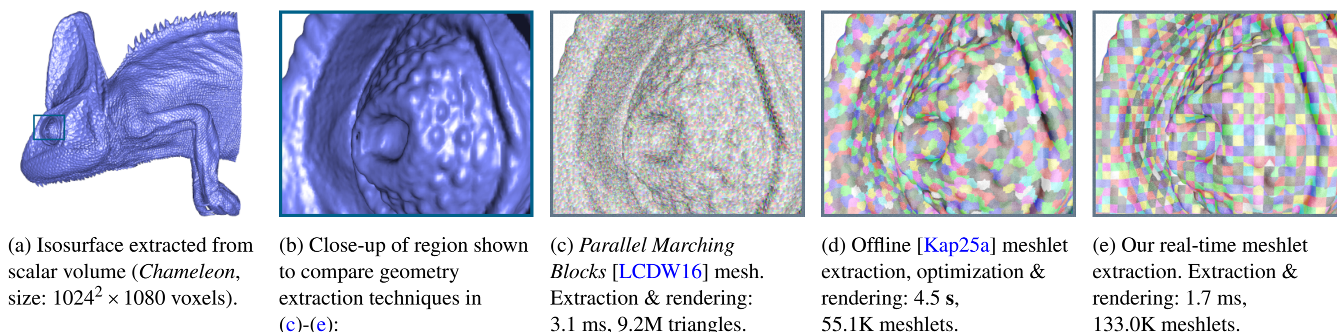
Eurographics Symposium on Parallel Graphics and Visualization (2026)
S. Rizzi, C. Gueunet, S. Dutta (Editors)

Real-Time Meshlet Extraction from Scalar Volumes

A. Kreskowski and G. Rendle and B. Froehlich

Virtual Reality and Visualization Research Group, Bauhaus-Universität Weimar, Germany



(a) Isosurface extracted from scalar volume (*Chameleon*, size: $1024^2 \times 1080$ voxels).

(b) Close-up of region shown to compare geometry extraction techniques in (c)-(e):

(c) *Parallel Marching Blocks* [LCDW16] mesh. Extraction & rendering: 3.1 ms, 9.2M triangles.

(d) Offline [Kap25a] meshlet extraction, optimization & rendering: 4.5 **s**, 55.1K meshlets.

(e) Our real-time meshlet extraction. Extraction & rendering: 1.7 ms, 133.0K meshlets.

Figure 1: Comparison of our real-time meshlet extraction approach with reference methods in an indirect isosurface rendering scenario (a) & (b). Existing block-based extraction approaches [LCDW16] efficiently extract isosurfaces with millions of individual triangles in real time (c). *Meshlets* are geometry clusters that allow efficient rendering of meshes, which can be created in a near-optimal manner by existing offline methods (d). Our proposed approach (e) extends previous real-time block-based isosurface extraction approaches to extract meshlets in a single, efficient isosurface extraction pass that outperforms the state of the art in many cases. Isosurfaces extracted using each approach consist of the same triangles and reference the equivalent vertex positions and normals, and are therefore visually indistinguishable from each other. Performance was measured with an *NVIDIA RTX 4090* GPU and an *Intel Core i9-13900K* CPU running at 3.00 GHz.

Abstract

In this paper, we propose the first approach for real-time extraction and rendering of isosurfaces from scalar volumes as non-trivial persistent meshlets, which are compatible with efficient mesh shader pipelines supported by recent GPUs. Whereas trivial approaches extract from small voxel blocks to ensure that meshlets adhere to geometry size limits, our method partitions and extracts from larger blocks, increasing the number of primitives per meshlet for improved rendering efficiency. Our isosurface extraction method performs fine-grained region analysis and adaptive partitioning of occupied blocks in a task shader stage, followed by efficient meshlet extraction in a mesh shader stage. The region analysis kernel provides the meshlet extraction kernel with compact occupancy information that allows computational resources to be focused on cells that contribute to the isosurface. An extensive analysis reveals that, compared to the state-of-the-art technique, our approach significantly improves extraction and rendering performance and reduces the memory footprint of extracted isosurfaces in most cases.

CCS Concepts
• ***Computing methodologies*** → *Massively parallel algorithms;*

1. Introduction

The highly responsive visualization of implicitly defined isosurfaces in scalar volume data is essential for interactive analysis of structures contained in such datasets. In real-time rendering contexts, two principal classes of approaches for isosurface visualization are commonly employed. Direct methods, such as isosurface ray casting (e.g., [RGW*03, KW03]), directly evaluate the underlying scalar field during rendering. Indirect methods extract an explicit and persistent representation of the isosurface (e.g., [LC87]), which can be rendered using standard geometry pipelines.

A persistent representation offers several notable advantages. It enables efficient reuse of the geometry corresponding to the current isovalue across multiple frames, viewpoints, and shader passes and it supports interactive workflows in which users continuously adjust

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.
This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

the isovalue to explore salient features of the scalar field, often as a precursor to more computationally demanding visualization techniques [KPH*03, ŠLMB24]. In such scenarios, surface extraction must be performed with minimal latency to ensure a seamless user experience, while the resulting geometry is well suited for repeated rendering and downstream processing.

With this paper, we present the first method for real-time, meshlet-based isosurface extraction from scalar volume data using the recently introduced task and mesh shader rendering workflows. Our approach performs a fine-grained region analysis in the task shader stage, which adaptively partitions the volume into subregions suitable for meshlet generation. A novel block-based analysis kernel characterizes the isosurface geometry at a subblock level and derives subregions whose meshlets approach optimal vertex and index counts while satisfying strict meshlet size constraints. The resulting dynamic partitioning provides both occupancy information and primitive counts to the mesh shader stage, where an efficient extraction kernel generates the meshlets and employs a state-of-the-art technique [LCDW16] to prevent duplicate vertex creation. The extracted meshlets are written to global memory for subsequent rendering using standard mesh shader pipelines.

The contributions described in our work are the following:

- A *task shader* based region analysis stage that hierarchically partitions blocks for extraction and produces fine-grained occupancy information, allowing real-time extraction of non-trivial size-constrained meshlets across all isovalues.
- An extension of the state-of-the-art *Parallel Marching Blocks* [LCDW16] isosurface extraction algorithm to extract meshlet geometry, descriptors, and bounding volumes in a *mesh shader* kernel while working only on occupied cells to reduce thread-level divergence.
- An evaluation of our proposed approach and appropriate online and offline reference approaches with respect to the memory footprint, extraction performance, as well as combined extraction and rendering performance for our dynamic isosurface exploration scenario.
- Our polished codebase, provided as an open-source repository in the supplementary material, to foster further research in the domain of real-time meshlet extraction and related areas.

Our evaluation reveals that our real-time meshlet-based isosurface extraction and rendering technique outperforms the state-of-the-art *Parallel Marching Blocks* extraction approach by *12% to 27%* for a variety of models and isovalue ranges. Furthermore, we show that our extracted meshlet representation results in a rendering time that is only 20% higher on average for large volumes than an offline-optimized meshlet representation, despite performing meshlet creation three orders of magnitude faster.

Notes on Terminology. In this paper, we use terminology related to *NVIDIA* hardware. Our approach is also valid for GPUs from other vendors that support mesh shader pipelines.

2. Related Work

This section provides an overview of related work on novel uses of mesh shader pipelines and GPU-based isosurface extraction.

2.1. Meshlets and Mesh Shader Pipelines

Following the definition of Kubisch [Kub18], the term *meshlet* in the context of state-of-the-art graphics APIs refers to a geometric representation consisting of vertices and primitives, often representing triangles [KRF22, NF24], which adhere to strict vertex and primitive size limits. Starting with the *Turing* generation of *NVIDIA* graphics cards and equivalent models from other manufacturers, *mesh shader pipelines* [Kub18] were introduced to efficiently process surface geometry encoded as meshlets. In those pipelines, two tightly coupled and freely programmable compute shader-like stages, *task shaders* and *mesh shaders*, replace the classic geometry transformation shader stages.

