Eurographics Conference on Visualization (EuroVis) 2016
K.-L. Ma, G. Santucci, and J. van Wijk
(Guest Editors)

Volume 35 (2016), Number 3

Parallel Marching Blocks: A Practical Isosurfacing Algorithm for Large Data on Many-Core Architectures

Paper ID: 103

Abstract

Interactive isosurface visualisation has been made possible by mapping algorithms to GPU architectures. However, current state-of-the-art isosurfacing algorithms usually consume large amounts of GPU memory owing to the additional acceleration structures they require. As a result, the continued limitations on available GPU memory mean that they are unable to deal with the larger datasets that are now increasingly becoming prevalent.

This paper proposes a new parallel isosurface-extraction algorithm that exploits the blocked organisation of the parallel threads found in modern many-core platforms to achieve fast isosurface extraction and reduce the associated memory requirements. This is achieved by optimising thread co-operation within thread-blocks and reducing redundant computation; ultimately, an indexed triangular mesh could be produced.

Experiments have shown that the proposed algorithm is much faster (up to $10\times$*) than state-of-the-art GPU algorithms and has a much smaller memory footprint, enabling it to handle much larger datasets (up to* $64\times$*) on the same GPU.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing Algorithms

1. Introduction

High-speed isosurface extraction has long been an important technique in the interactive visualisation of scalar fields and has been used in a wide variety of applications – medical imaging, molecular surface visualisation, physics-based simulation, etc. It provides a simple and effective way of identifying and visualising distinct surfaces present within volume data and has become an important tool for exploring and understanding such data sets.

Isosurfaces are typically visualised either by rasterising the polygons extracted by the Marching Cubes (MC) algorithm [LC87, DZTS08] or by direct ray casting [HL09, VMD08, KWH09]. While ray casting, when properly applied, can produce superior images, it does not create an explicit representation of the isosurface. Having such a surface model is useful in many applications, such as volume or surface area calculations, freeform modelling, surface-based simulation, surface fairing or surface-related effects for movies and games, as it allows further processing of the geometry. In such cases, the rapid availability of an explicit surface representation can be extremely beneficial. As a result, 30 years after its first introduction, Marching Cubes (and its many variants) still remains popular and in regular use.

In recent years, the resolution of volume datasets from both scanning devices and simulations has continually improved. Accordingly, tools to be applied to such data, including those that extract isosurfaces, must scale effectively to support these increased volumes and their associated memory requirements. However, the main stumbling block associated with large datasets is the GPU memory consumption – not only of the input volume data but also of the accelerating structures used and the output data associated with the large number of polygons generated. This places a heavy burden on the available GPU memory, which continues to be relatively small. If the memory footprint of an algorithm exceeds the available storage, it will simply fail. To cope with the larger datasets that are becoming increasingly available nowadays, there is an urgent need for a fast and efficient algorithm that makes reduced demands on the GPU memory.

Recent GPU-based approaches such as NVIDIA's CUDA SDK [NVI15b] and Histogram Pyramid [DZTS14] (which will be referred to as *nvsdk* and *hpmc*, respectively) provide high-speed MC implementations, but they fail to handle large datasets, as their large memory requirements rapidly deplete the available storage capacity.

Previous parallel methods have treated the MC cells independently, using an individual thread for each [DBG10, NVI15b, DZTS14, CJD15] — the triangles are thus created one-by-one, and coincident vertices are not reused. As a result, vertices are calculated and stored by multiple independent threads; these duplicated vertices are redundant, and their creation has a significant adverse effect on the computing time expended and the amount of memory used.

In a serial implementation, it is possible to merge the duplicated vertices [LCGR02], but this is not easy in a many-core parallel

computing environment. Since the numbers of vertices and triangles output by individual MC cells are not constant (a priori, the total numbers are unknown), and the output mesh is created dynamically, it is not trivial to parallelise efficiently the merging of the duplicated vertices and their assembly into a compact indexed triangle mesh. Indeed, NVIDIA's specialist Simon Green [Gre10] said that this was "too complicated" when he wrote their parallel MC code.

In this paper, we introduce Parallel Marching Blocks (PMB), a block-based parallel MC algorithm for fast, full-resolution isosurface-extraction. PMB generates an indexed triangle mesh, even for complex isosurfaces and dynamic isovalues. Its use provides the benefits of rapid empty-block removal, fast data caching and reuse, and real-time interactivity to support the identification of the best threshold (isovalue) for the desired isosurface, and it works on much larger datasets than was previously possible. The main contributions of the paper can be summarised as follows.

1) The new GPU isosurfacing algorithm presented is fast (up to 10× faster than existing state-of-the-art GPU algorithms) and has a low memory footprint; this enables it to cope with much larger datasets (64×) than them, as they often run out of GPU memory when the datasets are large.

2) In contrast to previous parallel algorithms, which assign one independent thread to each MC cube, PMB employs a simple two-level block layout in which the voxel-blocks are arranged to fit perfectly into the thread-block execution units to be found in modern many-core architectures. All of the threads within a block collaborate to jointly process a voxel-block in a synchronised manner, with shared memory being used to cache each thread's intermediate data so that they can be re-used by other threads.

3) Within the particular restrictions of the parallel computing environment, PMB parallelises the merging of the duplicated vertices and the creation of a compact indexed triangle mesh in which a vertex that is shared by multiple triangles (within a voxel-block) is calculated and stored only once – its unique index is calculated from an offset cached in the shared memory. This is in sharp contrast to the "triangle soup" (groups of unorganised triangles with many redundant/duplicated vertices) produced by previous GPU approaches.

4) PMB uses a new look-up table (with only 12 entries) to expedite the extraction of the vertices. The table maps an edge of an MC cell to a distinct edge of one of its neighbour cells. Its use means that each cell needs to output at most 3 vertices in contrast to the previous up-to-15 vertices. This significantly reduces the amount of computation associated with vertex generation. It should be emphasised that use of this table is not restricted to GPU or parallelised implementations; it can also deliver benefits in standard serial isosurfacing implementations.

5) To compute offsets for the output data, previous GPU MC algorithms have used the global prefix-sum function; unfortunately, this requires the use of several very large arrays stored in the GPU global memory. In place of this, we perform a new local prefix sum within each block. The resulting offsets are retained only in the local shared memory (but not in the global memory) – as a result, we avoid the need to store the large global arrays. Experiments have shown that this greatly reduces GPU memory usage.

6) The full source code for PMB will be released alongside this paper, which will be beneficial to many users who are interested in a fast isosurfacing implementation.

