Interactive Isosurface Visualization in Memory Constrained Environments Using Deep Learning and Speculative Raycasting

Landon Dyken[1,2], Will Usher[3], and Sidharth Kumar[2]

***Abstract*—New web technologies have enabled the deployment of powerful GPU-based computational pipelines that run entirely in the web browser, opening a new frontier for accessible scientific visualization applications. However, these new capabilities do not address the memory constraints of lightweight end-user devices encountered when attempting to visualize the massive data sets produced by today's simulations and data acquisition systems. We propose a novel implicit isosurface rendering algorithm for interactive visualization of massive volumes within a small memory footprint. We achieve this by progressively traversing a wavefront of rays through the volume and decompressing blocks of the data on-demand to perform implicit ray-isosurface intersections, displaying intermediate results each pass. We improve the quality of these intermediate results using a pretrained deep neural network that reconstructs the output of early passes, allowing for interactivity with better approximates of the final image. To accelerate rendering and increase GPU utilization, we introduce speculative ray-block intersection into our algorithm, where additional blocks are traversed and intersected speculatively along rays to exploit additional parallelism in the workload. Our algorithm is able to trade-off image quality to greatly decrease rendering time for interactive rendering even on lightweight devices. Our entire pipeline is run in parallel on the GPU to leverage the parallel computing power that is available even on lightweight end-user devices. We compare our algorithm to the state of the art in low-overhead isosurface extraction and demonstrate that it achieves $1.7\times$–$5.7\times$ reductions in memory overhead and up to $8.4\times$ reductions in data decompressed.**

***Index Terms*—Isosurface rendering, GPU raycasting, large-scale data techniques, deep learning.**

I. Introduction

RECENT advances in web technologies, specifically WebGPU and WebAssembly, have enabled the development of powerful GPU-based compute applications that run entirely in the browser. Scientific visualization applications can leverage these technologies to gain the ease of deployment and wide accessibility afforded by the browser without sacrificing the compute capabilities required to perform complex analysis and visualization tasks. Recent works have achieved interactive isosurface extraction on compressed data on the GPU [54], GPU-parallel layout computation of large graphs [9], and multi-channel volume rendering [20].

However, adopting a new technology alone does not address the fundamental issues of limited memory and compute capacity on lightweight end-user devices. Memory capacity constraints are a fundamental issue in scientific visualization even when targeting high-end workstations, and are especially problematic when processing the massive data sets produced by current simulations and data acquisition systems on lightweight consumer GPUs. While there exists a large body of work on large-scale volume rendering approaches [4], deploying such applications in the browser poses its own unique set of additional challenges (see, e.g. [20], [54]). Native applications typically leverage special purpose file formats to stream data from disk (e.g. [7], [12], [16], [17]); however, web applications are unable to perform such low-level I/O. Although prior work has leveraged a server to stream subsets of data [20], [46], [49], this introduces tradeoffs with latency and deployment cost.

Usher and Pascucci [54] recently proposed Block-Compressed Marching Cubes (BCMC) to achieve interactive isosurface extraction in the browser through on the fly decompression and caching of a compressed data set stored on the GPU. They reduce latency by transferring the entire compressed volume to the client, eliminating the need for a complex server, and achieve interactive isosurface extraction times through a fully GPU-driven decompression, caching, and isosurfacing pipeline. However, their approach extracts explicit surface geometry and thus its memory and compute costs scale with the size of the data set and the number of output triangles. BCMC is thus unable to render large data sets on lightweight devices as it runs out of memory to store the vertices.

In this paper, we begin from the on the fly GPU decompression strategy of Usher and Pascucci [54]; however, we make deliberate design choices to reduce memory consumption and the impact of data set size on memory footprint and compute cost. First, we eliminate the need to store a large triangle mesh for the surface by adopting an implicit ray-isosurface intersection approach [33]. Next, to avoid processing fully occluded blocks, we progressively traverse a wavefront of rays through the volume in a multipass approach, decompressing just the visible blocks in each pass. To address utilization issues that would arise when few active rays remain, we introduce ray-block speculation to exploit additional parallelism on the GPU to find intersections. Finally, we leverage a deep neural network to greatly enhance intermediate output from partially rendered images produced by the early passes of our progressive algorithm, allowing for better interactivity and further reduced memory use. Our algorithm can be easily scaled down to run on low power devices, as its costs are primarily tied to the image size. Our contributions are:

- A novel progressive algorithm for implicit isosurface

[1]ldyke@uic.edu
[2]University of Illinois at Chicago
[3]Luminary Cloud

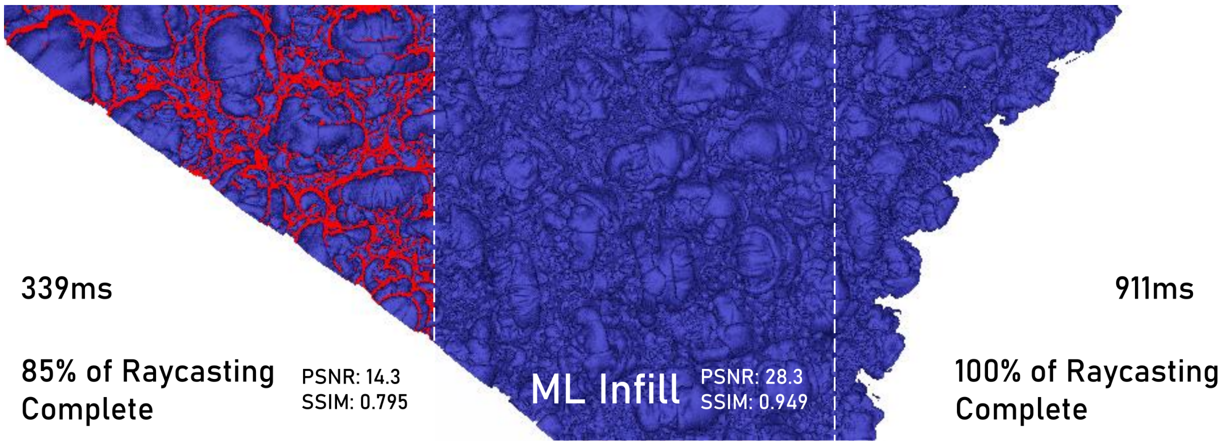


Fig. 1: Isosurface visualization of the $2048 \times 2048 \times 1920$ Richtmyer-Meshkov (R-M) data set in the browser. Our method renders this 32.2GB volume using just 4.2GB of memory. Left: after 85% of rays have completed traversal (active rays colored red); Middle: machine learning infill and reconstruction on the 85% image; Right: ground truth. We propose a new GPU algorithm for implicit isosurface rendering that progressively traverses rays through the volume and decompresses data on-demand to minimize memory requirements. Intermediate results can be drastically improved by reconstruction with our pretrained deep learning network. At 1280×720, the Richtmyer-Meshkov reaches 85% completion in 339ms and 100% completion in 911ms on a laptop RTX 4070. Inference time takes 68ms using ONNX Runtime Web, and only 16ms using TensorRT. We achieve up to 5.7× reductions in overall memory use and 8.4× reductions in data decompressed compared to the state of the art in memory constrained isosurface extraction [54].

raycasting that works directly on compressed data on the GPU;
- A per-pass view-dependent decompression and caching strategy built into the algorithm to minimize its memory footprint;
- A technique for displaying high-quality intermediate results and improving interactivity through deep learning based image in-fill;
- A dynamic work speculation strategy that exploits additional parallelism in the workload to increase GPU utilization and accelerate rendering completion;
- Evaluation of our algorithm against the state of the art on data sets with up to 8.05B voxels on lightweight end user devices.

II. Related Work

In Section II-A, we review recent work on bringing scientific visualization to the browser. Visualizing large-scale volumetric data is a fundamental problem in scientific visualization, and has been deeply explored (see surveys by Beyer et al. [4] and Rodriguez et al. [2]). Isosurface visualization techniques can be categorized as explicit surface extraction methods (Section II-B), or implicit surface rendering methods (Section II-C). Due to the similarities in isosurface ray-casting and ray-guided volume rendering algorithms, we also review relevant work on raycasting large volumes in Section II-D. Finally, we review relevant previous work on applying deep learning techniques for volume visualization in Section II-E.

A. *Scientific Visualization in the Browser*

Bringing scientific visualization to the browser greatly expands accessibility, enabling more scientists to gain better insights about their data. Prior work has brought compelling applications to the browser through the use of server-side processing, local GPU acceleration, and combinations thereof.

Server-based techniques move all computation to the server and stream images to the client [10], [24], [40]–[42], allowing lightweight clients to access large amounts of compute power. However, such approaches can face issues with latency, cost, and quality of service when faced with supporting large numbers of concurrent users. Prior work has demonstrated leveraging a remote server to query and stream subsets of data to the client for rendering [46], [49], thereby balancing between remote and local processing costs. Although moving the rendering work to the client reduces the impact of server latency and quality of service, such approaches can face similar issues as fully server-side approaches at scale.

In this work, we target a fully client-side approach to eliminate the need for backend servers and related challenges. We note that a combination of client- and server-side processing can provide the best scalability and performance for large data visualization; here we focus solely on expanding the capabilities of the client. Prior to WebGPU, browser applications leveraged WebGL to perform GPU accelerated rendering in applications ranging from LiDAR visualization [46] to volume rendering [36] and neuroscience [23], [49]. A fundamental limitation of WebGL compared to WebGPU is the lack of support for general compute shaders; although Li and Ma [27] proposed a method to work around this limitation by repurposing the rendering pipeline to perform a subset of parallel compute operations.

With the recent development of WebGPU, browser applications now have access to general purpose GPU compute and advanced rendering capabilities. Usher and Pascucci [54] leveraged WebGPU to deploy a GPU-driven isosurface extraction pipeline that achieved interactive visualization of massive data sets entirely in the browser. Dyken et al. [9] presented a graph layout algorithm in WebGPU to accelerate layout and rendering of large graphs. Herzberger et al. [20] used WebGPU and WebAssembly to build an efficient multi-channel volume rendering approach with a custom data structure that combines the advantages of page tables and octrees.