Task shaders can spawn multiple mesh shader workgroups, passing information to those workgroups through fast on-chip memory. This allows task shaders to allocate compute power efficiently by dynamically distributing work to mesh shader workgroups. During forward rendering of meshlets, on-the-fly meshlet culling [Wih16, AMHH*18, Kub18, Kap25b] is implemented by spawning mesh shader workgroups only for meshlets in the *potentially visible set* [Air91, COCSD03].

In the mesh shader stage, a cooperative thread model within workgroups [Kub18] eliminates the need to repeat vertex processing calculations, bringing a further advantage over vertex shader pipelines. Even forward renderers implemented purely using mesh shaders, which do not benefit from culling, can double the rendering performance with respect to equivalent vertex shader pipelines [Kub20]. While *offline*-created meshlets have been shown to result in excellent rendering performance (e.g., for terrain rendering [SAW20, Eng20] and skinned character animation [UKPW21]), *real-time* meshlet extraction for isosurface visualization has not yet been explored, motivating the work reported in this paper.

2.2. Efficient GPU-based Isosurface Extraction

The first widely known approach for general isosurface polygonization in a 3D scalar volume is Wyvill et al.'s *Continuation Method* [WCB86]. The method reduces the search space significantly compared to brute-force intersection testing by tracing isosurface intersections through neighboring cells. The *Marching Cubes* (MC) algorithm [LC87] and its many extensions (summarized in a 2006 survey [NY06]) search for isosurface intersections in each cell independently. This approach maps well to execution on massively parallel GPU cores. *NVIDIA* [NVI15] presented the first non-trivial parallel implementation of MC, which identifies cells that are intersected by the isosurface, before applying stream compaction through a prefix sum [HSO07] to focus extraction kernel computation on those “occupied” cells. The efficiency of the method comes at the price of allocating memory to store the occupancy, vertex count and primitive count for each cell, which severely limits application to larger volumes.

Ziegler et al. [ZTTS06] introduce an acceleration structure called the *Histogram Pyramid* (HistoPyramid) which allows efficient per-thread traversal through an occupancy hierarchy. The concept was later extended by Dyken et al. [DZTS08] for more efficient stream compaction and expansion in the context of MC-based isosurface

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

extraction, outperforming the *NVIDIA* implementation. HistoPyramids allow both *persistent* geometry extraction, by writing the stream-compacted geometry in a global memory buffer, and usage with pure vertex and geometry shaders to perform *transient* extraction of the MC-based geometry for direct rasterization of a scalar volume's isosurface. Liu et al. [LCD09] followed the idea of transient extraction and proposed an algorithm for direct rasterization of occupied cells, which can be sorted to support view-dependent culling through early depth testing of hidden geometry.

Schroeder et al. [SMG15] targeted faster persistent extraction of isosurfaces by processing rows of consecutive cells with each thread, reducing the amount of redundant work performed, since neighboring cells share vertices. Liu et al. [LCDW16] identified several issues with *NVIDIA*'s method and HistoPyramids: the severe memory overhead for occupancy analysis, strong read-write latency due to several explicit compute kernel stages, and suboptimal memory access patterns. They proposed *Parallel Marching Blocks* (*PMB*), which also avoids redundant vertex extraction within occupied blocks by creating a unique vertex per intersected edge. Buffer locations of the created vertices are shared among threads in a workgroup. This mostly avoids global storage of auxiliary information, so it requires significantly less off-chip memory than previous approaches [NVI15, DZTS08], and results in up to 10 times higher extraction performance and the ability to process larger volumes. We acknowledge *PMB* as the state-of-the-art method in block-based extraction scenarios and use it both as a basis for our novel real-time meshlet extraction approach and as a reference for the evaluation of our work.

With the advent of mesh shader pipelines (Section 2.1), Persson [Per19] explored transient isosurface extraction using task shaders to cull unoccupied blocks before extraction, but found compute shaders faster in this setting because the resulting meshlets were extracted from individual marching cubes cells only and were consequently very small. Kreskowski et al. [KRF22] improved this by using task shaders to locate occupied cells and mesh shaders to extract geometry from groups of occupied cells, yielding larger meshlets and performance competitive with ray marching and even surpassing it at higher rendering resolutions. Nishidate and Fujishiro [NF24] further showed that tracking unique cell edges within a block lets mesh shaders reuse shared vertices, increasing primitive output before hitting meshlet vertex limits, however, they enforced small, fixed-sized extraction blocks to satisfy meshlet size constraints.

Our work builds on the insight contributed by Kreskowski et al. [KRF22] that task and mesh shader pipelines can enable efficient isosurface extraction by dynamically allocating extraction work after region analysis. However, since our proposed approach has the purpose of extracting an isosurface as *persistent* meshlets, we take care that the resulting meshlets allow efficient rendering of the isosurface. To this end, we adopt *PMB* for isosurface extraction [LCDW16], thus avoiding redundant vertices within meshlets, and contribute a novel region analysis and partitioning approach, which (unlike [NF24]) hierarchically assigns subblocks of varying size without producing meshlets that exceed size constraints.

3. Real-Time Meshlet-based Isosurface Extraction

We introduce the first approach to real-time extraction of isosurfaces as meshlets from scalar volumes. The implementation of our pipeline (shown in Figure 2) is described in this section.

To maximize meshlet size within the given geometry count constraints, we extract a set of unique vertices for each meshlet, such that meshlets can encompass more primitives with the same number of vertices. We employ and extend the extraction kernel of the *PMB* algorithm, which efficiently extracts isosurfaces from a block of cells without duplicating vertices that are shared by isosurface geometry from neighboring cells. The isosurfaces extracted by standard PMB and by our approach are therefore equivalent and visually as well as numerically indistinguishable from each other. While standard *PMB* extracts meshlets from blocks of fixed size, potentially producing meshlets that do not conform to the desired size constraints, our approach addresses this by hierarchically partitioning blocks into subblocks from which meshlets are then extracted. Since the partitioning algorithm is informed by the behavior of the subsequent extraction algorithm, we first give a concise overview of *PMB* (Section 3.1) before detailing our extended approach (Section 3.2).

3.1. Overview of *Parallel Marching Blocks (PMB)*

PMB is a block-based isosurface extraction algorithm that consists of an offline preprocessing stage, which creates a min-max volume based on a scalar input volume V, and two runtime stages. The runtime components are conceptually the same as those comprising our *Extraction Pipeline* shown in Figure 2. The *Occupied Block Filtering* stage accesses the min-max volume to determine whether blocks of $B_x \times B_y \times B_z$ cells contain the isosurface. Through stream compaction, the indices of occupied blocks are stored in a dense global memory array. Subsequently, the *Isosurface Extraction* stage launches a compute shader workgroup for each occupied block.

PMB Isosurface Extraction. The *PMB* extraction kernel avoids duplication of vertices within a block by mapping each thread to one cell and instructing threads to extract vertices that lie on the three edges of a cell for which *only that thread* is responsible (see Figure 3). When building triangles, cells need to reference vertices from their neighbors. To do this, a shared-memory array is allocated to store the location of each cell's vertices in the block's dense vertex array. Threads calculate the location of their cell's vertices through prefix sum operations. These indices are retrieved when creating triangles, allowing generation of an index buffer for the block without duplicated vertices. Note that triangles from cells on the edge of the block require vertices from cells that are not in the block (see vertices V_6 and V_7 in Figure 3). The authors of *PMB* suggest in the paper's supplementary material to launch a workgroup of size $(B_x+1)\times(B_y+1)\times(B_z+1)$, with each thread creating vertices from edges assigned to either a cell within the block (*core* cell) or one of the neighboring cells (*padding* cells, colored blue in Figure 3). In this way, vertices are available for all triangles that can be created inside the block. For more details on the original *PMB* algorithm, we refer the reader to Liu et al.'s original paper [LCDW16].