The remainder of the paper is organised as follows. Section 2 covers previous work and Section 3 gives an overview of PMB. The algorithm is described in detail in Section 4; the results obtained are presented in Section 5 and discussed in Section 6, with concluding remarks following in Section 7.

2. Related Work

Marching cubes [LC87] is the most commonly used algorithm for extracting isosurfaces from a scalar field. It divides a voxel grid into cubes (the so-called MC cells) and marches through these one by one, processing each independently. In each MC cell, variable numbers of triangles (up to 5) are produced to approximate the isosurface within the cell. MC has spawned many variants; a survey of them was published by Newman and Yi [NY06].

Some acceleration techniques have used complex pre-processing strategies [KW05]. Unfortunately, this greatly inhibits data exploration (dynamic isosurfacing) in which users change the isovalues frequently at runtime in order to investigate structures within the dataset. While Johansson and Carr [JC06] improved MC performance by precomputing the topology for each cell and using a kd-tree to cull empty regions, they noted that the pre-processing on the CPU limited the overall speed of the algorithm. Other approaches have built hierarchical accelerating data structures, such as octrees [WVG92] and kd-trees [LSJ96]. Kipfer et al. [KW05] identified empty regions using an interval tree.

While MC was originally designed for serial processing, recent variants have attempted to harness the processing power of multi-core processors by introducing parallelised versions, with differing degrees of success. The ever-increasing sizes of the datasets to be analyzed have made parallel isosurface extraction very attractive. However, while one can conceptually map MC algorithms to data-parallel architectures as each MC cell is processed individually, a naive mapping will generally prove to be rather inefficient. The reason is that the distribution of the isosurfaces throughout the volume tends to be highly non-uniform, which presents unbalanced workloads to the processing cores. In addition, the number of MC cells is large, which leads to high memory bandwidth and heavy computational demands during processing. Moreover, it is not easy to efficiently parallelise the merging of the duplicate vertices produced when creating the indexed triangle mesh [Gre10].

Recently, there is a new parallel MC algorithm called Flying Edges (FE) [SMG15], which is implemented on multi-core CPUs, and the performance is reported to be up to 11× faster than previous CPU algorithms. In FE, a CPU thread processes a row of MC cells along X-axis (like a lamb kebab), so that multiple threads can independently process multiple rows of cells in parallel. While within each thread, all the MC cells along a kebab have to be processed one-by-one in a serial manner, such that coincident vertices could be merged during the serial processing of the consecutive

cells along a kebab by an individual thread. In this sense, FE is a coarse-grained parallel algorithm, where each thread has to process hundreds or thousands of cells in serial. If the volume resolution is $N \times N \times N$, one thread has to process N cells (along x-axis) one-by-one. Therefore, the coarse-grained parallelism of FE is more suitable for multi-core CPUs than for many-core GPUs, which are good at large amount of light-weight threads in a fine grained parallelism algorithm, such as the proposed PMB, in which each thread processes only one single MC cell, and all the N^3 cells could be processed in parallel as long as there are enough GPU cores.

Other examples have been provided by Tatarchuk et al. [TSD07] who proposed a technique using GPU geometry shaders to generate the triangle geometry and Dias et al. [DBG10] who employed the same global prefix sum approach as *nvsdk* in an application for polygonising convolution molecular surfaces. Schmitz et al. [SSO*10] implemented a modified dual contouring on the GPU, while Loffler et al. [LS12] extracted isosurfaces from volumetric terrain datasets with complex caves and overhangs.

Recent CUDA-based MC implementations have generated indexed triangles [GWBO12, CJD15], but results have shown that their speed performance is worse than approaches that create triangle soup [NVI15b, DBG10], as generating the indexed triangles introduces severe overheads. If triangle soup is generated, a post-processing step [Wnb12] can weld triangle vertices by eliminating redundant vertices and coincident edges, but this may introduce even more computing time and extra memory usage. Miller et al. described a faster and more reliable vertex merging approach in [MMM14].

Although view-dependent or LoD-based isosurfacing methods implemented on GPUs can work in real time [SBD15], this performance level is usually gained only at the cost of a reduced quality of the resulting mesh, which may be too coarse for further processing or analysis.

Hughes et al. [HLJ*13] proposed in-kernel stream compaction on the GPU that performs stream compaction using bitwise binary operations; however, it is unable to perform stream expansion. Unfortunately, the MC algorithm is a mix of stream compaction and expansion.

Parallel prefix sum (or parallel scan) creates, in parallel, a table that associates each input element with output offsets; Harris [HSO07] designed an efficient implementation of this. NVIDIA's CUDA SDK (*nvsdk*) [NVI15b] provides a high-speed MC implementation using the global prefix-sum function from the highly optimised Thrust library to compute output offsets for the triangle data generated. However, as explained below, *nvsdk* is unsuitable for medium or large datasets as it rapidly exhausts the available GPU memory.

For an N^3 volume, *nvsdk* performs global prefix-sum operations for two arrays: the "number of vertices" array indicates how many vertices each cell will generate, and the "voxel occupied" array indicates whether or not an MC cell is empty. For these two global prefix-sum operations, four 32-bit-integer arrays must be available, each having the same length as the total number of voxels in the volume. To skip empty space, *nvsdk* also needs another large 32-bit-integer array to compact all of the active voxels. These five arrays consume $5 \times 4 \times N^3$ bytes. Thus, for a 1024^3 volume dataset, the memory consumption of the accelerating structure will be 20 GB, which is far beyond the capacity of even the most modern GPU. Moreover, this does not even include the storage requirements of the input volume data and the output mesh data.

Dyken et al. [DZTS08, DZTS14] proposed *hpmc*, a high speed MC algorithm on the GPU, based on the Histogram Pyramid (HP) data structure introduced by Ziegler et al. [ZTTS06]. However, its pyramid structure leads to large memory consumption (often larger than the input volume itself). This is because all of the HP levels are stored as mipmapped textures in which each 32-bit integer texel has to store the sum of the vertex counts from all of its sub-level texels, with the total vertex count being stored in the single texel at the top mipmap level.

When the data size increases, the memory footprint of the accelerating structure inevitably grows too. The GPUs memory has to accommodate not only the input volume data, but also the output mesh and the accelerating structure involved. So an accelerating algorithm with high performance and a small memory footprint can offer significant benefits in the GPU-based processing of the large datasets that are becoming the norm [BHP14]; PMB has this potential.

3. Algorithm overview

In modern many-core architectures, hardware cores and threads are organised into many fixed-sized groups (blocks of cores/threads), as shown in Figure 1. And all of the threads within a same block can use the shared memory to exchange reusable data with each other. PMB explicitly exploits this feature using a block-based accelerating structure. This is in sharp contrast to previous GPU algorithms in which each thread processes a single MC cell independently of the others.