B. *Explicit Isosurface Extraction*

Marching Cubes [32] is an object-order technique that computes explicit triangle geometry for each voxel to render the isosurface. The extracted triangle geometry can then be

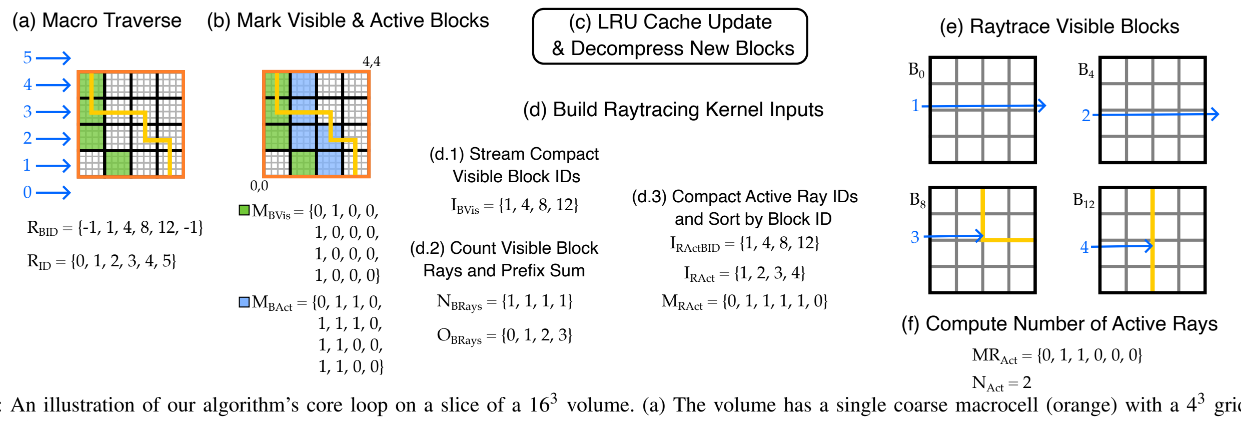

(a) Macro Traverse
5
4
3
2
1
0
R_{BID} = {-1, 1, 4, 8, 12, -1}
R_{ID} = {0, 1, 2, 3, 4, 5}
(b) Mark Visible & Active Blocks
4,4
0,0
🟩 M_{BVis} = {0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0}
🟦 M_{BAct} = {0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0}
(c) LRU Cache Update & Decompress New Blocks
(d) Build Raytracing Kernel Inputs
(d.1) Stream Compact Visible Block IDs
I_{BVis} = {1, 4, 8, 12}
(d.2) Count Visible Block Rays and Prefix Sum
N_{BRays} = {1, 1, 1, 1}
O_{BRays} = {0, 1, 2, 3}
(d.3) Compact Active Ray IDs and Sort by Block ID
$I_{RActBID}$ = {1, 4, 8, 12}
I_{RAct} = {1, 2, 3, 4}
M_{RAct} = {0, 1, 1, 1, 1, 0}
(e) Raytrace Visible Blocks
B_0
1
B_4
2
B_8
3
B_{12}
4
(f) Compute Number of Active Rays
MR_{Act} = {0, 1, 1, 0, 0, 0}
N_{Act} = 2

Fig. 2: An illustration of our algorithm's core loop on a slice of a 16^3 volume. (a) The volume has a single coarse macrocell (orange) with a 4^3 grid of ZFP blocks within it. After computing the initial set of rays we repeat steps (a-f) until all rays have terminated, displaying the partial image after each pass. (a) Rays are advanced to the next active block (green), storing its ID in R_{BID}. Rays one and two traverse blocks whose value range combined with their neighbors contains the isovalue, indicating that their dual grid may contain the isosurface. (b) Blocks in R_{BID} are marked visible and active (M_{BVis}, green), and their neighbors to the positive side marked active (M_{BAct}, blue). The neighbors are required to populate the visible block's dual grid. (c) M_{BAct} is passed to the LRU cache [54], which decompresses and caches any new blocks, potentially evicting those that are no longer needed. (d) We then prepare the inputs for the block raytracing kernel through stream compactions and parallel sorts on the GPU. (e) Each block traverses its rays through its local data, terminating those that intersect the isosurface. (f) Finally, we compute the remaining number of active rays to check completion and display the current image to the user.

rendered interactively. Subsequent work proposed constructing interval trees [5] or k-d trees over the span space [31] to accelerate Marching Cubes by skipping voxels that do not contain the isosurface. Isosurface meshes can contain large numbers of triangles, many of which will be occluded or subpixel for a given viewpoint. Livnat and Hansen [30] proposed a view-dependent technique that traversed an octree to find voxels to extract triangles from. Rendered triangles updated an occlusion buffer used to skip occluded octree nodes. Recent work has focused on leveraging parallel execution on GPUs to accelerate computation [6], [8], [26], [29], [34], [45], [54]. However, prior work has assumed that the entire volume fits in the memory of a single GPU [8], [26], [29], [45].

Usher and Pascucci [54] proposed the Block-Compressed Marching Cubes (BCMC) algorithm for interactive GPU-parallel isosurface extraction on massive data sets. BCMC uploads a ZFP fixed-rate compressed volume to the GPU and decompresses and caches the blocks required for a given isosurface on demand using GPU decompression and an LRU cache. BCMC achieves interactive isosurface extraction times on consumer GPUs; however, as with other surface extraction techniques, it produces large vertex buffers and its cost scales with the total number of blocks containing the isosurface. Although we adopt a similar on-demand decompression and caching strategy, we do not store a vertex buffer and processes blocks in a view-dependent wavefront to reduce memory use and the impact of data size on scalability.

C. Implicit Isosurface Rendering

Parker et al. [38] proposed the first implicit isosurface rendering technique, where rays were traversed through the volume grid, and ray-voxel intersections computed directly by solving a cubic polynomial. Parker et al. [38] accelerated ray-traversal by skipping empty space using a multi-level grid hierarchy. Marmitt et al. [33] improved the quality and speed of ray-voxel intersection through a root finding approach based on isolation and iterative root finding. Wald et al. [55] further accelerated empty space skipping through an implicit k-d tree that tracked value ranges of subregions of the volume.

Hadwiger et al. [18] proposed an implicit isosurface rendering technique that combined object and image order empty space skipping to accelerate rendering, coupled with a brick cache to reduce memory use. Their algorithm constructs a fine grid over the volume and rasterizes the cells that potentially contain the isosurface to generate ray start and end positions, then performs ray marching on the GPU to find intersections in these intervals. Hadwiger et al. employed a brick cache using a coarse grid to reduce memory use, data for a grid cell is only uploaded to the GPU if its value range contains the isovalue. However, this strategy does not take into account visibility, and will upload data for occluded regions of the volume. In contrast, our algorithm performs data decompression on-demand as rays traverse the volume, reducing the working set to just the blocks visible in a single pass. Moreover, our proposed decompression and caching pipeline runs entirely on the GPU, eliminating CPU communication bottlenecks.

D. Ray-guided Large Volume Rendering

A large body of work has explored techniques to address memory constraints in ray-guided volume rendering [4], [7], [11], [12], [16], [17], [20], which we briefly review here due to their applicability to implicit isosurface raycasting. Ray-guided techniques for large volume rendering typically combine GPU-driven cache requests, made as rays encounter missing data during traversal, with a CPU-side data management system that services these requests by uploading new data to the GPU [7], [12], [16], [17], [20]. The CPU-side data management system is typically coupled with a special purpose file format and takes advantage of low-level file system APIs to efficiently stream massive data sets off disk. Prior work has demonstrated interactive rendering of data sets ranging in size from hundreds of gigabytes [12] to terabytes [7], [16], [17].

Volume rendering techniques that operate on compressed data have been proposed to alleviate disk space and in-memory working set requirements [2], [13]–[15], [35], [44], [56]. Schneider and Westermann [44] proposed a hierarchical quantization scheme that decomposes the data into 4^3 blocks and computes a $1/4$ resolution quantized representation of each block along with two codebooks for the volume. Fout et al. [13], [14] proposed a vector quantization technique combined with deferred filtering for two-pass slice-based rendering, using a decompression pass followed by a filtering one. Subsequent works have leveraged compressed GPU textures [35], combinations of bricking, quantization and run-length encoding [56], and extending these techniques with tensor approximations [52] and octrees over compressed blocks [15].

Similar to prior work, we adopt a brick-based compression scheme to allow decompression of spatial subregions on-demand. While we use ZFP [28] in this work, it is possible to leverage other brick-based compression schemes, and to combine our method with multiresolution hierarchies to address undersampling or with out-of-core streaming to support larger data sets.

E. Deep Learning for Volume Visualization

In recent years, deep learning techniques have been successfully used to improve the performance of volume visualization tasks through a myriad of different methods. These methods work to either reduce the number of data samples required for rendering [3], [58], [60], or create simplified volume representations to sample from [59], [62]. As our method reconstructs isosurface images from intermediate rendering results, the reconstruction task we target is most similar to the former methods, and we discuss those in more detail.

Reducing data samples by upscaling isosurface images from low resolution depth and normal field samples was shown by Weiss et al. [58], who proposed a recurrent super-resolution network. Bauer et al. [3] used deep learning based image denoising techniques to reconstruct full volume rendering frames from foveated sparse input, with great performance gain. Weiss et al. [60] show the possibility of a network for learning adaptive sampling and image reconstruction for volume visualization with one joint neural network. While these techniques all involve inferring full-resolution frames of volume datasets, they differ from our target task in that the sampling patterns of the input to the networks are either uniform [58], generated from sampling maps built from noise patterns around a focal point [3], or inferred from a trained importance network [60]. In contrast, our network input is simply the output of early passes of our multipass progressive rendering algorithm, and the sampling pattern is determined solely by which rays happen to terminate in these passes.