Limitations of PMB. Although *PMB* avoids extraction of redundant vertex and index data within a block, allowing larger meshlets

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

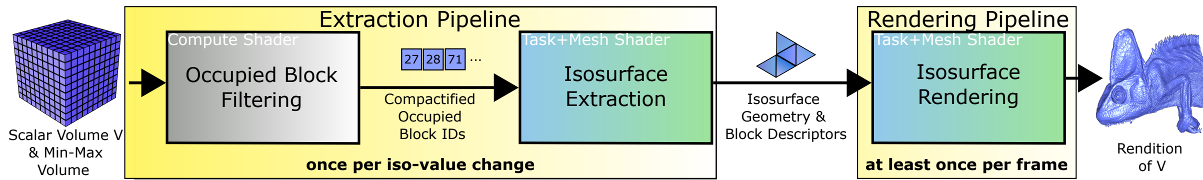


Figure 2: Overview of the run-time components of our real-time indirect isosurface visualization pipeline. The *Occupied Block Filtering* stage identifies occupied blocks based on an auxiliary data structure and is identical to the corresponding stage proposed by Liu et al. [LCDW16]. We contribute the *Isosurface Extraction* and *Isosurface Rendering* stages, which leverage task and mesh shaders to exploit the advantages of on-the-fly extracted meshlets, as explained in Section 3. The ability to extract and render meshlets from scalar volumes offers raw performance benefits in many cases and extends the design space for the core discipline of real-time isosurface extraction. For a more detailed illustration and explanation of our extraction and rendering components we refer the reader to Figure 4 and Sections 3.2 and 3.3, respectively.

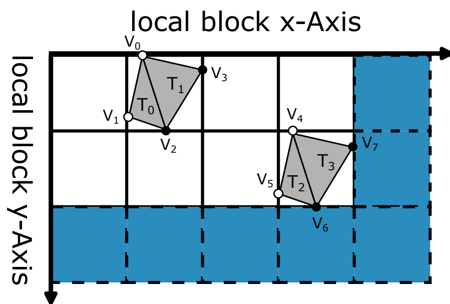


Figure 3: Sketch of *PMB* extraction kernel within a 2D slice of a 3D block adapted from Liu et al. [LCDW16]. Each thread in a workgroup processes either a *core* (white) or *padding* cell (blue). Any cell intersected by the isosurface generates up to three vertices on its uniquely assigned edges. White vertices are extracted from the cells containing the triangles shown, whereas black vertices are created by adjacent cells. Triangles T_0 and T_1 borrow vertices V_2 and V_3 from adjacent core cells, whereas T_2 and T_3 reference vertices V_6 and V_7 extracted from padding cells.

to be created without breaking size constraints imposed by hard technical limitations [Kub18, Kub20] or user-defined optimal sizes, *PMB* provides little control over the maximum number of vertices and indices extracted from a block. While decreasing the block size does limit the maximum meshlet size, this leads to severe degradation in both extraction and rendering performance, because many of the extracted meshlets will be significantly smaller than the optimal meshlet size.

We emphasize that this is not merely a technicality that could be expected to be improved upon in upcoming GPU generations; indeed, research shows that optimal meshlet rendering performance occurs when meshlet sizes are well below the current hardware-based geometry count limitations [Kub18, Kap23]. Our approach, described in Section 3.2, addresses this issue by extracting isosurfaces from hierarchically partitioned subblocks.

3.2. Meshlet Extraction using Mesh Shader Pipelines

In this section, we introduce the first method for real-time extraction of meshlets from scalar volumes. While the first part of the extraction stage of our pipeline follows the same *Occupied Block Filtering* approach as *PMB* to locate occupied blocks where isosurfaces should be extracted (see Figure 2), our *Isosurface Extraction* kernel is implemented as a coupled pair of task and mesh shaders, instead of a compute shader. There are four reasons for replacing the compute shader stage with the recently introduced concept of task and mesh shaders [Kub18]. Firstly, task shaders and mesh shaders are conceptually two freely programmable compute-style shader stages [KBU*18] that can share information via fast on-chip memory. This design enables efficient transfer of occupancy and geometry count data from task to mesh shaders via their lightweight shared interface, whereas an equivalent compute-shader pipeline would rely on slower global memory transactions. Secondly, the tree-expansion capabilities of task shaders make it possible to spawn multiple mesh shader workgroups, allowing us to divide extraction blocks into subblocks and spawn a mesh shader workgroup to extract a meshlet from each one. Thirdly, task and mesh stages execute asynchronously, so mesh workgroups launched by earlier region-analysis invocations can overlap with later analysis work, providing practical pipeline asynchrony without the complexity of multi-queue compute orchestration (e.g., in *Vulkan*). Finally, splitting the original PMB extraction into two tightly coupled kernels reduces thread-level divergence (TLD). The PMB case evaluation is executed for each cell inside a block and therefore typically has low TLD. In contrast, the geometry extraction is only performed on a sparse subset of those cells (the occupied ones), and therefore exhibits higher TLD in the original single compute-kernel design. Our approach ameliorates the extraction TLD by spawning mesh shader workgroups proportional to the number of occupied cells.

In the following, we detail the *Region Analysis* (Section 3.2.1) and *Meshlet Extraction* (Section 3.2.2) stages of our approach.

Preliminaries. We implicitly divide the entire scalar volume V into 3D blocks, such that the block size along each axis $B_{x|y|z}$ is a power of two. This is useful for the efficient computation of Morton code indices, which are used heavily in our approach. We also calculate a min-max volume from V, as is also required for *PMB*.

3.2.1. Task Shader: Region Analysis

Our block-based region analysis kernel subdivides a block of $B_x \times B_y \times B_z$ cells so that meshlets extracted from each resulting subblock do not exceed vertex and index count constraints.

Kreskowski et al. showed that a similar problem could be solved

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

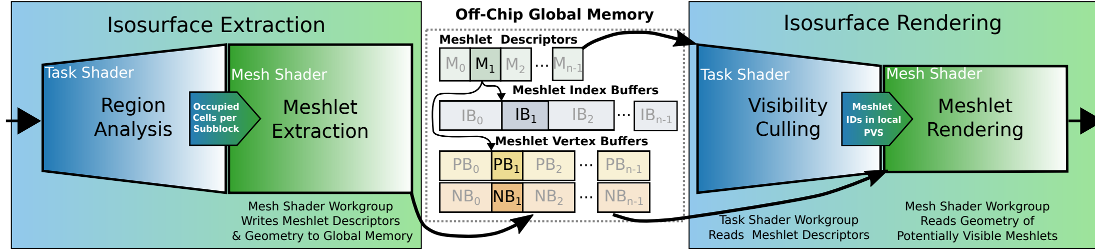


Figure 4: Our meshlet extraction pipeline connected to a typical meshlet rendering stage. Illustration of the extraction and rendering stages of our proposed indirect isosurface visualization approach. The coupled task and mesh shaders on the left-hand side are responsible for extracting large but size-restricted meshlets (see Section 3.2), *meshlet descriptors* ($M_{0..n-1}$), *meshlet index buffers* ($IB_{0..n-1}$) and *meshlet vertex buffers* for both positions ($PB_{0..n-1}$) and normals ($NB_{0..n-1}$), where the same subscript refers to the data used to represent a single meshlet. The coupled task and mesh shaders on the right-hand side represent a conventional meshlet forward-rendering pipeline, which performs meshlet visibility culling in the task shader and passes the information about potentially visible meshlets to the mesh shader stage.

in their work on efficient transient isosurface extraction [KRF22] by calculating the number of vertices produced by each cell in the block and then assigning cells to groups that complied with vertex count constraints. However, our region analysis kernel has the additional constraint that the subsequent meshlet extraction stage, which is based on the *PMB* extraction kernel, works on convex subblocks (such that core and padding cells can be implicitly identified given the size and index of the subblock). Accordingly, our region analysis kernel must divide a block into a set of convex subblocks contained within the initial block. We note that the block size used in our implementation is $8 \times 8 \times 4$. This is a result of combining 32 convex subblocks (one for each thread within a warp) of size $2 \times 2 \times 2$, which trivially guarantee to not exceed our meshlet size constraints regardless of the isovalue and resulting occupancy level.