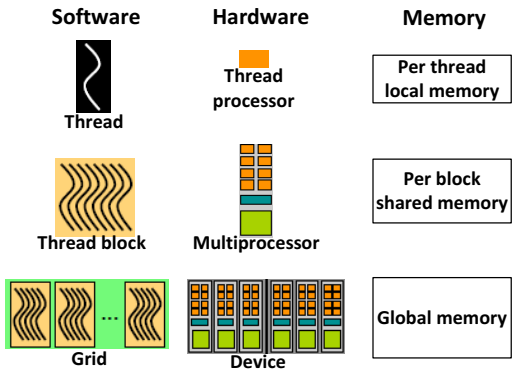

Figure 1: *Block-based hierarchies of software, hardware and memory on GPU architectures, which are corresponding to each other along the horizontal direction.*

As shown in Figure 2, the PMB algorithm is organised with a two-level 3D blocking hierarchy, where the input 3D volume is broken into regular voxel-blocks, and each voxel-block will be processed by a single thread group, which forms a 3D thread-block. Each thread in the thread-block will process only one cuboid MC

cell, but will co-operate closely with other threads within the same thread-block.

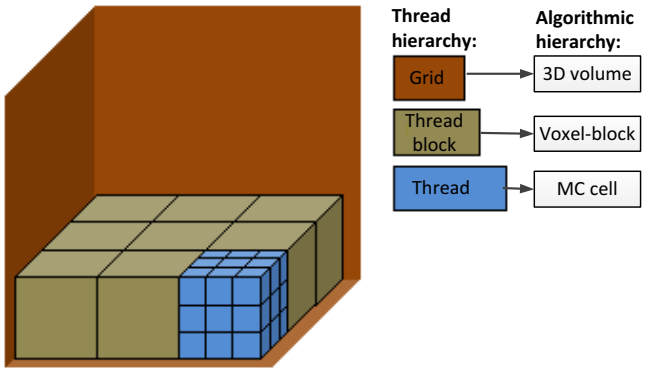

Figure 2: *Mapping of algorithmic hierarchy onto GPU thread hierarchy.*

Since all of the GPU threads are grouped into an array of 3D thread-blocks, instead of building a full-level accelerating hierarchy from the input volume data (as in an octree or span space decomposition), we use a simple, two-level blocking layout in which the 3D voxel-blocks fit exactly into the blocked structure of the execution units of the many-core hardware being targeted. Thus, the input volume data (resolution $Nx \times Ny \times Nz$) is treated as an array of cuboid bricks of voxels; the voxel resolution of each brick is set to $Bx \times By \times Bz$. As illustrated in Figure 2, PMB's algorithmic hierarchy is perfectly mapped onto the hardware thread hierarchy of the modern GPUs, which enables us to maximize the utility of the hardware.

Furthermore, to fully exploit the high-speed shared memory on the GPUs, we especially optimise thread co-operation within each thread-block – via shared memory, all threads within a block can share data that are loaded and computed by other threads. This helps to avoid redundant computation and storage.

We also employ a new local intra-block prefix sum operation to compute the output offset for the vertex and triangle data generated, so that the threads can identify the correct output address at which to jointly output the compact indexed triangle mesh. Since the local prefix-sum operations are performed only inside the thread-blocks, the resulting offset data is stored purely in the local registers and shared memory (i.e., on-chip memory), which are distinct from the global GPU memory (i.e., off-chip memory). By this means, we replace slow global memory access with fast local access. This also avoids having to store the five large memory-consuming global arrays involved in *nvsdk*.

To skip empty voxel-blocks, we launch a CUDA kernel to compute the min-max values for each block. The indices of only the active blocks are compacted into a 1D array, by the use of which empty blocks (which generate no triangles) can be removed, leaving only active blocks on which to run the kernel to generate the triangles.

The following section provides greater algorithmic detail; for further implementation detail, please refer to the pseudocode that accompany the paper.

4. The algorithm in detail

The algorithm involves three steps, each implemented by a CUDA kernel. The subsections below elaborate on these individual steps. Figure 3 explains the general data flow through the GPU global memory.

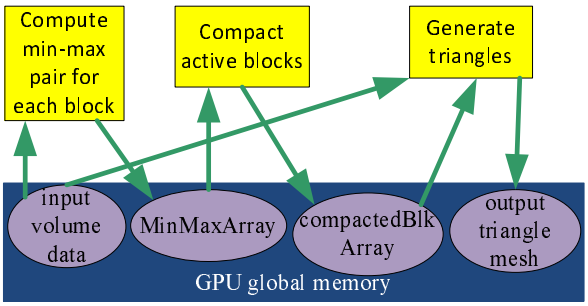

Figure 3: *The data flowchart of the algorithm. Yellow boxes stand for the three CUDA kernels; pink ellipses stand for the data in the GPU global memory, and green arrows indicate the direction of data fetches or writes to the GPU global memory.*

4.1. Computing the min-max pair for each block

In applications in which isosurface extraction is employed, the volume datasets tend to be rather sparse, with many inactive voxels. These occupy much of the volume but make no contribution to the output, so it is expedient to remove them before starting to generate the triangles.

To test whether a block is empty or not, we compute a min-max pair of scalar values for each block, and store the result in a 1D array, the length of which is determined by the total number of blocks: $totalBlkNum = (Nx-1)/(Bx-1) \times (Ny-1)/(By-1) \times (Nz-1)/(Bz-1)$.

This is accomplished by a "block-based stream-reduction" kernel, using CUDA's fast shuffle instructions, which enable a thread to read a register directly from another thread in the same warp. This allows threads in a warp to exchange or broadcast data collectively [NVI15a].

We launch $totalBlkNum$ 3D thread-blocks, each with $Bx \times By \times Bz$ threads, which collaboratively compute a min-max pair for a voxel-block. Overall, the kernel has three steps. 1) Compute the min-max pair for each warp using a warp-reduce procedure [Dem13], which uses the fast shuffle instruction to exchange data between threads. 2) The first thread of each warp now holds the result of the warp; this partial result is written to the shared memory. 3) After thread synchronisation, the first warp alone reads the partial results from the shared memory and applies the warp-reduce procedure to the partial results of all the warps to produce the min-max pair of the block, which is output to $MinMaxArray$ in the global memory.