III. Progressive Wavefront Isosurface Raycasting

Our algorithm is designed with a focus on reducing overall memory consumption and on achieving scalable and controllable rendering performance that is not strongly impacted by the data set size. These properties enable the algorithm to be used for visualizing massive data sets in the browser on lightweight end user devices. To achieve this, we propose an implicit isosurface raycasting algorithm that progressively traverses a wavefront of rays through a block-compressed volume (Figure 2). In each pass, new visible blocks that potentially contain the isosurface are decompressed and cached in an LRU cache to enable re-use of decompressed blocks across passes. Thus our algorithm's memory footprint and compute cost is dependent on the image size, view position, and isovalue. The progressively rendered image is displayed after each pass to improve interactivity.

At a high-level, our algorithm proceeds as follows. First, volume data compressed using ZFP's [28] fixed-rate compression mode is uploaded to the GPU. We then construct a two-level macrocell grid [38] on the GPU to accelerate ray traversal (Section III-A). For each new isovalue or camera viewpoint, we compute the view rays for each pixel (Section III-B). The following steps are then repeated to traverse the wavefront of rays through the volume to progressively render the isosurface (see Figure 2). First, we traverse the rays through the macrocell grids to find the next block they must test for intersections with (Figure 2a, Section III-C). We then mark all the blocks that are visible or active in the current pass (Figure 2b, Section III-D). The data for uncached blocks are decompressed using a WebGPU port of ZFP's CUDA decompressor, and cached for re-use between passes through a GPU-driven LRU cache, as done by Usher and Pascucci [54] (Figure 2c, Section III-E). We then construct arrays of the visible block IDs, the number of rays intersecting each block, and the ray IDs sorted by their block ID to provide inputs to the block raycasting kernel (Figure 2d, Section III-F). Each block then intersects its rays with its local region of data to find ray-isosurface intersections (Figure 2e, Section III-G). Finally, we compute the remaining number of active rays to determine if rendering has completed (Figure 2f) and display the current image.

A. Macrocell Grid Construction

As done by Usher and Pascucci [54], we leverage the 4^3 block decomposition of the volume used by ZFP's fixed-rate compression mode to define a macrocell grid over the volume. The macrocell grid is used to skip blocks that do not contain the isovalue [38], and thereby skip decompressing them. In addition to the ZFP block macrocell grid, referred to as the fine grid, we compute a coarse macrocell grid by grouping 4^3 regions of ZFP blocks to form coarse cells. Each coarse cell contains 16^3 voxels, allowing larger regions of space to be skipped more efficiently to accelerate the rendering of sparse isosurfaces. The value range of each cell in the fine (or coarse) grid is computed by combining the range of the cell's voxels (or blocks) with those of its neighbors in the $+x/y/z$ direction. The neighbor ranges are required to ensure we do not miss values contained in the cell's dual grid, which would lead to cracks.

When a new volume is loaded, we compute the value range of each block and then combine each cell's range with its neighbors to populate the coarse and fine grids. These computations are run in parallel on the GPU. We note that our approach can be combined with an octree or other hierachical multiresolution acceleration structure over the ZFP blocks for LOD, rather than a two-level grid.

B. Compute Initial Rays

For each new camera position or isovalue, we begin by computing the initial camera rays. This is done through a

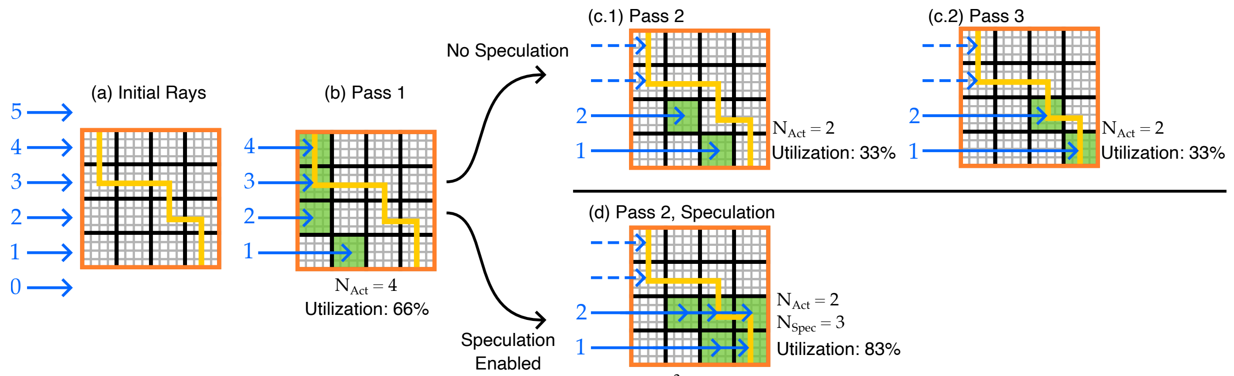


Fig. 3: An illustration of the ray traversal passes for an example isosurface on a 16^3 volume, without (b, c.1, c.2) and with (b, d) speculation. Green squares mark the blocks currently being traversed by a ray. (a) Four of the six initial rays intersect the volume's bounds. (b) Pass one is identical in both cases, as not enough rays have terminated to enable speculation. (c.1, c.2) Without speculation, rays one and two traverse one block at a time until they hit the isosurface, requiring two additional passes with low GPU utilization to complete the rendering. (d) With speculation, enough rays have terminated after pass one that $N_{\text{Spec}} = 3$, increasing utilization to 83% and completing the rendering in one additional pass by intersecting rays one and two against multiple blocks. A trade-off of speculative execution strategies is the potential for wasted computation. This is illustrated by ray two, which traverses an extra occluded block in pass two. Overall, our speculative execution strategy significantly reduces the total number of passes, and thus total time, required to render isosurfaces.

standard GPU volume raycasting approach where the backfaces of the volume's bounding box are rasterized and ray directions computed in the fragment shader [51]. The fragment shader writes the pixel's ray direction and the t value that it enters the volume out to an image-sized ray data buffer, requiring 16 bytes per-ray. Rays that miss the volume are marked as terminated.

C. Macrocell Grid Traversal

Each pass of the wavefront ray traversal begins by finding the next block along the ray that potentially contains the isosurface (Figure 2a). We traverse the two-level macrocell grids using the algorithm of Amanatides and Woo [1], skipping cells whose value range does not contain the isovalue. Rays begin by traversing the coarse grid. When a coarse cell containing the isovalue is encountered, we traverse the 4^3 grid of its blocks to determine if the ray intersects a block containing the isovalue. If such a block is found, we record the block ID for the ray in R_{BID}, save the coarse and fine grid iterator traversal states, and exit the macrocell grid traversal kernel. R_{BID} is an image-sized buffer that stores the block ID each ray intersects, or `UINT_MAX` if none. Rays that exit the volume are marked as terminated. The macrocell grid traversal is run over all $w \times h$ rays; rays that have terminated simply early exit from the kernel.

The grid iterator states are saved and restored between passes to ensure that we do not skip cells due to precision issues that would occur when simply tracking the ray's current t value. Iterator states are stored in an image-sized buffer that tracks $t_{\max}$ and the current cell ID, requiring 16 bytes per-grid for a total of 32 bytes per-ray.

D. Mark Visible and Active Blocks

Next, we determine which blocks need to be decompressed to process ray-block intersections (Figure 2b). A block is marked both visible and active if a ray is traversing it (Figure 2b, green blocks); blocks that are $+x/y/z$ neighbors of visible blocks must also be decompressed to provide data for the visible block's dual grid, and are marked active (Figure 2b, blue blocks). This pass is run on the GPU over the entire R_{BID} buffer, and thus scales with the image size rather than the number of blocks. Kernel invocations for terminated rays simply exit early.

E. GPU-driven LRU Block Cache

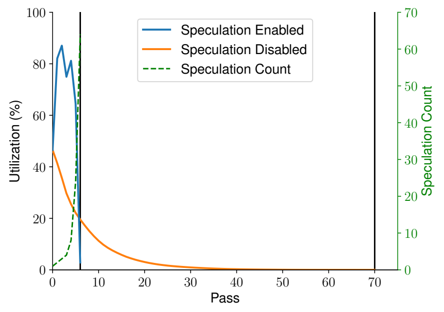
The buffer marking active blocks, M_{BAct}, is passed to the GPU-driven LRU block cache of Usher and Pascucci [54] to produce a list of the new blocks that need to be decompressed and cached for the current pass (Figure 2c). These blocks are decompressed into their assigned cache slots using a WebGPU port [54] of ZFP's [28] CUDA fixed-rate decompression algorithm. Rays are likely to require data from the same blocks traversed in the previous few passes. The data from these blocks will be readily available in the cache, reducing the decompression cost for the pass. Similarly, rays are likely to require data from the same blocks as their neighbors in a given pass. Shared blocks will be decompressed once and cached, amortizing the decompression workload over multiple rays.

Performing the cache update each pass allows us to replace unneeded blocks with new ones each pass, reducing the algorithm's working set to just the active blocks in an individual pass. This is in contrast to surface extraction based methods [54], which decompress and store all the blocks that potentially contain the isosurface at once, regardless of visibility.

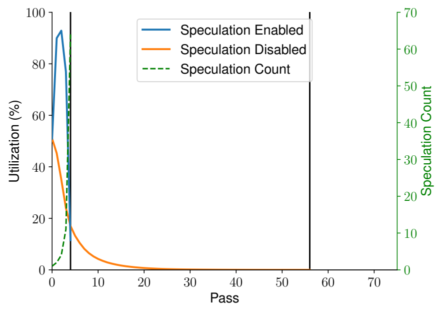
F. Build Raytracing Kernel Inputs

At this point, we have all the volume and ray data required to traverse rays through the blocks they intersect and test for ray-isosurface intersections. However, a large number of rays will likely traverse the same block in each pass. If we were to run the raytracing kernel in parallel over the rays we would waste bandwidth by repeatedly re-loading the same block from memory. Instead, we run the raytracing kernel in parallel over the visible blocks. The raytracing kernel then loads each block's dual grid from memory just once and computes ray-isosurface intersections for the rays passing through it.