Region Analysis through Subgroup Communication. Our region analysis task shader program partitions blocks of $8 \times 8 \times 4$ in a hierarchical manner. First, the kernel determines whether all isosurface geometry within the block can be represented by a single meshlet. If this is not the case, it determines whether the isosurface in each of the subblocks of size $4 \times 4 \times 4$ can be represented by a single meshlet. Any of those subblocks that cannot be represented by one meshlet are further partitioned into $2 \times 2 \times 2$ subblocks. A mesh shader workgroup is then launched to extract a meshlet from each of the resulting partitions, except for those that are found to contain no isosurface. Empty subblocks are not considered further.

Each block is analyzed by a single task shader workgroup. The partitioning is determined by analyzing the number of vertices and triangles needed to represent the isosurface in the block and its subblocks. To parallelize this calculation among the task shader workgroup with $T = 32$ threads, each thread counts the number of vertices and primitives produced by a subset of cells in the block. The number of cells analyzed by each thread $R = \beta / T$, where $\beta = B_x \times B_y \times B_z$ is the number of cells in the block (so that $\beta = 256$ and $R = 8$ in our implementation). Cells within the block are indexed in Morton order, with each thread T_i analyzing the ith subregion consisting of R cells. Morton ordering ensures that cells analyzed by individual threads and thread groups (if group size is a power of two) form compact regions rather than rows or slices. This allows us to analyze and summarize the block and all potential subblock regions with efficient subgroup communication [KHG*19] between threads, because all subblocks are analyzed by groups of consecutive threads. Please refer to pseudocode region analysis task shader listing in the supplementary material for more details.

While counting the vertices and triangles that their subregions create, threads also store the indices of cells that contribute to the isosurface. At the same time, the number and indices of the padding cells that contribute to the isosurface within the subregion are stored. This must be determined for each possible partitioning, as padding cell configurations change according to the subblock size (e.g., a cell that is not a padding cell of a $4 \times 4 \times 4$ subblock could be a padding cell of a $2 \times 2 \times 2$ subblock). A cell's status as a core or padding cell for each possible partitioning can be determined using modulo operations, given the cell's position. For a 2D overview of the distribution of padding cells in an extraction block for each partitioning, please refer to Figure 5.

Since the region analysis kernel calculates the number of vertices and indices created by a block, it reserves space in the dense global memory buffers for the meshlet descriptor, indices, and vertex attributes.

In addition, the index of the first meshlet extracted from the block and the number of meshlets the block contains are stored in a dense array aligned with the occupied block indices. This allows groups of meshlets extracted from a block to be culled during the rendering stage.

When determining block partitions, we consider meshlet vertex and index counts as hard constraints. We also introduce a third parameter, C, which defines the maximum number of occupied cells permitted within a subblock. This limit restricts the required size of the memory interface used to pass data between task and mesh shaders (see below) and limits the maximum amount of work that a single thread in an extraction workgroup has to perform.

Workgroup Size and Block Dimensions. The number of cells R analyzed by each thread is determined by the workgroup size T and the number of cells in a block β, which is a function of the block dimensions. Although task and mesh shaders do allow workgroups to consist of several subgroups (i.e., *warps* of 32 threads for NVIDIA GPUs), we found no performance benefit in increasing

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

the workgroup size T beyond that of a single subgroup. We attribute this to the overhead of communicating between subgroups, which involves shared memory and explicit barriers. Furthermore, we found that by allowing each thread to analyze multiple cells, we achieve an acceptable trade-off between extraction block sizes and processing overhead per thread, with experimentation showing that the best extraction performance was achieved if each thread analyzes $R = 8$ cells. For workgroups of $T = 32$ threads, which is the standard subgroup size on NVIDIA GPUs and a configurable size on modern RDNA-based AMD GPUs, we arrive at $\beta = 256$ as the ideal number of cells, which we achieve by setting $8 \times 8 \times 4$ cells as our extraction block size. We note that the smallest subblock size of $2 \times 2 \times 2$ was selected because it coincides with the region analyzed by a single thread if the favored value of $R = 8$ is adopted, and because the maximum number of vertices and indices that a meshlet created from that region could contain (54 vertices and 40 primitives) does not exceed typical size limits.

Task and Mesh Shader Memory Interface. The region analysis task shader stage passes information to its spawned meshlet extraction mesh shader workgroups through an efficient on-chip *memory interface*. To inform the mesh shader workgroups which cells they should extract geometry from, the task shader stage encodes all occupied cell indices in the memory interface, along with an offset to the first occupied cell and number of occupied cells for each subblock (similar to the cell list encoding in [KRF22]).

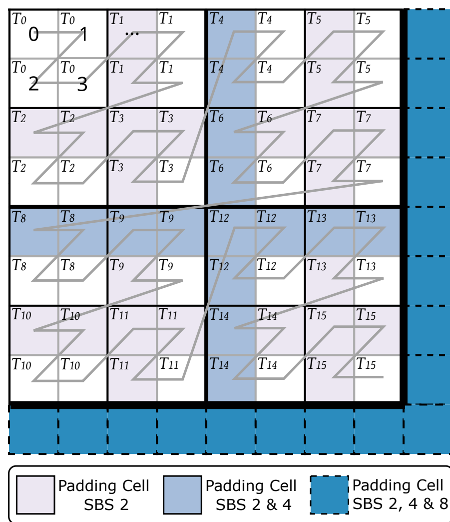


Figure 5: 2D illustration of the per-block thread analysis pattern (gray path) and cell classification across subblock sizes (SBS, shaded cells). In the region-analysis kernel (see Figure 4), subgroup threads $T_{0..31}$ cooperatively process compact Morton-ordered cell groups, accumulating produced vertex and triangle counts as well as recording contributing core and padding-cell indices. In 2D, T_0 analyzes a 2×2 region (cells 0-3), corresponding to a $2 \times 2 \times 2$ region in 3D. The workgroup then hierarchically partitions the block into subblocks whose extracted geometry respects meshlet size limits while avoiding arbitrarily small meshlets. This can produce 8×8, 4×4, and 2×2 subblocks, each requiring the indicated padding cells (dark blue for 8×8; *all* blue for 4×4; *all shaded* for 2×2).

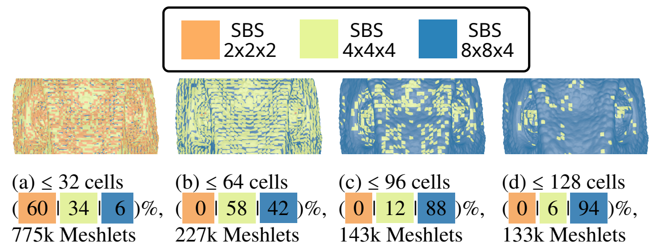


Figure 6: Isosurfaces consisting of meshlets colored according to the subblock size (SBS) that they were extracted from. Increasing the maximum number of occupied cells per meshlet extraction workgroup (a)-(d) increases the proportion of meshlets created from larger SBSs, reducing the overall number of meshlets. Note that larger numbers of occupied cells increase the size of the memory interface and require several extraction iterations for each thread when a single warp is used for extraction, which we found to be beneficial on the test hardware.