4.2. Compacting the active blocks

nvsdk skips empty space at the finest granularity (the voxel level) – all of the voxels are scanned and the non-empty ones are compacted into a large 1D array which is used to remove the empty voxels. We found this fine granularity to have three disadvantages: 1) the scanning and compaction of the non-empty voxels consume a great deal of memory as the number of the voxels to be scanned may be very large; 2) non-empty voxels that are adjacent in the compacted array may not be physically located in adjacent parts of the input volume – this can cause poor GPU memory- and cache-locality and hence impair the overall performance; 3) this separation of their physical locations also means that adjacent threads cannot share their intermediate results.

PMB, in contrast, skips empty space at the voxel-block level. This reduces demands on memory space as the scanning and compacting arrays will be much smaller (because there are many fewer blocks than voxels) and improves memory locality as adjacent threads in a thread-block deal only with voxels in the same voxel-block, which must therefore be located in neighbouring parts of the 3D volume. This also enables adjacent threads to share the intermediate results cached in the shared memory.

To remove the empty voxel-blocks, we compact the indices of the active blocks into a 1D array, which will be used later in the "generatingTriangles" kernel (Section 4.3). This compacting step is performed by a CUDA kernel which assigns one thread to each voxel-block. For this kernel, we launch $totalBlkNum$ threads, each of which computes a global offset for its voxel-block. Our experiments suggest that optimal performance is achieved by giving each thread-block 128 threads, as this provides suitable thread occupancy for a thread-block. As $warpSize$ is 32, the number of warps for a thread-block, $nWarp$, is 4.

To compute the output offset for the index of a non-empty voxel-block, we use a local prefix sum to find the intra-thread-block offset followed by an atomic operation (performed only once per thread-block) to obtain the inter-thread-block offset. Adding these produces the global offset for the current thread. Here, the calculation of the local prefix sum employs a procedure called $warp-scan$ [Dem13], which uses the fast shuffle instruction to exchange data between threads.

This compacting kernel acts as follows. First, each thread calculates a bool variable $bTest$ by testing if its voxel-block is empty (by comparing the input isovalue with the block's min-max pair). The $warp-scan$ procedure [Dem13] is then applied to $bTest$ among all the threads inside each warp. And then, $warp-scan$ is applied again but to the partial results of the multiple warps within a thread-block, with the results stored in the shared memory. Since $nWarp < warpSize$, a single warp can perform this warp-scan procedure on all of the partial results within the thread-block. After that, an atomic operation adds the entire thread-block's sum to the global sum ($activeBlkNum$), which is then added to each thread's sum to generate the global offset for each thread in the thread-block. Note that the atomic operation is performed only once per thread-block, so it does not have a detrimental effect on the overall performance. Finally, if the current voxel-block is not empty, its index is written to a 1D array $compactedBlkArray$ using the global offset as the address.

4.3. Generating the triangles

The final CUDA kernel generates triangles for only the active voxel-blocks, as identified by the 1D array $compactedBlkArray$. This kernel is a block-based parallel MC algorithm in which $activeBlkNum$ voxel-blocks are processed each by a 3D thread-block with $Bx \times By \times Bz$ threads.

Note that an MC cell is not processed independently by a thread as in previous parallel algorithms; rather, all of the cells in a block are jointly processed by a thread-block in a synchronised, collaborative manner. While each thread processes just a single voxel, it reuses intermediate results (the voxel sampling values, and the vertices generated and their offsets) from other threads via shared memory, so that duplicate vertices could be merged as a natural part of the mesh creation.

In these computations, each thread outputs at most 3 vertices (not up-to-15, as in standard MC implementations); these can be re-used by other threads. The outcome is an indexed mesh in which a vertex that is shared by multiple triangles (within a voxel-block) is computed and stored only once; moreover, its unique vertex-index is calculated from the cached local offset, which is reused by other threads when outputting multiple triangles that share this vertex. This kernel can be summarised in the 3 steps described in the following subsections.

4.3.1. Generation of at most 3 vertices by each thread

Firstly, each thread samples a voxel value from the input volume and caches it into the shared memory, for reuse by adjacent threads after thread synchronisation. From the 8 corner-voxel values cached, the thread computes an MC case index, $cubeCase$, which will be used to look up MC tables.

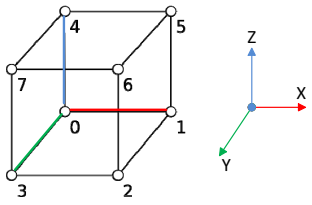

Figure 4: *A 3D MC cell (with 8 corner voxels) and its three distinct edges.*

Within a grid of 3D MC cells, a typical cell edge is shared by 4 adjacent cells. In previous GPU implementations, this edge is processed repeatedly by each of the adjacent MC cells. To avoid these redundant calculations, we associate the thread with the base voxel that lies at its MC cell's origin (relative to the 3 coordinate axes). We define the 3 distinct edges, $xEdge$, $yEdge$, $zEdge$, of base voxel (i,j,k), to be the edges from (i,j,k) to (i+1,j,k), (i,j+1,k), (i,j,k+1), respectively. The MC cell in Figure 4 is based at voxel 0, and its $xEdge$, $yEdge$ and $zEdge$ are shown in red, green and blue, respectively.

We restrict each thread to generate vertices along only the 3 distinct edges of its voxel. Vertices lying on the other edges of the MC cell are found from the distinct edges of adjacent threads. This avoids the redundancy that the standard MC approach introduces

when finding where the triangle vertices occur between the adjacent cuboid cells.

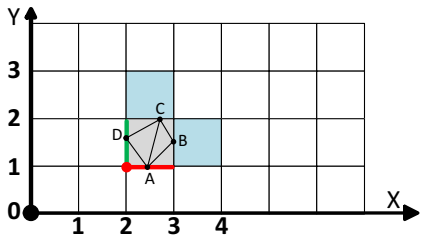

Figure 5: *2D illustration of vertex generation (A and D) along only distinct edges of a cell (gray). Vertices B and C will be "borrowed" from other adjacent cells (blue).*

This is most easily illustrated in 2D – see Figure 5, where the MC cell shown in grey, with origin at base voxel (2,1), will produce two triangles ($\triangle ABC$ and $\triangle ACD$); Vertices A and D (on $xEdge$ and $yEdge$ of this voxel, shown in red and green, respectively) are the only 2 vertices found by the thread processing this voxel. Vertices B and C are found by adjacent threads processing the MC cells shown in blue – $yEdge$ of voxel (3,1) and $xEdge$ of voxel (2,2), respectively. In order to assemble output triangles for the current thread processing the gray cell at base voxel (2,1), vertices B and C will be "borrowed" from these two adjacent threads.

The position of a vertex is found by linear interpolation between the two end-points of the distinct edge. The per-vertex normal vector is computed by central differencing of 6 samples; note that the block boundaries do not affect this calculation as the 6 samples are acquired from the original input 3D volume texture via hardware trilinear interpolation.