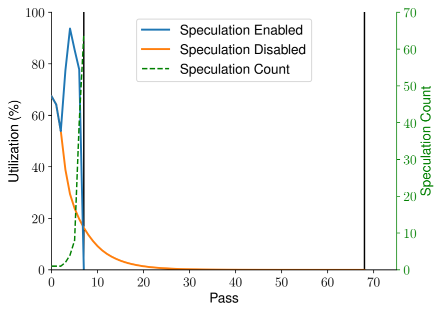
The inputs to the raytracing kernel are the list of visible block IDs (I_{BVis}), the number of rays intersecting each block (N_{BRays}), the offsets to the block's set of rays (O_{BRays}), and the active ray IDs sorted by their block ID (I_{RAct}). These inputs are



(a) Spec. Enabled: 7 passes, 1269ms, Spec. Disabled: 71 passes, 6025ms.



(b) Spec. Enabled: 5 passes, 922ms, Spec. Disabled: 57 passes, 4527ms.



(c) Spec. Enabled: 8 passes, 1466ms, Spec. Disabled: 69 passes, 6892ms.

Fig. 4: Our speculative ray traversal improves GPU utilization to reduce the number of passes needed to render the isosurface by 10× on average, thereby reducing the total time to complete the isosurface by 4.8× on average. Although average time per pass roughly doubles, this is more than made up for by the reduction in the total number of passes required. The vertical black lines mark when the surface was completed for each configuration. The dotted green line shows the speculation count, which is increased as rays terminate to process additional speculated ray-block intersections for the remaining active rays in parallel to terminate them sooner. Timings are reported on an RTX 3080. Datasets shown are a) TACC b) Plasma c) Miranda.

produced through a series of stream compactions, prefix sums, and parallel sorts on the GPU (Figure 2d). The list of visible block IDs, I_{BVis}, is computed via a stream compaction. The number of rays intersecting each block, N_{BRays}, is computed using a kernel run for each ray that atomically increments the block's ray count. The offset to each block's set of ray IDs, O_{BRays}, is computed by perfoming a prefix sum on N_{BRays}. Finally, we compute the list of active ray IDs (I_{RAct}) sorted by their block ID (I_{RActBID}) by compacting the active ray IDs and their block IDs, then performing a parallel sort by key, using the block ID as the key.

G. *Raytracing Visible Blocks*

The raytracing kernel is run in parallel over the visible blocks, and is responsible for taking the set of rays intersecting the block and traversing them through its dual grid to find ray-isosurface intersections (Figure 2e). The kernel consists of two steps: loading the block's dual grid data into shared memory, followed by traversing the rays through the dual grid to compute intersections.

The block's dual grid consists of its local data combined with the face/edge/corner values from its neighbors in the $+x/y/z$ direction, if those neighbors exist. We employ the parallel loading strategy of Usher and Pascucci [54] to load the dual grid data into shared memory. Kernel work groups are launched with 64 threads, corresponding to one thread per dual grid cell, and have a work group shared memory region with room for 5^3 floating point values to store the full set of local and neighbor values for the dual grid. First, the work group loads the 64 vertices corresponding to the block's local 4^3 data into the shared memory region, after which a subset of threads load data from the $+x/y/z$ face, edge, and corner neighbor blocks to complete the dual grid. Finally, the work group synchronizes on a memory barrier to ensure the complete dual grid data is visible to all threads in the group.

With the dual grid loaded into shared memory, we can now traverse rays through it to find ray-isosurface intersections. The 64 threads in the work group are used to process the block's rays in parallel in chunks of 64 rays, with each thread responsible for a different ray in the chunk. We again use the Amanatides and Woo [1] grid traversal algorithm to step rays through the dual grid. Ray-isosurface intersections are computed using the ray-voxel intersection technique of Marmitt et al. [33]. If an intersection is found, the shaded color and depth is output to the ray's pixel in the framebuffer and the ray is marked as terminated.

IV. Increasing GPU Utilization with Speculation

Our algorithm as described in Section III achieves isosurface rendering of massive data sets within a small memory footprint. However, we observed that the algorithm would take a large number of passes to complete the isosurface on average. Each pass incurs some fixed time costs, and this translated into long total surface rendering times. We further observed that, on average, after 10 passes there were $< 20\%$ of rays still active, and that by pass 25 there were $< 1\%$ of rays still active (see Figure 4). These long tail rays are those that just miss the surface and must be traversed through many blocks before finding an intersection or exiting the volume.

To address this issue, we extend our algorithm to enable *speculative* intersection of rays with additional blocks to terminate rays in fewer passes (Figure 3). We treat the various image-sized ray and block data buffers used by our algorithm as a virtual GPU with $w \times h$ threads and memory slots. As rays terminate, these slots become available for other active rays to use for speculation. For simplicity we use a constant speculation count for all rays, defined as $N_{\text{Spec}} = \lfloor \frac{w \times h}{N_{\text{Act}}} \rfloor$, where N_{Act} is the number of active rays. To balance between terminating rays in fewer passes and performing unnecessary computation, we limit the speculation count to a maximum of 64.

The following modifications are made to the algorithm described previously (Section III) to enable speculation. The macrocell grid traversal kernel now advances each ray through N_{Spec} blocks, recording multiple block IDs for each ray (Section IV-A). As the macrocell grid traversal will write out the same ray ID N_{Spec} times in R_{ID}, ray IDs in the buffer are no longer unique identifiers, and we must introduce an additional speculated ray-block offset buffer to the raytracing kernel inputs (Section IV-B). To prevent speculated ray-block intersections from trampling each other's results, the block raytracing kernel is modified to write intersection results out to a new RGBZ buffer instead of directly to the framebuffer (Section IV-C). A new kernel is introduced to select the closest hit found, if any, for a given ray and write the final color to the framebuffer (Section IV-D). At the end of each pass, we keep the prefix sum result buffer O_{Act} that is produced when computing N_{Act} and update N_{Spec}. O_{Act} is used to assign offsets in R_{ID} and R_{BID} to the remaining active rays.

A. Speculative Macrocell Grid Traversal

Our speculative macrocell grid traversal performs the same traversal as before (Section III-C), with the key difference being that it traverses the ray until finding up to N_{Spec} visible blocks instead of just one (see Figure 3), and records all the visible block IDs encountered to be tested for intersections. The set of blocks being traversed by a given ray may be disconnected due to empty space-skipping.

The macrocell grid traversal kernel is run over all $w \times h$ pixels as before, with terminated rays exiting early. The N_{Spec} entries for each active ray are written at offsets given by $o = O_{\text{Act}}[\text{ray}] \times N_{\text{Spec}}$. The visible block IDs for each active ray are written into R_{BID} starting at o, with up to N_{Spec} entries written for each ray. If the ray exits the volume early, its remaining R_{BID} entries are left filled with `UINT_MAX` and filtered out in subsequent passes in the manner as terminated rays. The ray ID buffer, R_{ID}, is populated by writing out N_{Spec} entries of the ray ID starting at o. As before, each ray maintains just one coarse and fine grid iterator state. The iterator states are saved out after N_{Spec} visible blocks have been found, to resume traversal after the last block being intersected in the pass.

As each speculated ray-block intersection writes its block ID to the R_{BID} buffer as before, the mark visible and active blocks kernel does not require modification to support speculation. The kernel is run over the entire R_{BID} buffer and marks blocks active as before, with the only difference being that some visible block IDs in the buffer correspond to speculated ray-block intersections.

We take note that, for sparse volumes and viewpoints of an isosurface outside the surface, most rays terminate in the first macro traverse of our algorithm through only empty space skipping, never intersecting an isovalue-containing block. Because of this, the first pass of our algorithm would not utilize buffer space to check for intersections well. To remedy this, we perform two macro traverse steps in the first pass of the algorithm: the first step only terminates rays through empty space skipping, then we compute N_{Spec} using the number of active rays, and immediately run the second macro traverse step using this count. The rest of the pass proceeds as normal.

B. Build Speculated Raytracing Kernel Inputs

The construction of the inputs for the raytracing kernel when speculation is enabled is nearly identical to the step without speculation (Section III-F). The key difference is that ray IDs are now repeated N_{Spec} times in the active ray ID buffer I_{RAct}, meaning that the ray ID alone is no longer a unique identifier for a ray-block intersection.

We introduce an additional offset buffer, O_{Spec}, that assigns a unique index to each ray-block intersection. O_{Spec} is produced by scanning the buffer that marks active ray-block intersections, M_{RAct}. M_{Ract} is produced as before during the compaction of active ray IDs (Figure 2d.3). As with I_{RAct}, O_{Spec} is compacted down to just the entries for active ray-block intersections and sorted by block ID to match the order of I_{RAct}. The list of visible block IDs (I_{BVis}), the number of rays to process for each block (N_{BRays}), and the offsets (O_{BRays}) are produced as before.

C. Raytracing Visible Blocks with Speculation

With the entries in I_{RActive}, I_{BVis}, N_{BRays} and O_{BRays} already accounting for speculated ray-block intersections, few modifications are needed to the raytracing kernel. As before, after loading the dual grid data each visible block reads its ray IDs from the offset given in O_{BRays} and traverses the rays through its dual grid to find ray-isosurface intersections. However, as intersections may be found in multiple blocks for a given ray when speculation is enabled, the kernel is modified to output intersection results to a new RGBZ buffer instead of directly to the framebuffer. Color and depth values for ray-isosurface intersections are written at offsets given in O_{Spec} for the ray-block intersection.

D. Depth Compositing Speculated Intersections

The final step in our speculative rendering pipeline is to perform depth compositing on the set of intersections found for each ray. A kernel is run for each active ray that iterates through its N_{Spec} potential intersections to select the closest one, if any, and writes it to the framebuffer. We note that it would be possible to skip the depth compositing step if WebGPU supported 64-bit atomics, as the depth sorting could be performed using atomic min operations in the raytracing kernel instead [47]. Rays that exit the volume without finding a hit are also marked as terminated in this step.