3.2.2. Mesh Shader: Meshlet Extraction

Each mesh shader workgroup spawned by the region analysis task shader will extract exactly one meshlet from its assigned subblock. Workgroups read data from the task and mesh memory interface to determine which of the parent block's cells should be used to create meshlet geometry. The ability to focus extraction threads on cells that are known to contain the isosurface, as demonstrated by Kreskowski et al. [KRF22], improves performance by reducing the number of idle threads in an extraction workgroup.

Each mesh shader workgroup extracts a meshlet with a *PMB*-style extraction kernel, avoiding redundant vertices within each meshlet. Since the total number of occupied cells per block C can exceed the total number of threads within a workgroup, we extend *PMB* to allow each thread to process multiple cells within a subblock. This is achieved by separating the vertex extraction and index extraction stages, so that the workgroup first extracts vertices and vertex indices from all occupied cells before using this information to derive the triangle indices. Allowing threads to extract vertices from multiple occupied cells results in larger meshlets (see Figure 6) and fewer mesh shader workgroups being spawned, significantly improving extraction and rendering times (see Figure 7). Although threads could be instructed to extract geometry from any number of cells, no additional benefit was observed with more than 4 cells per thread, i.e., $C = 128$ occupied cells per subblock.

The vertex and index data describing the meshlet extracted by each mesh shader workgroup is written to global vertex and index buffers. To allow the meshlet rendering components to access the correct meshlet geometry information, a meshlet descriptor, which contains offsets to the relevant buffer segments, is also stored (see Figure 4). For pseudocode of the meshlet extraction kernel, please refer to the mesh shader listing in the supplementary material.

Culling Information. Extracting additional information that can be used by the rendering task shader to cull (i.e., avoid reading and rendering) meshlets that do not contribute to the rendered im-

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

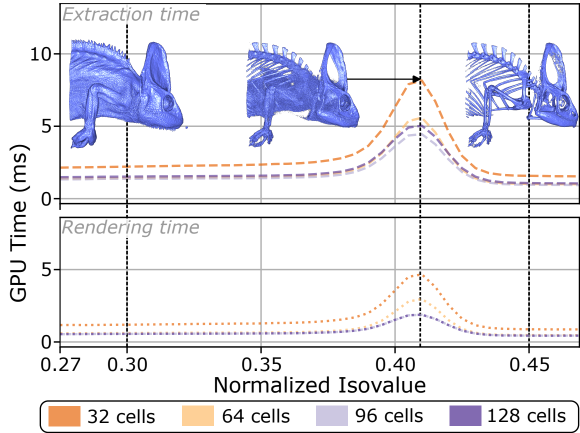


Figure 7: Isosurface extraction (top) and rendering (bottom) times, colored according to the maximum number of occupied cells C handled by a single mesh shader workgroup during extraction, for a representative isovalue range of the *Chameleon* dataset.

age can improve rendering performance. In our tests, we experimented with the creation of different bounding volume representations during meshlet extraction, including implicit and explicit axis-aligned bounding boxes (AABBs), bounding spheres, and normal cones. These approaches impose varying amounts of computational overhead and consequently exhibit drastically different impacts on meshlet extraction performance. We observed that implicit AABBs (defined by the position and size of a meshlet's extraction block) and explicitly computed AABBs provided the best trade-off between extraction overhead and rendering benefit, since they have low creation overhead and result in effective culling because the extracted meshlets are naturally compact. In contrast, bounding sphere and normal cone creation takes a higher toll on extraction performance, while the cones' suitability for cluster backface culling is limited because geometry within a meshlet can exhibit an arbitrary normal spread. Therefore, we do not recommend computing normal cones with our current approach. For additional details and an evaluation of the in-kernel bounding geometry computation, please refer to the supplementary material.

3.3. Meshlet-based Isosurface Rendering

This section summarizes adaptations made to the meshlet forward-rendering pipeline for efficient rendering of the extracted meshlets.

3.3.1. Task Shader: Visibility Culling

Our meshlet extraction pipeline supports optional extraction of per-meshlet culling information (see Section 3.2.2). If no culling information is extracted, the rendering component's task shader stage is omitted, avoiding launch overhead [Kub18]. In this case, no culling will be performed, with all meshlets automatically considered as part of the *potentially visible set* [Air91]. When culling is active, we launch enough task shader workgroups so that each thread checks whether one meshlet should be culled with respect to the viewing frustum. In the case of implicit AABBs based on extraction block position, each thread checks if a group of meshlets extracted from the same block should be culled.

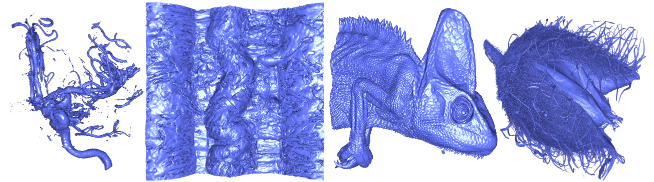
After a task shader workgroup has classified meshlets, the global indices of potentially visible meshlets are stored in the on-chip memory interface (see Figure 4). Then, a mesh shader workgroup is launched for each potentially visible meshlet, as is standard procedure for meshlet rendering pipelines (see e.g., *Niagara* [Kap25b]).

3.3.2. Mesh Shader: Meshlet Rendering

The meshlet rendering stage follows the standard procedure from previous works that render persistent meshlets with mesh shaders [Kub18, Kap25b]. Each mesh shader workgroup processes a meshlet by retrieving geometry buffer offsets and counts based on local meshlet descriptors (see Figure 4) before reading, projecting and writing meshlet vertex information to a local vertex buffer and transferring index data to a local index buffer that defines meshlet topology. The rasterization and fragment shading steps are equivalent to classic rendering pipelines.

4. Evaluation and Discussion

In the following, we evaluate our meshlet-based isosurface extraction approach, comparing our proposed method against PMB and offline isosurface extraction. We analyze the memory footprint of isosurfaces extracted by each approach, as well as extraction performance, rendering times and combined rendering and extraction times in a dynamic isosurface exploration scenario, which is the main use case for our technique.



Name	Voxel Resolution	Bit Depth	Size
Aneurism$_{iso=0.23}$	256^3	8 (uint)	16 MB
Magnetic Reconnection Simulation$_{iso=0.3}$	512^3	32 (float)	0.5 GB
Chameleon$_{iso=0.20}$	$1024^2 \times 1080$	16 (uint)	2.1 GB
Beechnut$_{iso=0.22}$	$1024^2 \times 1546$	16 (uint)	3.0 GB

Table 1: Evaluation datasets in size order. The datasets are scalar volumes containing implicit isosurfaces of varying complexity. The *iso* subscript denotes the isovalue for the depicted isosurfaces.

Datasets. In our evaluation, we render isosurfaces from scalar volume datasets with a range of resolutions and bit depths (see Table 1). The datasets are commonly used in scientific visualization research and can be obtained through the *Open SciVis Dataset* collection [Kla25]. Isosurface geometry extracted in our evaluation always consists of per-vertex positions and normals (represented with 32-bit floating-point precision), as well as per-triangle vertex indices, which vary in bit depth depending on the method.

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

Test System Specifications. Our test machine is equipped with an *NVIDIA RTX 4090* graphics card with 24 GB of video RAM and an *Intel Core i9-13900K CPU* running at 3.00 GHz, with 128 GB of RAM. The entire code base, including the evaluation code, is written in C++. All performance timings were taken with a rendering resolution of 3840×2160 pixels from the viewing perspectives depicted above Table 1. All measurements were repeated 10 times and averaged to reduce variance in the data. Our entire framework and application source code is included with the supplementary material as an open-source *CMake* project to promote reproducibility and encourage further research in the field.