4.3.2. Computing output offsets

To compute offsets for the output mesh data, previous GPU MC algorithms [NVI15b, CJD15, DBG10] used the global prefix-sum function; as noted earlier, this requires several large arrays to be stored in the GPU global memory. PMB, however, performs a local prefix sum within each block, and the resulting offsets are retained only in the local register and shared memory (but not the global memory), thus avoiding the need to store the global arrays.

The output offsets are calculated using the local intra-block prefix sum described in Section 4.2, with shuffle instructions again supporting the exchange of data between threads. Here the $warp-scan$ procedure is applied to two variables ($numVerts$ and $numTris$, the numbers of vertices and triangles generated by each thread) in one go. A $warp-scan$ is first performed inside a thread-warp, and then it is applied to the partial results of multiple warps within the thread-block to obtain the local offsets within a block. Finally, an atomic operation adds the sum of the block (local sum) to the global sum, which will be added to each thread's sum to determine the global offset for each thread. This produces the local offset (within a block) and the global offset for the vertex data ($offset2$) and the triangle data ($offset1$).

Each thread now outputs the vertices thus generated into a 1D array $VertexArray$ at the address indicated by the global vertex offset $offset2$. The local offset ($intraBlockOffset$) for each vertex is cached in the shared memory and is reused (see Section 4.3.3) to calculate a unique vertex-index by other threads when assembling their own triangles that share this vertex. The prefix sum of all the previous blocks' vertices, $interBlockOffset$, which is returned by the atomic operation above, is also cached for later use.

edgeID	*edgemap.x*	*edgemap.y*	*edgemap.z*	*edgemap.w*
0	0	0	0	0
1	1	0	0	1
2	0	1	0	0
3	0	0	0	1
4	0	0	1	0
5	1	0	1	1
6	0	1	1	0
7	0	0	1	1
8	0	0	0	2
9	1	0	0	2
10	1	1	0	2
11	0	1	0	2

Table 1: *neighborMappingTable maps an edgeID (of one of the 12 edges of an MC cell) to a distinct edge of a neighbour cell indicated by the four integers stored in edgemap.xyzw.*

4.3.3. Assembling triangles using vertex-indices

Each thread will output up to 5 triangles depending on its MC case index, $cubeCase$. To output the triangle data in an indexed form, one must output three correct vertex-indices for each triangle to the storage locations indicated by the global offset $offset1$. Their calculation is described below.

For each thread's base voxel, we have cached in the shared memory the 3 bool variables in $xyzEdges$ which indicate if its 3 distinct edges will produce a vertex. Vertices on the other 9 edges of the MC cell are "borrowed" from the distinct edges of neighbouring voxels, as shown in Figure 5. For this, we first find which of those edges have an intersection with the isosurface; an active edge's ID ($0 <= edgeID < 12$) can be retrieved from the MC edge table accessed by $cubeCase$.

We now design another table, $neighborMappingTable$ (see Table 1) whose 12 entries reflect the 3D relations of the adjacent MC cells – it maps an edge, $edgeID$, of the current cell to a distinct edge of one of its neighbour cells. Of the four integer components in $edgemap$, the first three ($edgemap.xyz$, with each component 0/1) indicate the XYZ offsets to the neighbour voxel, and the fourth ($edgemap.w$) indicates the distinct edge of this neighbour to which $edgeID$ is mapped (0, 1, 2 for $xEdge$, $yEdge$, $zEdge$, respectively). The small size of this new table allows it to be stored conveniently in the high-speed constant memory of the GPU.

The position of the neighbour is (i+$edgemap.x$, j+$edgemap.y$, k+$edgemap.z$), and from its location in shared memory are retrieved the correct $intraBlockOffset$ and the information about its distinct edges ($xyzEdges$).

The vertex-index for the original edge, $edgeID$, is now evaluated as $vertexIndex = edgeSum + intraBlockOffset + interBlkOffset$. Here, $edgeSum$, the prefix sum among the 3 distinct edges of this neighbour voxel, is calculated from $xyzEdges$ and $edgemap.w$.

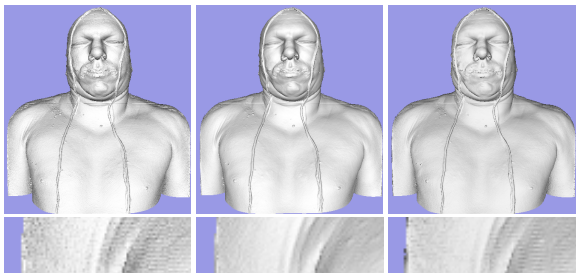

Figure 6: *Quality comparison for nvsdk (left), PMB (middle) and hpmc (right) using the visMale dataset. The lower images show corresponding close-up views.*

volume resolution	$n1$ (in 2^{20})	$n2$ (in 10^6)	*nvsdk* (fps)	*hpmc* (fps)	*PMB* (fps)
512^3	128	1.3	25.6	69.3	193.1
512^2x1024	256	2.1	13.3	58.3	138.7
1024^2x512	512	3.4	fail	fail	87.5
1024^3	1024	5.0	fail	fail	55.2
1024^2x2048	2048	8.4	fail	fail	32.5
2048^2x1024	4096	13.4	fail	fail	21.3
2048^3	8192	20.1	fail	fail	13.2
2048^2x4096	16384	33.5	fail	fail	7.6

Table 2: *Performance comparison (in frames per second) for dynamic isovalues among nvsdk, hpmc, and PMB using the analytical Cayley surface with isovalue -0.012. $n1$ is the number (in 2^{20}) of input voxels; $n2$ is the number (in 10^6) of output triangles. "fail" means the program ran out of GPU memory and crashed.*

vertexIndex is output to the global array for triangle data at the address indicated by $offset1$.

The result is an indexed mesh in which each vertex shared by multiple triangles (within a voxel-block) is computed and stored only once (this includes both its XYZ-coordinates and its per-vertex normal vector) allowing the vertex to be accessed by a unique index, which is reused by multiple threads when assembling and outputting their own triangles.

As the vertex and triangle data of the mesh are already stored in the global memory of the GPU, for display we can map them into OpenGL VBOs (Vertex Object Buffer) without any memory copies and thus render the mesh with a single draw call $glDrawElements()$.