E. Optimizing Utilization by Starting with Speculation

Our speculative traversal algorithm decouples the number of blocks checked for ray-block intersections from the image size. The algorithm can be configured to search for an arbitrary number of intersections per pass by increasing the size of our ray and block data buffers instead of only waiting for rays to terminate. By increasing these buffers' size from $w \times h$ to $w \times h \times s$, we effectively set a guaranteed starting speculation count s. Increasing the starting speculation count increases the size of the virtual GPU and could lead to higher physical GPU utilization on more powerful systems, at the trade off of slightly higher memory consumption and potentially unnecessary computation. In Section VI we explore the effect of increasing the starting speculation count.

V. Deep Learning for Intermediate Image In-Fill

Although speculation improves utilization and runtimes substantially (Figure 4), we still observe that the later passes with few highly-speculated glancing rays contribute disproportionately to the total surface rendering time. As our algorithm progressively builds up the image over multiple passes, it is simple to stop the rendering after it has reached some image completion threshold. Benchmarks on all machines and datasets show that average rendering speedups of 2.54×, 2.02×, and 1.63×, could be achieved when only completing 85%, 90%, and 95% of the image respectively. This motivates us to explore a deep learning approach that can improve the quality of these 85%, 90%, and 95% complete images, replacing the computationally expensive final passes of our renderer with a dataset-independent inference step. Skipping these final passes improves interactivity by quickly providing high-quality approximations of the completely rendered image, and when full accuracy is needed, inference can be disabled or 100% completion can be required.

A. *Network Architecture*

Although our target is reconstruction of 85-95% complete images, due to rays skipping through empty space terminating immediately and volumes having turbulent surfaces, in many cases the regions to be inferred are surrounded with only sparse input. Thus we follow the network architecture of Bauer et al. [3] for reconstruction of sparse images, and adopt FoVolNet's two-stage hybrid architecture for our model. This architecture is based on W-Net [53], and utilizes two U-Net [43] networks in sequence. The first network is used for directly filling in the incomplete image and the second for refining output to a high-quality final image. To produce temporally stable output over sequences of frames, several recurrent connections are made to accumulate state over time.

While FoVolNet's architecture is designed with a focus on inference speed, limitations of targeting lightweight devices in the browser necessitate some adaptations for our use case. First, while their work is able to take advantage of post-training quantization and half-precision floating point format to optimize performance, these are not currently supported in web-based machine learning inference engines. Additionally, while their work targets systems with dedicated graphics cards, we focus our algorithm for targeting lightweight consumer devices which may have only integrated graphics hardware. In order to maintain inference speed with these requirements, we configure our model's U-Net networks with much shallower encoder and decoder blocks. Denoting encoder/decoder blocks with e/d and following with the convolution depth of the block, FoVolNet's block configuration is e64-e64-e80-d96-d80-d64-d32, while ours is e4-e8-e16-d32-d16-d8-d4. With testing, we found this configuration greatly improved inference speed (especially on systems with integrated graphics) while only slightly decreasing inference quality.

B. *Loss*

We utilize a loss function solely on RGB for training. Although Weiss et al. [58] find losses on color input to degrade quality compared to losses on normal, hit mask, and ambient occlusion map input for superresolution tasks on isosurface images, our system performs well with only RGB loss. This is likely due to our rendering application using simple Blinn-Phong shading, while theirs uses more complex shading with ambient occlusion, leading to more ambiguity in the shading of colored pixels.

Our loss function consists of a combination of spatial and temporal losses accumulated over all the frames in an input image sequence. For spatial loss, we found best model performance when using a loss function that combines a small mean absolute error (L1) term with the multi-scale structural similarity index (MS-SSIM) [63]. This loss is exponentially down-weighted for earlier images in an input sequence to incentivize the model to use recurrent connections. For temporal loss, we use a simple L1 loss over previous frames to reduce temporal flickering [19]. We experimented with adding a small term based on optical flow [25], but this led to hole artifacts appearing in predicted isosurfaces, especially in early frames of an image sequence.

C. *Training*

We train our model using 16-frame camera orbit video sequences of the Skull, TACC, Plasma, and Miranda datasets from Table I rendered with our application. Initial position, isovalue, and rotation speed are chosen randomly for each sequence. Input is given as the frames of these sequences where rendering was stopped with an 85% image completion threshold, along with the reference frames at 100% completion. To improve the model's ability to learn what regions to infer, we embed an active ray mask into the partial images by coloring pixels whose rays are still active a solid red color, effectively using the red channel of the input images to pass an active ray mask. As we only use a single color for surfaces, we are able to embed a mask marking active rays by coloring active pixels red; however, this mask could also be passed as a separate image when using more complex shading. Early results showed that the active ray mask led to much higher inference quality compared to models trained without the mask, due to better ability to identify empty vs. incomplete regions.

For each dataset, 250 sequences were collected, for a total of 32,000 input and reference images. Sequences were split randomly into training and validation datasets with a 9:1 ratio. While these sequences were collected at a resolution of 1280×720, we randomly tile these sequences at 256×256 during training to reduce training time and improve model generalizability. We apply data augmentations of flipping horizontally, flipping vertically, and rotating 90 degrees to sequences at a probability of 50% each to effectively increase the size of our training set.

All training was done in Pytorch [39] on an NVIDIA RTX 4070 laptop GPU. Due to memory constraints, we conduct training with a mini-batch size of 1, but implement gradient accumulation to mimic an effective batch size of 5. For the final version of our model, we use an initial learning rate of 5×10^{-3} with a cosine annealing schedule to gradually reduce the learning rate to a minimum of 1×10^{-8}. To improve training stability and optimization, we use the Ranger [61] optimizer with a weight decay of 1×10^{-2} along with adaptive gradient clipping [48] with a clip percentile of 10. Training was done for 300 epochs and took 16 hours. Our model's size is 0.135MB, which is negligible compared to the datasets we target.

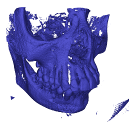
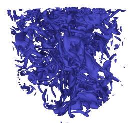
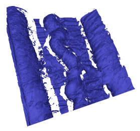
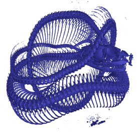
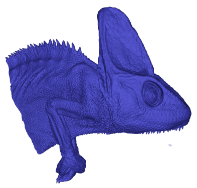
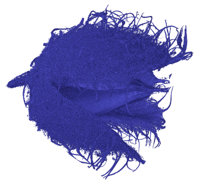
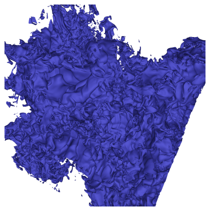
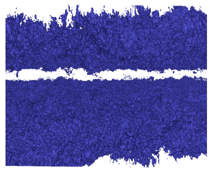
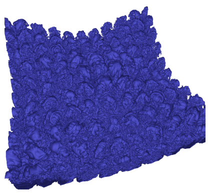
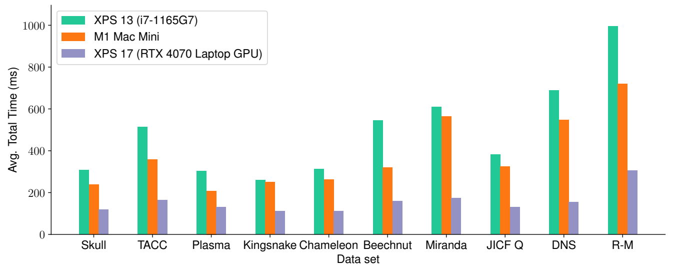
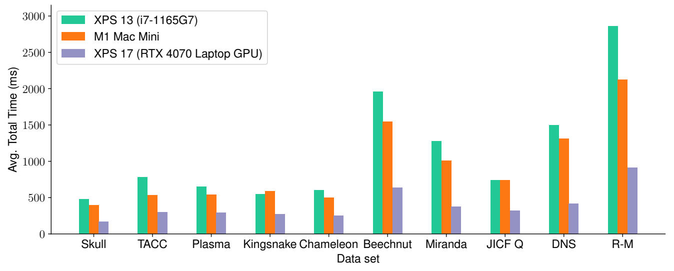
 Skull 256^3	 TACC 256^3	 Plasma 512^3	 Kingsnake $1024 \times 1024 \times 785$
 Chameleon $1024 \times 1024 \times 1080$	 Beechnut $1024 \times 1024 \times 1546$	 Miranda 1024^3	
JICF Q $1408 \times 1080 \times 1100$	 DNS $1920 \times 1440 \times 288$	 R-M $2048 \times 2048 \times 1920$	

TABLE I: The data sets used for evaluation range from small to massive and come from both measured data sets and simulations, covering a wide range of isosurface visualization use cases. The DNS combines adaptive precision and resolution techniques [21] to enable visualization of the original 1TB ($10240 \times 7680 \times 1536$) volume in the browser.



(a) Average time to 85% image completion.



(b) Average time to 100% completion.

Fig. 5: With progressive rendering, our method achieves interactive framerates to reasonable isosurface approximates (a) across the data sets tested, even on the XPS 13 and M1 Mac Mini. Moreover, rendering cost does not scale significantly with data size, allowing large and complex to be rendered interactively on lightweight systems. Our speculative approach completes rendering in few passes, allowing for reasonable surface completion times (b).

VI. Evaluation

We evaluate the rendering performance and memory consumption of our method on data sets ranging in size from 256^3 (16.7M voxels) up to $2048 \times 2048 \times 1920$ (8.05B voxels) (Table I). Each data set is compressed offline with ZFP to produce the compressed data used by the renderer. As ZFP only supports single- and double-precision floating point values, the compression step also converts any non single-precision data sets to single-precision. Each data set is benchmarked on 10 random isovalues sampled over values of interest in the data. Each isovalue is rendered over a 10 position camera orbit around the volume. We also demonstrate visualization of complex isosurfaces on the 1TB DNS data set; the DNS is first resampled from $10240 \times 7680 \times 1536$ to $1920 \times 1440 \times 288$ through a combination of adaptive precision and resolution techniques [21], then compressed with ZFP.

