemann, and M. Stamminger, editors, *Proceedings Vision, Modeling and Visualization 2005*, pages 241–248. IOS Press, infix, 2005.

[10] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. *Pacific Graphics 2004 Proceedings*, 2004.

[11] W. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (SIGGRAPH 87 Proceedings)*, 21(4):163–170, 1987.

[12] C. Montani, R. Scateni, and R. Scopigno. A modified look-up table for implicit disambiguation of Marching Cubes. *The Visual Computer*, 10:353–355, 1994.

[13] G. M. Nielson. On marching cubes. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):283–297, July-Sept 2003.

[14] Nvidia. CUDA programming guide version 1.1, 2007.

[15] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. *Joint Eurographics - IEEE TVCG Symposium on Visualization*, pages 292–300, 2004.

[16] The Stanford volume data archive. `http://graphics.stanford.edu/data/voldata/`.

[17] Y. Uralsky. DX10: Practical metaballs and implicit surfaces. GameDevelopers conference, 2006.

[18] Volvis volume dataset archive. `http://www.volvis.org/`.

[19] G. Ziegler, A. Tevs, C. Theobalt, and H.-P. Seidel. GPU point list generation through histogram pyramids. Technical Report MPI-I-2006-4-002, Max-Planck-Institut für Informatik, 2006.