5. Results

PMB was compared with two state-of-the-art GPU isosurface-extraction algorithms *nvsdk* and *hpmc*, the source codes of which are publicly available. For *nvsdk*, we used NVIDIA's latest CUDA SDK (v7.0) [NVI15b], which provides a high speed MC implementation using the global prefix-sum function from the highly optimised Thrust library. *hpmc* [DZTS14], the fastest Marching Cubes implementation yet reported, is an open-source library that extracts isosurfaces using a pyramid structure [DZTS08]. Without changing any algorithmic part of their code, we measured their performance using precisely the same datasets, isovalues, viewing configurations and hardware platform to ensure fair comparisons.

The tests were performed on a desktop workstation equipped with an NVIDIA GeForce GTX Titan GPU, which has 6 GB video memory. This GPU was chosen because of its super-large video memory so that *nvsdk* and *hpmc* would not fail too early, and the comparisons could run on as many datasets as possible. All results were rendered at a screen resolution of 1024^2.

As all computation at runtime takes place solely on the GPU, the performance is CPU-independent. The frame-rates (involving both extraction and rendering) are presented in Tables 2 and 3, which also show the numbers of input voxels and output triangles. All timings in the tables are for dynamic isovalues, i.e., empty-space skipping is performed at every frame on the assumption that the user is interactively changing the isovalue, so a new isovalue is provided for every frame. Screen shots shown in Figures 6 and 7 correspond precisely with the data given in the tables.

data names	volume resolution	$n2$ (in 10^6)	iso-value	*nvsdk* (fps)	*hpmc* (fps)	*PMB* (fps)
visMale	512^3	2.8	0.094	23.8	56.5	135.2
backpack	512^3	3.8	0.2	23.0	49.5	136.8
hazelnut	512^3	3.4	0.2	23.2	53.6	123.2
Xmastree	512^3	1.5	0.15	25.3	67.2	218.9
ncat	512^3	2.5	0.15	23.9	58.5	145.8
aneurism	512^3	0.7	0.19	26.3	75.3	360.1
trabecula	512^3	1.6	0.184	25.0	66.1	194.7
macoessix	512^3	3.8	0.215	22.8	48.6	99.8
abdomen	512^3	3.3	0.315	23.2	52.3	120.5
manix	512^3	3.3	0.354	23.3	53.6	122.2
melanix	512^2x1024	2.5	0.374	fail	55.9	163.3
visFemale	512^2x1024	3.4	0.294	fail	50.6	126.8
stagBeetle	1024^2x512	6.5	0.214	fail	fail	61.3
flower	1024^3	22.1	0.193	fail	fail	18.5
rm	1024^3	49.4	0.369	fail	fail	8.3
beechnut	1024^2x1546	64.2	0.233	fail	fail	6.2

Table 3: *Performance comparison for dynamic isovalues among nvsdk, hpmc, and PMB using a variety of input volumes. $n2$ and fail are as in Table 2.*

5.1. Comparison of rendering quality

As expected, the three algorithms extract triangular meshes of essentially the same quality, with the same number of triangles; the quality of the rendering is determined by the different methods used for calculating the vertex normals.

For each triangle, *nvsdk* computes only a single planar normal. From Figure 6, it is clear that the rendering quality of *nvsdk* is the worst – the faceted appearance is very obvious when the camera is close. *hpmc* uses forward differences to compute a per-vertex normal vector; this requires 3 extra texturing samples. While its results are better, there are still some "wood grain" artefacts. PMB uses central differences, for which 6 extra texturing samples have to be taken. This makes the per-vertex normal vectors more accurate and produces much smoother results, as illustrated in Figure 6. Despite this additional calculation, PMB is still much faster than the other methods, as described below.

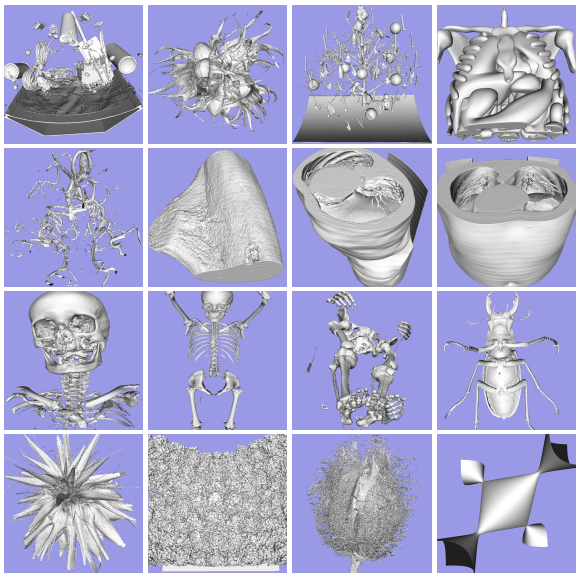

Figure 7: *Rendering results of PMB for a variety of datasets (from left to right and top to bottom): backpack, hazelnut, Xmastree, ncat, aneurism, trabecula, macoessix, abdomen, manix, melanix, visFemale, stagBeetle, flower, rm, beechnut, and cayley.*

5.2. Comparison of performance

To compare the performance of the algorithms under various loads, we first extracted isosurfaces from an implicit surface at 8 different input resolutions. As the input voxel values are evaluated at runtime from an algebraic function, there is no need to store any input volume data. The surface used was the Cayley surface in which the 32-bit float scalar voxel values are found from the function $f(x,y,z) = 1 - 16xyz - 4x^2 - 4y^2 - 4z^2$. PMB extracted isosurfaces at resolutions up to $2048^2 \times 4096$, while *nvsdk* and *hpmc* failed at any resolution higher than $512^2 \times 1024$ on the same GPU – this shows that PMB can cope with $64\times$ the number of voxels that *nvsdk* and *hpmc* can. The performance statistics are shown in Table 2, and the resulting isosurfaces in Figure 7.

Results from comparisons on a wide variety of input volume datasets from CT/MRI scans and physically-based simulations are reported in Table 3. These show that *nvsdk* is the slowest algorithm, and, as it consumes the most GPU memory, it is the earliest to fail for large datasets. PMB was consistently the fastest in all examples and it continued to deliver speedy results even for the largest datasets, for which both *nvsdk* and *hpmc* failed.

5.3. Comparison of memory consumption

The memory consumption of an isosurfacing algorithm depends upon the sizes of three items: the input volume data, the accelerating structure and the output mesh data. As the input volume size is identical for all three implementations, we consider only the latter two items in the following subsections, where the tests found that PMB performed significantly better than *nvsdk* and *hpmc* in both aspects.

5.3.1. The accelerating structures

As mentioned in Section 2, the accelerating structures for *nvsdk* and *hpmc* can consume considerably more GPU memory than the input volume data. Consequently, when applied to large datasets, both methods will run out of memory and crash. In our tests, neither method ran successfully at any data resolution greater than $512^2 \times 1024$, shown as "fail" in Tables 2 and 3.