The test data sets cover a range of isosurface visualization scenarios, with some being especially challenging for surface extraction techniques. The Skull, Kingsnake, Chameleon, and Beechnut were produced through various scanning technologies. The Skull and Chameleon consist of relatively smooth shell-like isosurfaces, while the Kingsnake contains many fine features. The Beechnut is a challenging case with many fine features and noise, resulting in large isosurface mesh where large numbers of triangles will be occluded. The TACC, Plasma, Miranda, JICF Q, DNS, and Richtmyer-Meshkov (R-M) were produced through various simulation codes. The Miranda, DNS, and R-M pose similar challenges to surface extraction techniques as the Beechnut; they consist of highly dense isosurfaces that result in large meshes with large numbers of occluded triangles. The JICF Q is similarly challenging, as a few isosurfaces cover a substantial portion of the domain, producing a large surface that requires a large amount of data to be decompressed.

We report performance results of our algorithm on three different laptops. Two are representative of lightweight end user systems: a laptop with an i7-1165G7 CPU and integrated Intel Iris Xe Graphics (XPS 13), and a Mac Mini with an M1 chip (M1 Mac Mini). The final system is a more powerful XPS 17 laptop with an RTX 4070 Laptop GPU and an i9-13900H CPU. The XPS 17 also includes integrated Intel Iris Xe Graphics.

We conduct a detailed evaluation of our method's performance and scalability and evaluate it against the state of the art in GPU-based large-scale isosurface extraction [54]. In Section VI-A, we discuss overall rendering performance of our method. In Section VI-B, we evaluate the scalability of our method with respect to data set size and image resolution compared to the state of the art. In Section VI-C, we evaluate our deep learning approach for reconstructing incomplete isosurface images. Finally, Section VI-D evaluates the memory consumption of our method against the state of the art.

A. *Rendering Performance*

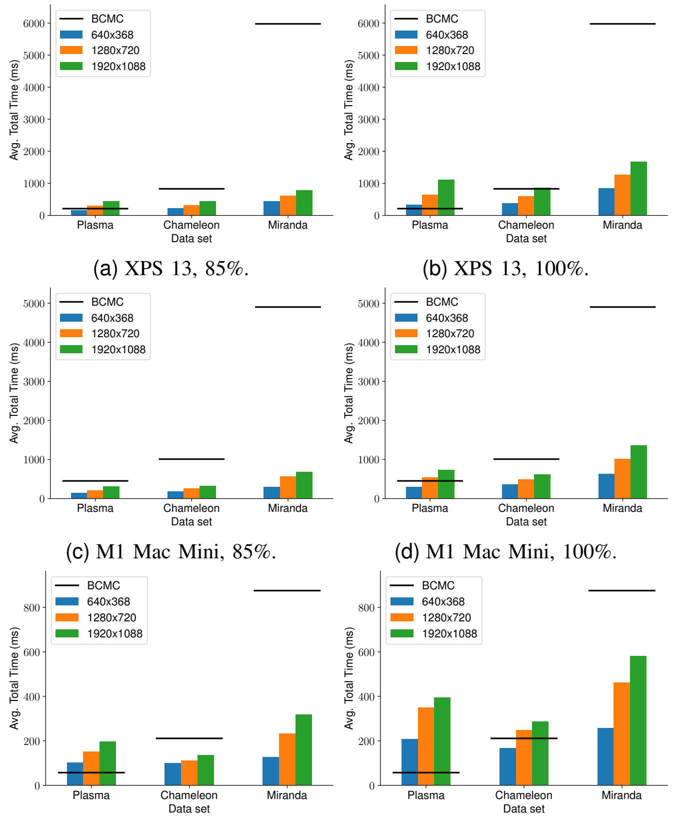
The average time to 85% and 100% isosurface image completion at 720p across the data sets and hardware platforms tested are shown in Figure 5. Because of progressive rendering, our method can achieve interactivity even when visualizing the massive and complex isosurfaces of the Beechnut, Miranda, JICF Q and DNS data sets on the XPS 13 and M1 Mac Mini. When comparing performance across data sets, we observe that our algorithm's performance is nearly independent of data set size. Instead, ours scales with the visible surface area and complexity of the isosurface. We achieve similar performance on data sets with similar isosurface structure, such as the Skull, Plasma, Kingsnake, Chameleon and JICF Q, even though these data sets range in size from 256^3 to $1408 \times 1080 \times 1100$. These data sets have relatively smooth isosurfaces, where rays can quickly skip empty space to reach the isosurface and find an intersection. In contrast, data sets with noisier or more complex isosurfaces such as the Beechnut, Miranda, DNS, and R-M, see higher rendering times, as more data must be processed for each ray to find an intersection. We also find that our progressive approach is valuable to quickly provide a nearly complete image of the data set, with times for 85% image completion much faster than 100% image completion.

We additionally experiment with increasing our starting speculation counts, and corresponding ray data buffers, by factors of 2, 4, and 8. We find that speed-ups are achieved on especially dense and noisy data sets, such as the Beechnut and Miranda, or smaller datasets, such as the Skull and TACC. On dense and noisy datasets, many rays remain active for many passes, and larger ray data buffers ensure these rays can still have high speculation counts and terminate faster. Speedups for the XPS 13, M1 Mini, and XPS 17 were 1.10×, 1.20×, and 1.25× at starting speculation counts 2, 4, and 2 on the Beechnut. On smaller datasets, larger ray data buffers allow increasing work per pass to more fully utilize the GPU. Speedups for the XPS 13, M1 Mini, and XPS 17 were 1.33×, 1.08×, and 2.05× at starting speculation counts 4, 4, and 8 on the Skull dataset. In Figure 5, times reported are for the best performing starting speculation count for each dataset and machine.

B. Scalability with Image and Data Size

The performance of our method is primarily driven by the visible surface area and complexity of the isosurface being rendered, and is less tied to the data set size. Another main driver of rendering cost in our method is the number of pixels, allowing rendering performance to be improved by reducing the image size. This is in line with prior implicit isosurface and volume raycasting techniques, which have image-order scaling. Explicit isosurface extraction techniques, such as BCMC [54], typically extract the complete triangle mesh for the isosurface, including triangles that will be occluded in the final rendering. These techniques scale with the size of the output isosurface, and are more affected by data set size.

Figure 6 quantifies the benefits of these properties of our algorithm against BCMC [54]. We conduct benchmarks rendering at 1920×1088 (1080p), 1280×720 (720p), and 640×368 (360p) on the Plasma, Chameleon, and Miranda data sets, and compare against the isosurface extraction times achieved by BCMC. To be compatible with our machine learning architecture, image width and height must be divisible by 2^4, so typical widths of 360 and 1080 are slightly modified to 368 and 1088 for our benchmarks. Benchmarks for BCMC were run over 100 random isovalues, while benchmarks for our method were measured over a 10 position camera orbit for 10 random isovalues, starting positions, and rotation speeds. Results for our method are shown for each resolution, while BCMC is shown as a solid line as its compute costs are resolution independent. Our method achieves an average 1.32× reduction in 85% and fully complete times when scaling down



(a) XPS 13, 85%. (b) XPS 13, 100%.

(c) M1 Mac Mini, 85%. (d) M1 Mac Mini, 100%.

(e) RTX 4070 Laptop, 85%. (f) RTX 4070 Laptop, 100%.

Fig. 6: The performance scaling of our approach and BCMC with image resolution and data set size. BCMC's compute cost is tied to the size of the data set and the size of the output triangle mesh, making it difficult to scale down to ensure interactivity. In contrast, our approach can scaled down easily by reducing the image resolution, and is less effected by data size overall, enabling interactive rendering of massive data sets on lightweight devices.

from 1080p to 720p, and an additional 1.56× reduction when scaling down from 720p to 360p.

We compare our algorithm's interactivity and total isosurface computation times against BCMC by comparing 85% image completion (Figures 6a, 6c and 6e) and total times (Figures 6b, 6d and 6f) against the surface extraction times achieved by BCMC. Through our progressive rendering approach, we are able to show an 85% complete image with better interactivity than BCMC in all but two cases, the Plasma on the XPS 13 and RTX 4070, while requiring substantially less memory. In addition, the time to fully complete the isosurface with our algorithm was on par or faster for both the larger datasets on the lightweight machines. The interactivity improvement achieved by our method is especially pronounced on data sets with large and complex isosurfaces such as the Miranda, where BCMC struggles with the large number of active blocks and the size of the surface mesh. At 1080p on the Miranda we achieve 7.6×, 7.1× and 2.8× faster 85% completion times and 3.6×, 3.6× and 1.5× faster total times on XPS 13, M1 Mac Mini and RTX 4070 respectively, compared to BCMC's surface extraction times. These speedups become even greater if comparing BCMC against the 720p or 360p rendering results.

As before, all benchmarks in Figure 6 were performed with starting speculation counts of 1, 2, 4, and 8, with the best performing reported. Results only increased significantly

Device	Backend	360p	720p	1080p
RTX 4070	TensorRT	4ms	16ms	37ms
	Pytorch	5ms	17ms	41ms
	ONNX Web	25ms	68ms	189ms
XPS 17 Integrated Graphics	Pytorch	87ms	246ms	477ms
	ONNX Web	93ms	266ms	718ms
XPS 13 Integrated Graphics	ONNX Web	101ms	374ms	933ms

TABLE II: Average inference times for our model. ONNX web performs much worse than native libraries on dedicated GPU, likely due to memory management issues or falling back to CPU execution for operations missing from the still in-development WebGPU backend. For integrated graphics systems, this is much less impactful.