Implementation Details. All online extraction methods and rendering routines are implemented in modern *OpenGL 4.6* using the *GL_NV_mesh_shader* extension, which is available on *NVIDIA* hardware. At the time of writing, the vendor-independent mesh shader extension has recently been made available as an OpenGL extension [Yu25], which would allow our approach to be ported for use on GPUs of other vendors. Nevertheless, we rely on the *GL_NV_mesh_shader* extension for our evaluation, as the more generic version currently only has driver support for AMD GPUs. In addition to our OpenGL-based implementation, we experimented with a *Vulkan* implementation of our approach using the vendor-independent mesh shader extension. The *Vulkan* implementation exhibited similar performance as our OpenGL implementation when tested with the *Chameleon* volume. However, we observed performance degradations in the *Vulkan* implementation when employing workgroups consisting of more than one warp (i.e., $T > 32$). Therefore, we decided to focus on the OpenGL implementation. Throughout our evaluation, data is recorded with a workgroup size $T = 32$, with the maximum number of occupied cells per subblock set to $C = 128$ to maximize meshlet sizes (see Figure 7) while limiting the total number of vertices to 128 (still addressable with 8-bit indices and offering a good rendering and culling trade-off [Kap23]) and 254 triangles.

Real-Time Isosurface Extraction Reference. We compare our approach against Liu et al.'s *Parallel Marching Blocks* (*PMB*) algorithm [LCDW16], which is the state-of-the-art in block-based isosurface extraction. Our compute-shader-based reimplementation is based on pseudocode provided in the publication's supplementary material. Since our proposed method builds on the *PMB* approach, comparing it against our PMB reference implementation allows us to directly measure the performance differences that result from our extraction and rendering adaptations that leverage task and mesh shader pipelines. Our tests showed that no single *PMB* block size performed best for all datasets, so we compare our approach against *PMB* extraction kernels with a range of block sizes (denoted as PMB_{xyz} for block size $B_x \times B_y \times B_z$). We chose a number of block sizes in the vicinity of the optimal size of PMB_{844} reported by Liu et al. [LCDW16]. Depending on the dataset, we observe the best extraction performance between PMB_{444} and PMB_{888}. Compute shaders using block sizes larger than PMB_{888} failed to compile because of workgroup size limitations. Since *PMB* does not provide any control of the number of vertices extracted from a block, vertex indices are stored as 16-bit integers. During rendering, geometry extracted using *PMB* and our approach was drawn using indirect draw calls in a purely GPU-driven manner.

Offline Meshlet Reference. As a second reference, we use an offline approach for the creation of optimized meshlets, based on Kapoulkine's *meshoptimizer* [Kap25a] library which is used in related meshlet research [KRF22, JFB23] and implements linear-complexity algorithms for creating well-optimized meshlets. We target meshlets with a maximum of 128 vertices and 254 primitives, since this allows us to create meshlets that exhibit a reasonable trade-off between vertex transform and culling efficiency (as reported by Kapoulkine [Kap23]), performing well in initial tests. Since *meshoptimizer*'s clustering of triangles into meshlets involves sequential parsing of cache-optimized geometry, the approach takes orders of magnitude longer than the real-time approaches to extract meshlets (see Figure 1). Therefore, we do not include creation times for the offline reference. However, the rendering performance of the offline-created meshlets serves as a reference for the best achievable performance by online methods.

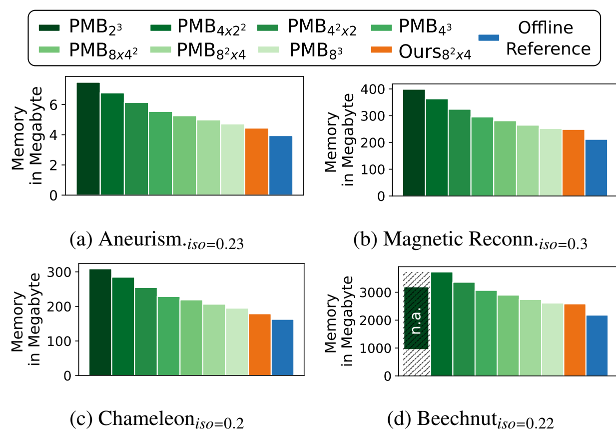


Figure 8: Memory footprint for all models in our dataset. Our online-extracted meshlets (orange bars) exhibit memory footprints that lie between the best *PMB* modes (green bars) and the offline reference (blue bars).

4.1. Memory Footprint of Isosurfaces

One of the main advantages of compact, size-restricted meshlets over arbitrarily large block-based isosurface patches created by *PMB* is that vertex data within a meshlet can be indexed using 8-bit indices, reducing the memory footprint of the index buffer. While our approach does add memory overhead for block data, meshlet descriptors, and culling information, this accounts for only 16-50 bytes per block or meshlet, depending on the mode, which does not significantly increase the memory footprint currently dominated by the 32-bit floating-point vertex attributes.

To evaluate the memory reduction with respect to *PMB*, we recorded the persistent memory footprint of the isosurfaces shown in Table 1 (see Figure 8). We find that meshlet-based isosurfaces generated by our approach have a memory footprint that is on average 4.5% smaller than the nearest *PMB* reference.

Our approach results in geometry that is on average 12.6% larger than the offline reference, which produces optimal meshlets. For isosurfaces with little turbulence or noise, such as

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

Chameleon$_{iso=0.2}$, our approach improves on *PMB* by just under 10%, and is less than 10% larger than the offline reference. For noisier isosurfaces (like *Magnetic Reconn.*$_{iso=0.3}$ or *Beechnut*$_{iso=0.22}$) the offline meshlet reference has a clear advantage of around 15% over our method and *PMB*.

We note that decreasing the per-vertex memory footprint by quantization, similar to Usher and Pascucci [UP20], would increase the advantage of our approach and the offline reference over PMB modes, because the index buffer would become a more dominant component of the memory footprint.

While we restrict our evaluation to datasets that fit onto a single GPU while leaving ample space for the extracted isosurface representations, volumes which clearly exceed available memory resources on GPU hardware could be processed by running our approach in a multi-GPU fashion by subdividing the volumes. Our approach naturally handles subdivision because of the block-based nature of the approach.

4.2. Extraction Performance

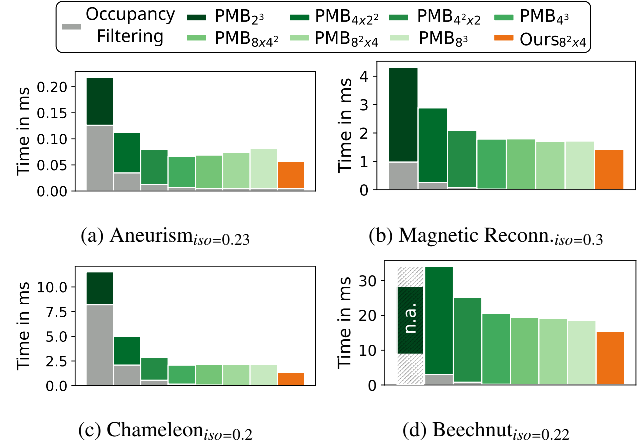
In this section, we evaluate the performance of the isosurface extraction stage of our approach, which encompasses the *Occupied Block Filtering* and *Isosurface Extraction* components (see Figure 2). We measure extraction time (i.e., the time taken to produce the persistent geometric surface representation) and compare our approach with extraction times achieved by the real-time reference (*PMB*). Results are shown in Figure 9.

When comparing our approach with the fastest *PMB* block size for each model, we achieve a relative extraction time improvement of 20.8% (averaged across models). The smallest speed-up for the evaluated isovalues was 14.0% for the *Aneurism* dataset and the highest improvement was achieved for the *Chameleon* dataset (36.1%). We note that the execution time of the occupied block filtering stage, required for all methods, varies drastically depending on the size of the extraction block, because the size of the min-max volume used to filter the blocks, as well as the compactified buffer containing block indices, increases as block sizes decrease.