Apart from the input volume data and output mesh data, PMB needs to store only two 1D arrays: $MinMaxArray$ and $compactedBlkArray$; the length of both of these is less than or equal to the number of voxel-blocks, $totalBlkNum$, which is much smaller than the number of voxels. Our experiments showed that, to balance the thread occupancy in a thread-block with the threads' joint usage of the limited shared memory and registers, the optimal values for the voxel-block resolution $Bx \times By \times Bz$ were $8 \times 4 \times 4$; this provides each thread-block with the optimal 128 threads. As an example, the *rm* dataset, with resolution 1024^3, had 17 million voxel-blocks, and the memory usage of our accelerating structure was less than $17 \times 2 + 17 \times 4 = 102$ MB, which is less than 10% of the input volume's size.

5.3.2. The output triangle mesh

nvsdk and *hpmc* both output "triangle soup" in which the mesh is represented by a group of unorganised triangles, each of which is stored as 3 separate vertices. Vertices shared by multiple triangles are stored multiple times (once for each triangle), so the mesh will contain many repeated vertices. Each vertex has to store the XYZ-coordinates and its normal vector, which fill $3 \times 4 + 3 \times 4 = 24$ bytes. Thus, if a dataset outputs $numTri$ triangles, the resulting vertex data size of *nvsdk* and *hpmc* will be $numTri \times 3 \times 24$ bytes. As an example, the *backpack* dataset outputs 3.8 million triangles, and the resulting vertex data size of *nvsdk* and *hpmc* is $3.8 \times 3 \times 24 = 273.6$ MB.

In contrast, PMB outputs an indexed mesh in which all vertices are computed and stored only once (within each voxel-block). For the same *backpack* dataset, it outputs only 3 million vertices, so the resulting vertex data size is only $3 \times 24 = 72$ MB, that is roughly a quarter ($72/273.6 = 26.3\%$) of the original size of the storage needed by the other algorithms.

6. Discussion

6.1. The ingredients for PMB to be efficient

By developing this technique, we found the following ingredients for the algorithm to be more efficient than previous GPU algorithms.

1) The blocking feature of the algorithm is the key to reducing redundant computation. It enables threads in a block to collaborate closely to jointly produce an indexed triangular mesh by sharing the cached intermediate results, with each MC cell having to output at most 3 vertices. In contrast, *nvsdk* and *hpmc* output up to 15 vertices for each cell.

2) Since we perform the local prefix sum (within the blocks) and add the sum of all the preceding blocks using an atomic instruction to obtain the global offset, the resulting offset data is stored

purely in the local registers and shared memory. As a result, we avoid the need to store in the GPU global memory the five memory-consuming large arrays that are required for the global prefix sum operations used in [NVI15b, CJD15, DBG10]. Experiments have shown that this strategy greatly reduces GPU memory usage.

3) The offsets that are needed to arrange the output compactly can be computed efficiently by exchanging data among threads within thread-warps and thread-blocks.

4) *nvsdk* and *hpmc* skip empty space at the voxel level, which hurts memory coherence since the active voxels that are processed adjacently in a thread-block may not be physically adjacent in the input voxel grid. A side-effect of this is that adjacent threads cannot share and reuse each other's intermediate data. Performing empty-space skipping at the voxel-block level made it possible to ensure that GPU memory- and cache-locality are well preserved among threads within the same block.

6.2. Possible extensions

Our implementation using CUDA on an NVIDIA GPU is only a prototype of PMB, which could possibly be extended in the following directions.

1) It is feasible to generalise the algorithm to other many-core architecture (including AMD GPUs and many-core CPUs such as intel's MIC architecture). Due to the similarities of the blocked thread-organisation and the memory hierarchy (including global memory and shared memory) on these architectures, we cannot see any algorithmic obstacles to stop us from implementing it on these platforms.

2) The block-based algorithm presented is complementary to existing hierarchical accelerating structures (such as an octree) and hence could be combined with them. As a result, much larger out-of-core data could be organised into a block-based octree [GMI08], in which each leaf node is a voxel-block to be processed by PMB.

3) It is straightforward to extend PMB to a multi-GPU environment, in which each GPU deals concurrently with a separate sub-volume of the data. This would open the possibility of real-time dynamic isosurfacing of truly huge volume datasets on a commodity PC fitted with multiple GPUs. Similarly, it could be extended to a multi-node cluster, in which each computing node could have multiple GPUs. We anticipate that PMB will demonstrate good scalability (due to its blocking feature), which will allow significant expansion of the size of the datasets to which it can be applied.

6.3. Limitation of the algorithm

Since PMB operates within individual voxel-blocks (as shown in Figure 8) and there is no direct data communication between blocks (which could be very expensive on GPUs), its output is multiple separate manifold triangular patches (one for each block). The coincident vertices on the cell-edges along the block boundaries are not merged, which means that there are still redundant vertices located only on the boundaries of the blocks. For visualisation purpose, this is not a problem because the resulting mesh has no geometrical seam anywhere of the mesh, which can be guaranteed by the underlying MC algorithm logics employed by PMB. But there are still topological seams across the boundaries of the triangular patches of the individual blocks, because the coincident vertices across blocks are not merged. Since the proportion of the edges

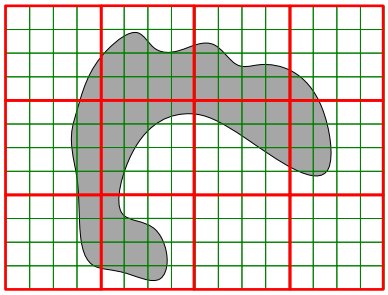

Figure 8: *2D illustration of the limitation: vertex merging are only performed at the interior edges (green) of the blocks, but not performed at the edges on the block boundaries (red).*

(shown in red) on the block boundaries are much smaller than that of the interior edges (shown in green) of the individual MC cells, the majority of the redundant vertices are already merged by PMB. This can be confirmed by the experiment in Section 5.3.2, where the resulting vertex data size is only 26.3% of the size needed by the traditional MC algorithms, which means roughly three quarters of the total vertices are already removed as redundant ones.

However, for applications where a single manifold triangular patch (for the whole volume) is required for further processing such as topological analysis, a postprocess will be necessary to merge those small percentage of redundant vertices located on the block boundaries in order to avoid the topological seams there.

7. Conclusion

We have introduced PMB, a practical isosurfacing algorithm suitable for use with large datasets. Experiments demonstrated that it is much faster than state-of-the-art algorithms, exhibiting a speed-up factor of up to 10× in the tests. Moreover, it uses far less memory than previous GPU methods (providing 64× memory improvement), so it is suitable for use on memory-limited GPUs, and it can be applied to much larger datasets than was previously possible.