Data Set	85% Threshold		90% Threshold		95% Threshold	
	SSIM	PSNR	SSIM	PSNR	SSIM	PSNR
Chameleon	0.92	22.64	0.96	25.87	0.99	32.48
Beechnut	0.89	21.00	0.93	21.62	0.98	27.75
JICF Q	0.93	22.42	0.97	26.71	0.99	32.71
DNS	0.92	26.28	0.96	29.39	0.99	36.63
R-M	0.93	27.47	0.97	30.28	0.99	38.06

TABLE III: Average SSIM and PSNR values for the 5 largest datasets after 10 orbit sequences of 16 frames each. Input was given as rendering stopped at 85%, 90% and 95% completion thresholds for the same isovalue and viewpoint.

with higher starting counts at 360p, likely due to better GPU utilization by performing more work per pass when the number of rays is small. Average speedups across datasets on the XPS 13, M1 Mini, and XPS 17 were 1.27×, 1.21×, and 1.56× at best performing starting speculation counts 2, 2 and 8.

C. Inference

Up to this point, we have discussed performance when rendering 85% complete images, but have left out our deep learning inference on intermediate results. As our algorithm's main target is the browser, we experimented with web-based machine learning inference engines (namely TensorFlow.js [50] and ONNX Runtime Web v1.18.0) to run our model directly in the rendering application. These libraries both support GPU utilization for inference with WebGL, and have experimental WebGPU backends. Initial tests with our model showed that both web libraries could not support WebGL-powered inference for our case, either because of WebGL texture size limits or missing operators used by our architecture, so we used the experimental WebGPU backends for both. We found significantly faster performance for our model using ONNX Runtime Web (ORT Web) compared to TensorFlow.js, so we choose this library's web inference engine for our application.

To test the performance of our model, we measure ORT Web inference timings on the dedicated RTX 4070 GPU and the integrated graphics of the XPS 17, along with the integrated graphics of the XPS 13. For comparison, we also measure inference timings using two native GPU-accelerated libraries: Pytorch, using the CUDA backend for dedicated GPU and DirectML backend for integrated graphics, and TensorRT, which only supports dedicated GPU. For ORT Web and TensorRT, our models were converted to ONNX format before being used for inference.

Inference benchmarks were performed by first running 10 warm-up inferences, then taking the average time of 500 inferences on example images from our training dataset. Results are reported in Table II. Inference times using native libraries (Pytorch and TensorRT) are shown to be significantly (around 4-5×) faster than ORT Web on the dedicated GPU. Because this trend does not occur when using integrated graphics, this is likely because of silently falling back to CPU-based operators when the model tries to run operations not supported by the WebGPU backend. While these results are disappointing, ORT Web's WebGPU backend is experimental and in active development, and we expect performance to improve in the future.

By adding the inference times to our dataset benchmarks at 85% image completion and comparing with the 100% completion times, we can see the effective speedup of using ML-reconstructed intermediates rather than full rendering. Because the inference times are dataset-independent, the speedup is naturally best for the datasets that are slowest to render. For the four datasets with the highest time to completion (Beechnut, Miranda, DNS, and R-M), we achieve average speedups of 2.16×, 2.27×, and 1.75× at 360p, 720p, and 1080p when using ORT Web on the RTX 4070 GPU. For the same system, TensorRT gives speedups of 2.71×, 2.59×, and 2.63×, showing how the performance should increase as the ORT Web WebGPU backend improves. Even using the integrated graphics of the XPS 13 with ORT Web, we see average speedups of 1.73× and 2.15× at 360p and 720p, although the performance gain does not scale to 1080p due to the inference workload becoming too heavy for the lightweight system.

While this shows the improved interactivity of using ML-reconstructed intermediates, it is important to show the perceptual benefit of the model inference compared to the intermediate images. To do this, we conduct a thorough study on the quality of our model's inference output using datasets that were not seen during model training. For each of the 5 largest unseen datasets (Chameleon, Beechnut, JICF Q, DNS, and R-M), we run 10 camera orbit benchmarks, outputting 16 frames each. For each frame, we output images of the rendering stopped at 85%, 90%, 95%, and 100% completion thresholds. Images at 100% completion are used as ground truths, while images at other thresholds are grouped to be used as test sets for our model. Inference is done for each camera orbit sequence at each threshold, and recurrent state is cleared between sequences.

To evaluate the model output on these test sets, we compare against the ground truth images using conventional metrics of structural similarity (SSIM [57]) and peak signal to noise ratio (PSNR). SSIM measures image distortion in a way that correlates with the quality perception of the human visual system, while PSNR provides a measure directly on the magnitude of numerical differences between images [22]. Better scores for output imply the model has given a better perceptual approximate of the ground truth (higher SSIM) and has more faithfully reconstructed the precise values of the ground truth (higher PSNR).

Average PSNR and SSIM values of model output are reported in Table III for each dataset and completion threshold. As the image completion threshold is increased, the SSIM and PSNR values for image reconstruction of all datasets increases dramatically, with near ground truth results at 95%. By providing users the ability to set the image completion threshold, we provide another axis for users to trade some output image quality for performance to improve interactivity.

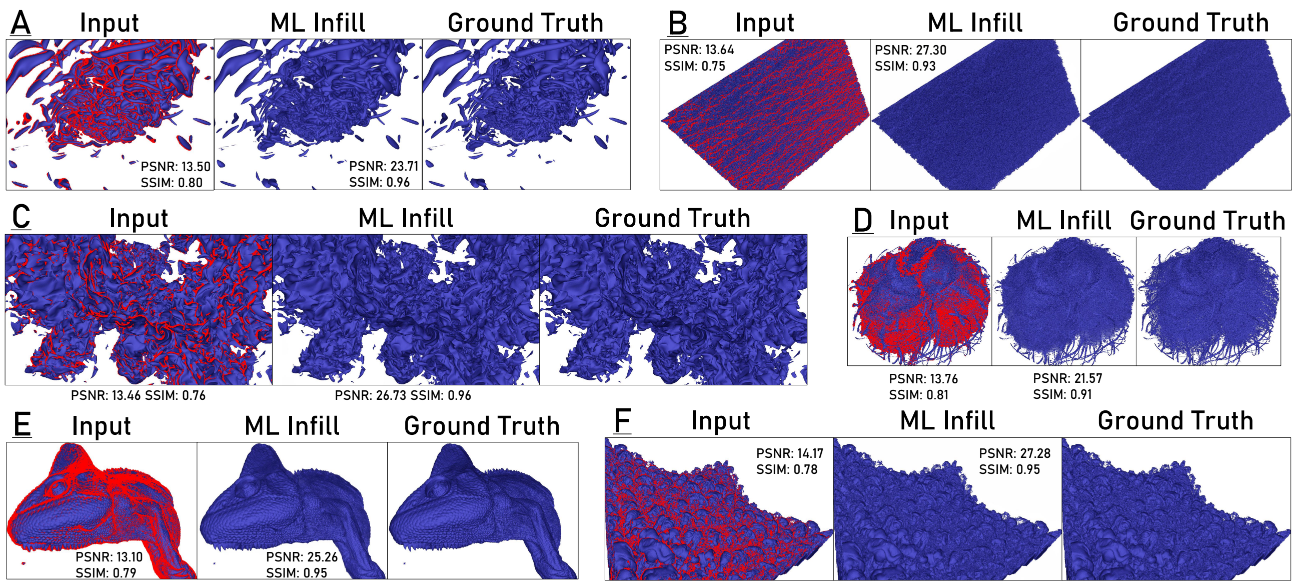


Fig. 7: Example inference from our model on the six largest datasets with image quality metrics compared to ground truth. Input is given as the intermediate result when our multipass algorithm reaches 85% image completion. Rays that are still active are marked by red pixels to form the active ray mask. Datasets are A) JICF Q, B) DNS, C) Miranda, D) Beechnut, E) Chameleon, and F) R-M. Our model performs best when the structure of the dataset leads to uniform samples for our model to infer from (F), and struggles when sampling is uneven due to sparse noisy regions (D).

Data set	BCMC Avg. Mem	Our Avg. Mem	Reduction
Skull	329MB	109MB	3.02×
TACC	187MB	108MB	1.73×
Plasma	563MB	191MB	2.95×
Kingsnake	1.34GB	607MB	2.22×
Chameleon	2.09GB	691MB	3.02×
Beechnut	—	1.06GB	—
Miranda	4.20GB	737MB	5.70×
JICF Q	—	1.00GB	—
DNS	—	875MB	—
R-M	—	4.19GB	—

TABLE IV: The average total compute memory overhead required by our algorithm vs. BCMC. We achieve an average 3.1× reduction in total memory overhead. Entries marked by — crashed due to exceeding the 4GB buffer binding limit in WebGPU.

Data set	BCMC Avg. Cache Mem	Our Avg. Cache Mem	Reduction
Skull	66.0MB	16.9MB	3.9×
TACC	55.0MB	16.0MB	3.4×
Plasma	107MB	44.7MB	2.4×
Kingsnake	545MB	145MB	3.8×
Chameleon	375MB	102MB	3.7×
Beechnut	1.99GB	243MB	8.2×
Miranda	1.40GB	167MB	8.4×
JICF Q	—	170MB	—
DNS	2.02GB	406MB	5×
R-M	—	522MB	—

TABLE V: The average cache size required by our algorithm vs BCMC. Our progressive wavefront traversal achieves a significant reduction in the volume working set size, providing a 4.8× reduction in cache size on average. Entries marked by — crashed due to the cache exceeding the 4GB buffer binding limit in WebGPU.