For the *Beechnut* dataset, we were unable to perform measurements for *PMB* block size of 2^3, because the conservatively allocated extraction buffer sizes, in combination with volume and auxiliary memory blocks, exhausted the GPU memory. In any case, the high overhead of the occupancy filtering stage, caused by a large number of extraction blocks for PMB_{222}, in combination with reduced extraction performance for small block sizes suggests that this extraction mode is expected to exhibit low performance.

4.3. Dynamic Isovalue Exploration Performance

In the final part of our evaluation, we compare the combined real-time extraction and rendering performance of our approach with the reference approaches to gain an understanding of the benefits and limitations of our approach in a scenario where the isovalue changes frequently, such as during exploration of isosurfaces within a volume. We simulate a scenario in which the user dynamically adjusts the isovalue to explore the different isosurfaces of a volume by programmatically increasing the isovalue at each frame



(a) Aneurism$_{iso=0.23}$ (b) Magnetic Reconn.$_{iso=0.3}$

(c) Chameleon$_{iso=0.2}$ (d) Beechnut$_{iso=0.22}$

Figure 9: Comparison of extraction times (green and orange) between *Parallel Marching Blocks* modes (*PMB*$_{xyz}$) based on compute shaders and our proposed meshlet extraction approach for all models in our dataset. *Ours* significantly decreases isosurface extraction time with respect to the most efficient *PMB* mode. All compared methods also rely on occupancy filtering (grey), which strongly varies in duration depending on block sizes and consequently number of extraction blocks. The offline reference method is not included in the comparison because its meshlet creation times are several orders of magnitude longer than the real-time methods.

across a representative range of isovalues. Results are shown in Figure 10. We excluded the *Beechnut* dataset because the range of isovalues that produce relevant surfaces is very limited. To avoid artificially accelerating our approach compared to the offline reference by activating culling, we ensure that the extracted isosurfaces are completely contained within the viewing frustum. The measurements shown in the graph report the time required to extract *and* draw the isosurface with solid lines, with pure draw times shown as dashed lines.

Our evaluation reveals that the combined frame time of our approach is significantly faster than that achieved by the fastest *PMB* mode, outperforming it by 20.3% on average across all datasets and isovalue ranges. Our approach performs best for *Chameleon*, exhibiting a 27.4% performance increase, and worst for *Magnetic Reconnection Simulation*, where we achieve 11.6% performance improvement with respect to the fastest *PMB* mode. The lower performance improvement of our meshlet extraction approach for the *Magnetic Reconnection Simulation* dataset is largely explained by the strong turbulence which results in high occupancy of the extraction blocks, for which our approach tends to create a higher proportion of smaller meshlets than when extracting isosurfaces from organic datasets such as *Aneurism* and *Chameleon*.

Our approach results in 20% higher pure draw times than the offline reference throughout the evaluation and performs similarly to the best PMB mode. Enabling backface culling for the offline reference improved its rendering performance only by 5 to 10%, due to the task shader launch overhead and limited culling capabilities for full-screen views. We expect that the performance gap between our approach and the offline reference could be decreased by combin-

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

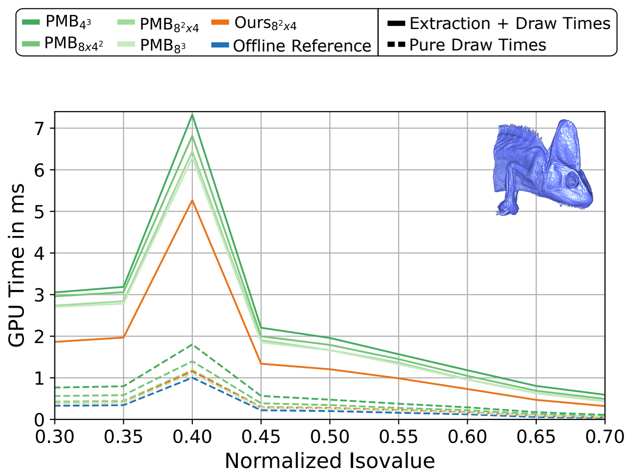


Figure 10: Combined extraction and draw times as they are encountered in a dynamic isosurface exploration scenario for the proposed meshlet extraction and rendering approach (*Ours*) and *Chameleon* (see the supplementary material graphs for the other volumes), compared to the extraction and draw times of the four most competitive non-meshlet extraction modes of *PMB* and the draw times of the offline-optimized meshlet reference, which serves as a best-case reference where no time is spent on extraction during runtime. In addition to the combined extraction and draw times (solid lines), we also plot the corresponding pure draw times (dashes) of the online extraction approaches.

ing our approach with hierarchical z-buffer-based culling [GKM93] in the task shader stage, similar to Kapoulkine [Kap25b], as occlusion culling is compatible with our compact meshlet representation.

5. Conclusion

We presented the first approach that leverages modern mesh shader pipelines for both real-time extraction *and* rendering of isosurfaces from scalar volumes. Our method extracts isosurface geometry as meshlets, allowing efficient rendering of geometry with task and mesh shaders, while producing meshlet sizes that allow 8-bit indices. We extend the state-of-the-art isosurface extraction kernel presented by Liu et al. [LCDW16], achieving superior performance by focusing extraction computations on cells that are known to produce geometry, similar to Kreskowski et al. [KRF22]. We show that, despite not attaining exact control over meshlet sizes, our method reduces extraction times across all evaluation datasets, achieving performance improvements of up to 36.1% and 20% on average across datasets. The combined extraction and draw times achieved by our approach also outperform the state-of-the-art real-time approach by 11.6% in the worst case and 27.4% in the best case, making it highly suited to interactive volume exploration scenarios. Compared to offline-created meshlets, rendering performance is on average 20% lower across the tested volumes and similar to the best *PMB* modes. However, even the fast offline extraction and meshlet creation method used as a reference [Kap25a] in our evaluation is three orders of magnitude slower than our real-time approach, while alternative meshlet creation strategies such as k-medoids-based clustering [JFB23] incur additional processing overhead of at least two further orders of magnitude.

Our approach works in a GPU-driven manner, completely avoiding CPU readback. Therefore, our real-time meshlet extraction is suited for low-latency and high-performance indirect isosurface rendering, and should be considered a drop-in replacement for existing extraction kernels and rendering approaches if mesh shader extensions are available. This becomes especially relevant given that mesh shader pipelines are now also supported by modern mobile GPUs such as the Adreno A8x architecture [QT25], allowing the potential of efficient culling and rendering of compact meshlet data to be exploited on a wide range of devices.

Limitations and Future Work. The application of mesh shader pipelines for meshlet-based isosurface extraction introduces a new set of paradigms for the extraction of isosurfaces as indexed triangle meshes because of the compute-shader-style rendering process and the cooperative thread groups which are employed in both task and mesh shaders. Although the presented approach achieves non-trivial control over meshlet sizes, it does not provide precise control over meshlet vertex and primitive counts as it operates at a per-block granularity. As a result, our approach extracts two to three times more meshlets than the offline-optimized reference. Starting from larger block sizes and falling back to smaller ones if meshlets exceed geometry size constraints could lead to fewer meshlets overall.

Furthermore, we see potential for on-the-fly compression of real-time extracted geometry, similar to the offline compression approach presented by Mlakar et al. [MSS24] for compression-domain meshlet rendering, or more lightweight approaches such as adaptive bit-rate coding as shown by Schütz et al. [SKW22]. As a starting point for future research on the use of mesh shader pipelines for isosurface extraction and rendering, we provide our implementation as an open-source framework in the supplementary material. For a maintainable version of this framework, please refer to the GitHub repository linked in the README.md file included with the supplementary material.

Acknowledgements

We would like to thank the following institutes and people for their dataset courtesy: *volvis.org* and Philips Research, Hamburg, Germany (*Aneurism*); Bill Daughton (LANL) and Berk Geveci (KitWare) (*Magnetic Reconnection Simulation* [GLDL14]), Digital Morphology Project at the University of Texas at Austin, USA (*Chameleon*) and The Computer-Assisted Paleoanthropology group and the Visualization and MultiMedia Lab at University of Zurich (UZH) (*Beechnut*).