As the computation is performed wholly on the GPU, PMB is suitable for dynamic isosurface extraction, as demonstrated in the accompanying executable program and the videos. Its speed and ability to cope with large datasets provide opportunities for its use in topics beyond the current major applications.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and very helpful suggestions, which have enabled significant improvements to the paper. We thank NVIDIA and Dr. Christopher Dyken for make their code open-source, which were instrumental in enabling us to perform comparisons under identical conditions. We acknowledge the organizations that made their datasets available on which we experimented.

References

[BHP14] BEYER J., HADWIGER M., PFISTER H.: A Survey of GPU-Based Large-Scale Volume Visualization. *Proceedings EuroVis 2014)* (2014). 3

[CJD15] CHEN J., JIN X., DENG Z.: GPU-based polygonization and optimization for implicit surfaces. *The Visual Computer 31*, 2 (2015), 119–130. 1, 3, 6, 9

[DBG10] DIAS S., BORA K., GOMES A. J. P.: CUDA-based triangulations of convolution molecular surfaces. In *HPDC'10* (2010), pp. 531–540. 1, 3, 6, 9

[Dem13] DEMOUTH J.: Kepler's SHUFFLE (SHFL): Tips and Tricks. GTC 2013 talk, 2013. 4, 5

[DZTS08] DYKEN C., ZIEGLER G., THEOBALT C., SEIDEL H.-P.: High-speed marching cubes using histopyramids. *Computer Graphics Forum 27*, 8 (Dec. 2008), 2028–2039. 1, 3, 7

[DZTS14] DYKEN C., ZIEGLER G., THEOBALT C., SEIDEL H.-P.: Marching cubes using histogram pyramids. `http://www.sintef.no/hpmc`, 2014. 1, 3, 7

[GMI08] GOBBETTI E., MARTON F., IGLESIAS GUITIÁN J.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7-9 (July 2008), 797–806. 9

[Gre10] GREEN S.: Algorithm to remove shared vertices. `https://devtalk.nvidia.com/default/topic/468576/algorithm-to-remove-shared-vertices/`, 2010. 2

[GWBO12] GRIFFIN W., WANG Y., BERRIOS D., OLANO M.: Real-time GPU surface curvature estimation on deforming meshes and volumetric data sets. *IEEE Trans Vis Comp Graph 18*, 10 (Oct 2012), 1603–1613. 3

[HL09] HUGHES D. M., LIM I. S.: Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on GPUs. *IEEE Trans Vis Comp Graph 15*, 6 (2009), 1555–1562. 1

[HLJ*13] HUGHES D. M., LIM I. S., JONES M. W., KNOLL A., SPENCER B.: Ink-compact: In-kernel stream compaction and its application to multi-kernel data visualization on general-purpose GPUs. *Computer Graphics Forum 32*, 6 (2013), 178–188. 3

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with CUDA. *GPU Gems 3, Chapter 39* (2007), 851–873. 3

[JC06] JOHANSSON G., CARR H.: Accelerating marching cubes with graphics hardware. In *CASCON '06* (Riverton, NJ, USA, 2006), IBM Corp. 2

[KW05] KIPFER P., WESTERMANN R.: GPU construction and transparent rendering of iso-surfaces. In *VMV05* (2005), pp. 241–248. 2

[KWH09] KNOLL A. M., WALD I., HANSEN C. D.: Coherent multiresolution isosurface ray tracing. *Vis. Comput. 25*, 3 (Feb. 2009), 209–225. 1

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3D surface construction algorithm. *Comput. Graph. 21*, 4 (1987), 163–169. 1, 2

[LCGR02] LINGRAND D., CHARNOZ A., GERVAISE R., RICHARD K.: Experimenting with marching cubes. `http://users.polytech.unice.fr/~lingrand/MarchingCubes/applet.html`, 2002. 1

[LS12] LÖFFLER F., SCHUMANN H.: Generating smooth high-quality isosurfaces for interactive modeling and visualization of complex terrains. In *VMV12* (2012), pp. 79–86. 3

[LSJ96] LIVNAT Y., SHEN H.-W., JOHNSON C. R.: A near optimal isosurface extraction algorithm using the span space. *IEEE Trans Vis Comp Graph 2*, 1 (Mar. 1996), 73–84. 2

[MMM14] MILLER R., MORELAND K., MA K.-L.: Finely-threaded history-based topology computation. In *Proceedings of the 14th Eurographics Symposium on Parallel Graphics and Visualization* (Aire-la-Ville, Switzerland, Switzerland, 2014), PGV '14, Eurographics Association, pp. 41–48. 3

[NVI15a] NVIDIA: CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-C-programming-guide/index.html`, 2015. 4

[NVI15b] NVIDIA: CUDA SDK Code Samples. `http://docs.nvidia.com/cuda/cuda-samples/index.html#marching-cubes-isosurfaces`, 2015. 1, 3, 6, 7, 9

[NY06] NEWMAN T. S., YI H.: A survey of the marching cubes algorithm. *Computers & Graphics 30*, 5 (2006), 854 – 879. 2

[SBD15] SCHOLZ M., BENDER J., DACHSBACHER C.: Real-time isosurface extraction with view-dependent level of detail and applications. *Computer Graphics Forum 34*, 1 (2015), 103–115. 3

[SMG15] SCHROEDER W., MAYNARD R., GEVECI B.: Flying edges: A high-performance scalable isocontouring algorithm. In *Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on* (Oct 2015), pp. 33–40. 2

[SSO*10] SCHMITZ L. A., SCHEIDEGGER L. F., OSMARI D. K., DIETRICH C. A., COMBA J. L. D.: Efficient and quality contouring algorithms on the GPU. *Computer Graphics Forum 29*, 8 (2010), 2569–2578. 3

[TSD07] TATARCHUK N., SHOPF J., DECORO C.: Advanced Real-Time Rendering in 3D Graphics and Games. In *ACM SIGGRAPH Courses 26* (New York, NY, USA, 2007), pp. 122–137. 3

[VMD08] VIDAL V., MEI X., DECAUDIN P.: Simple empty-space removal for interactive volume rendering. *J. Graphics Tools 13*, 2 (2008), 21–36. 1

[Wnb12] WNBELL: This example welds triangle vertices together. `https://code.google.com/p/thrust/source/browse/examples/weld_vertices.cu`, 2012. 3

[WVG92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM Trans. Graph. 11*, 3 (July 1992), 201–227. 2

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.-P.: On-the-fly point clouds through histogram pyramids. In *VMV06* (2006), pp. 137–144. 3