Our research is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under the project ID 444831328, SPP2236 - AUDICTIVE - Auditory Cognition in Interactive Virtual Environments. We thank the members of the Virtual Reality and Visualization Research Group at Bauhaus-Universität Weimar (http://www.uni-weimar.de/vr) for their support and the reviewers of this paper for their constructive feedback that helped to improve the paper significantly.

© 2026 The Author(s).
Proceedings published by Eurographics - The European Association for Computer Graphics.

References

[Air91] AIREY J. M.: *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, 1991. 2, 7

[AMHH*18] AKENINE-MÖLLER T., HAINES E., HOFFMAN N., PESCE A., IWANICKI M., HILLAIRE S.: *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, 2018. 2

[COCSD03] COHEN-OR D., CHRYSANTHOU Y., SILVA C., DURAND F.: A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (2003), 412–431. 2

[DZTS08] DYKEN C., ZIEGLER G., THEOBALT C., SEIDEL H.-P.: High-speed marching cubes using histopyramids. *Computer Graphics Forum 27*, 8 (2008), 2028–2039. 2, 3

[Eng20] ENGLERT M.: Using mesh shaders for continuous level-of-detail terrain rendering. In *ACM SIGGRAPH 2020 Talks* (2020). 2

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical z-buffer visibility. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques* (1993), p. 231–238. 10

[GLDL14] GUO F., LI H., DAUGHTON W., LIU Y.-H.: Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection. *Phys. Rev. Lett. 113* (oct 2014), 155005. 10

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with cuda. In *GPU Gems 3*. Addison-Wesley Professional, 2007, ch. 39, p. 851–873. 2

[JFB23] JENSEN M. B., FRISVAD J. R., BÆRENTZEN J. A.: Performance comparison of meshlet generation strategies. *Journal of Computer Graphics Techniques (JCGT) 12*, 2 (12 2023), 1–27. 8, 10

[Kap23] KAPOULKINE A.: Meshlet size tradeoffs, 2023. URL: https://zeux.io/2023/01/16/meshlet-size-tradeoffs/. 4, 8

[Kap25a] KAPOULKINE A.: meshoptimizer, 2025. URL: https://github.com/zeux/meshoptimizer. 1, 8, 10

[Kap25b] KAPOULKINE A.: Niagara task and mesh shaders, 2025. URL: https://github.com/zeux/niagara. 2, 7, 10

[KBU*18] KUBISCH C., BROWN P., URALSKY Y., SMITH T., KNOWLES P.: NV_mesh_shader extension specification for OpenGL, 2018. URL: https://registry.khronos.org/OpenGL/extensions/NV/NV_mesh_shader.txt. 4

[KHG*19] KOCH D., HENNING N., GLANVILLE J., FREDRIKSEN J.-H., LEESE G., HALL J.: KHR_shader_subgroup specification for OpenGL, 2019. URL: https://registry.khronos.org/OpenGL/extensions/KHR/KHR_shader_subgroup.txt. 5

[Kla25] KLACANSKY P.: Open SciVis Dataset, 2025. URL: http://klacansky.com/open-scivis-datasets/. 7

[KPH*03] KNISS J., PREMOZE S., HANSEN C., SHIRLEY P., MCPHERSON A.: A model for volume lighting and modeling. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (2003), 150–162. 2

[KRF22] KRESKOWSKI A., RENDLE G., FROEHLICH B.: Efficient direct isosurface rasterization of scalar volumes. *Computer Graphics Forum 41*, 7 (2022), 215–226. 2, 3, 5, 6, 8, 10

[Kub18] KUBISCH C.: Introduction to Turing Mesh Shaders, 2018. URL: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/. 2, 4, 7

[Kub20] KUBISCH C.: Using Mesh Shaders for Professional Graphics, 2020. URL: https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/. 2, 4

[KW03] KRÜGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), p. 38. 1

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques* (1987), p. 163–169. 1, 2

[LCD09] LIU B., CLAPWORTHY G. J., DONG F.: Fast isosurface rendering on a gpu by cell rasterization. *Computer Graphics Forum 28*, 8 (2009), 2151–2164. 3

[LCDW16] LIU B., CLAPWORTHY G. J., DONG F., WU E.: Parallel marching blocks: A practical isosurfacing algorithm for large data on many-core architectures. *Computer Graphics Forum 35*, 3 (2016), 211–220. 1, 2, 3, 4, 8, 10

[MSS24] MLAKAR D., STEINBERGER M., SCHMALSTIEG D.: End-to-end compressed meshlet rendering. *Computer Graphics Forum 43*, 1 (2024), e15002. 10

[NF24] NISHIDATE Y., FUJISHIRO I.: Efficient particle-based fluid surface reconstruction using mesh shaders and bidirectional two-level grids. *Proc. ACM Comput. Graph. Interact. Tech. 7*, 1 (may 2024). 2, 3

[NVI15] NVIDIA: nvsdk: CUDA samples, 2015. URL: https://github.com/NVIDIA/cuda-samples. 2, 3

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

[Per19] PERSSON E.: Meshlet extraction demo: Metaballs 2, 2019. URL: http://www.humus.name/index.php?page=3D&ID=93. 3

[QT25] QUALCOMM TECHNOLOGIES I.: Qualcomm game developer guide, 2025. URL: https://docs.qualcomm.com/bundle/publicresource/topics/80-78185-2/overview.html#mesh-shading. 10

[RGW*03] RÖTTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart hardware-accelerated volume rendering. In *Proceedings of the Symposium on Data Visualisation 2003* (2003), p. 231–238. 1

[SAW20] SANTERRE B., ABE M., WATANABE T.: Improving GPU real-time wide terrain tessellation using the new mesh shader pipeline. In *2020 Nicograph International (NicoInt)* (2020), pp. 86–89. 2

[SKW22] SCHÜTZ M., KERBL B., WIMMER M.: Software rasterization of 2 billion points in real time. *Proc. ACM Comput. Graph. Interact. Tech. 5*, 3 (July 2022). 10

[ŠLMB24] ŠMAJDEK U., LESAR Ž., MAROLT M., BOHAK C.: Combined volume and surface rendering with global illumination caching. *The Visual Computer 40*, 4 (2024), 2491–2503. 2

[SMG15] SCHROEDER W., MAYNARD R., GEVECI B.: Flying edges: A high-performance scalable isocontouring algorithm. In *2015 IEEE 5th Symposium on Large Data Analysis and Visualization (LDAV)* (2015), pp. 33–40. 3

[UKPW21] UNTERGUGGENBERGER J., KERBL B., PERNSTEINER J., WIMMER M.: Conservative meshlet bounds for robust culling of skinned meshes. *Computer Graphics Forum 40*, 7 (2021), 57–69. 2

[UP20] USHER W., PASCUCCI V.: Interactive visualization of terascale data in the browser: Fact or fiction? In *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)* (2020), pp. 27–36. 9

[WCB86] WYVILL G., CRAIG M., BRIAN W.: Data structure for soft objects. *The Visual Computer 2*, 4 (Aug. 1986), 227–234. 2

[Wih16] WIHLIDAL G.: Optimizing the Graphics Pipeline with Compute. Slideset, 2016. URL: https://archive.org/details/GDC2016Wihlidal. 2

[Yu25] YU Q.: EXT_mesh_shader extension specification for OpenGL, 2025. URL: https://registry.khronos.org/OpenGL/extensions/EXT/EXT_mesh_shader.txt. 8

[ZTTS06] ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.-P.: On-the-fly point clouds through histogram pyramids. In *11th International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV2006)* (Aachen, Germany, 2006), European Association for Computer Graphics (Eurographics), Aka, pp. 137–144. 2

© 2026 The Author(s).