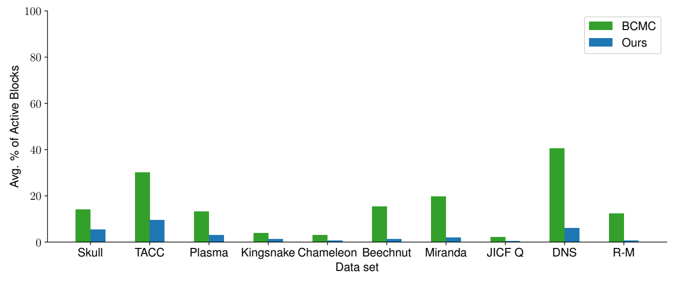
Examples of 85% completion input, output, and ground truths are given for the 6 largest datasets in Figure 7, along with PSNR and SSIM values for the input and model output. Results show that our method can increase PSNR and SSIM dramatically from intermediate results of our progressive rendering algorithm. However, some datasets and viewpoints improve in quality much more than others. We find the reason for this in most cases to be the sampling pattern of the input images. Input that includes sparsely sampled regions performs worse due to lack of rich pixel neighborhoods for the model to draw information from. This can occur due to the structure of datasets or the camera viewpoint causing uneven ray termination for the input image; regions closer to the camera or intersecting smooth areas of a dataset will terminate faster than those far from the camera or glancing through many blocks of a turbulent area. This can be seen in the quality of inference on the R-M dataset (Figure 7, F) vs. the Beechnut dataset (Figure 7, D). Compared to Beechnut, R-M has much more even sampling of the entire volume in the input, leading to much higher PSNR and SSIM values for the model output.

D. Memory Consumption

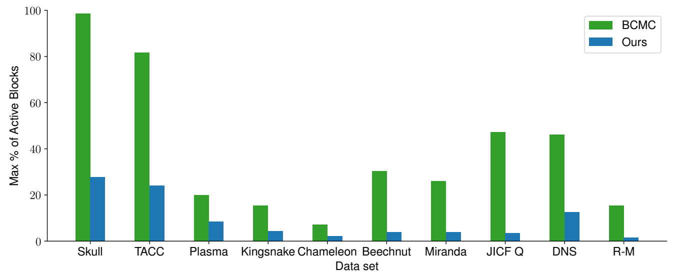
Finally, we compare the memory overhead of our technique against BCMC [54]. BCMC provides a direct comparison point for explicit isosurface extraction algorithms, as it also works directly on compressed data sets and performs on the fly decompression to reduce memory overhead. We report average memory statistics over 100 random isovalue and 10 camera position orbit benchmarks, rendering at 1280×720 (Tables IV and V).

We achieve an average memory overhead reduction of 3.1× compared to BCMC on the data sets BCMC is able to compute on without running out of memory (Table IV). These memory reductions are achieved through our algorithm's use of implicit ray-isosurface intersection, which eliminates the need for a large vertex buffer, and our progressive wavefront traversal, which significantly reduces the amount of data that must decompressed to render the isosurface.

Furthermore, BCMC failed to compute the isosurface on the Beechnut, JICF Q, DNS, and R-M, due to exceeding WebGPU's buffer size limit of 4GB. These large data sets have noisy or turbulent isosurfaces, resulting in some isosurfaces containing over 500M triangles. Even with BCMC's quantized vertex format, these large isosurfaces exceed 4GB, resulting in a crash. These results were run on an RTX 3080, which has 12GB of GPU memory; however, on the XPS 13 or M1 Mac Mini systems these data sets would fail due to running out of GPU memory, even if the size limit was lifted or otherwise



(a) The average percentage of active blocks.



(b) The max percentage of active blocks.

Fig. 8: The (a) average and (b) max percentage of active blocks required by BCMC and our algorithm. Our approach updates the cache each pass, storing just the blocks needed by active rays. In contrast, BCMC decompresses all blocks that may contain the isosurface.

worked around. Our algorithm is able to achieve interactive rendering of these massive isosurfaces, even on the XPS 13 and M1 Mac Mini.

By progressively stepping rays through the volume and decompressing and caching just the blocks required for each pass, we achieve significant reductions in the amount of data that must be decompressed. Table V compares the average cache memory required by BCMC and our algorithm. The Miranda and DNS results for BCMC were measured by disabling the vertex extraction step. However, on the JICF Q and R-M, BCMC's active block cache memory alone exceeded 4GB, resulting in a crash. We achieve an average cache size reduction of 4.8× compared to BCMC on the data sets it is able to compute, with far greater reductions achieved on the Beechnut (8.2×) and Miranda (8.4×). The Beechnut is a noisy microCT scan and the Miranda is from a turbulent fluid mixing simulation, resulting in large numbers of active but occluded blocks being decompressed and processed by BCMC.

Our view-dependent algorithm achieves substantial reductions in the number of blocks that must be decompressed (Figure 8). These reductions come from a number of factors: our algorithm can replace unneeded blocks with active ones each pass to minimize its working set; blocks that are occluded or otherwise not visible are not decompressed; and the number of visible blocks is driven by the image size and view position. Compared to BCMC, we achieve a 6.7× reduction in the average number of active blocks and a 5.7× reduction in the maximum number of active blocks. The JICF Q and R-M results for BCMC were measured by only recording the number of active blocks for each isovalue and skipping all other computation to avoid crashing.

VII. Conclusion and Limitations

We have proposed a new view-dependent isosurface rendering algorithm designed specifically for interactive visualization of massive isosurfaces on lightweight consumer platforms. This is achieved through a progressive wavefront ray traversal algorithm with per-pass block cache updates, where blocks of the data are decompressed and cached on demand for each pass. We accelerate isosurface rendering completion and increase GPU utilization by introducing ray-block speculation into the algorithm. Speculation enables us to fill open compute slots with speculated ray-block intersections to better leverage the GPU's parallel compute power and complete rendering in fewer passes and less time. By leveraging a neural network for image reconstruction, we are able to improve interactivity by providing high quality approximations of complete rendering from intermediate passes, even for datasets unseen in model training.

Our progressive, view-dependent isosurface rendering algorithm is well suited to large scale isosurface visualization on end-user devices. The memory costs of our algorithm are not strongly affected by data set size and are much smaller than the state of the art, enabling larger datasets to be rendered. The progressive rendering provided by our algorithm makes it well suited to provide low-latency interactive visualization. The compute costs of our algorithm can be easily reduced by simply reducing image resolution, effectively scaling rendering down to maintain interactivity on lightweight systems and mobile devices. Interactivity can be further improved by using our deep learning approach to reconstruct approximates of a complete rendering from early intermediate passes, enabling the trade-off of rendering accuracy for performance by varying the necessary completion threshold. Our algorithm runs entirely in the browser on the GPU through WebGPU to expand access to large scale data visualization, and is available on GitHub[1], along with a live demo[2].

Our approach is not without its limitations. Although our method scales up well to large data sets, it does not scale down as well to small data sets. For example, BCMC achieves faster surface extraction times on the Plasma and, in some cases, on the Chameleon. Our approach still uses less memory on these data sets; however, BCMC's overhead on smaller data sets is likely acceptable for the performance improvement. Further optimization efforts would be worthwhile to improve performance on smaller data sets, improve scalability with image size, and reduce overhead to improve per-pass and total rendering times overall. We also find call overhead in JavaScript and WebGPU and note that better performance could be achieved with a CUDA implementation where optimized libraries such as Thrust and CUB are available. Bringing these libraries to WebGPU would be a valuable effort.

There are also a number of interesting avenues left open for future work. Although our speculation approach increases utilization and achieves large speed-ups in total surface rendering time, our use of a global speculation count for all rays is restrictive. It may be possible to achieve higher utilization by tracking a per-ray speculation count; however, the added complexity may introduce additional overhead. It would also be worthwhile to explore other acceleration structures that

[1]https://github.com/ldyken53/TVCG-progiso
[2]https://ldyken53.github.io/TVCG-progiso/

can be built over the macrocell grid instead of our two-level grid to improve space skipping and provide level of detail or multiresolution hierarchies to address current limitations of our method with respect to undersampling the data. For example, an implicit *k*-d tree [55] built over the blocks could further accelerate empty space skipping, or multiresolution and compression techniques from work on compressed volume rendering could be integrated [2], [13]–[15], [35], [44], [56]. Leveraging multiresolution hierarchies within our method would address limitations with respect to undersampling of the high-resolution data, and enable rendering larger data sets. To improve image quality, it would be worth exploring support for secondary ray tracing effects in our pipeline to add shadows, ambient occlusion, and global illumination with denoising. To improve the reconstruction quality of our model, it could be worthwhile to explore training with a custom loss function built specifically for our unique pass in-fill task. Previous work [37] has shown this strategy to be beneficial for other image reconstruction applications.

Finally, as our algorithm's rendering and memory costs are primarily driven by the number of rays traced and the number of passes, it would be worthwhile to combine our deep learning method (which reduces the number of passes required) with machine learning approaches for image up-scaling [58] and foveated rendering [3] in order to also decrease the number of rays. Such a combination would further lessen the number of data samples required for our algorithm; reducing total surface rendering times and memory footprint.

Acknowledgements

This work was funded in part by NSF RII Track-4 award 2132013, NSF PPoSS planning award 2217036, NSF PPoSS large award 2316157 and, NSF collaborative research award 2221811.

References

[1] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *EG 1987-Technical Papers*, 1987.

[2] M. Balsa Rodríguez, E. Gobbetti, J. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. Suter. State-of-the-Art in Compressed GPU-Based Direct Volume Rendering: State-of-the-Art in Compressed GPU-Based DVR. *Computer Graphics Forum*, 2014.

[3] D. Bauer, Q. Wu, and K.-L. Ma. FoVolNet: Fast Volume Rendering using Foveated Deep Neural Networks. *IEEE Transactions on Visualization and Computer Graphics*, 2023.

[4] J. Beyer, M. Hadwiger, and H. Pfister. State-of-the-Art in GPU-Based Large-Scale Volume Visualization. *Computer Graphics Forum*, 2015.

[5] P. Cignoni, C. Montani, E. Puppo, and R. Scopigno. Optimal isosurface extraction from irregular volume data. In *Proceedings of 1996 Symposium on Volume Visualization*. IEEE Press, 1996.

[6] M. Ciżnicki, M. Kierzynka, K. Kurowski, B. Ludwiczak, K. Napierała, and J. Palczyński. Efficient Isosurface Extraction Using Marching Tetrahedra and Histogram Pyramids on Multiple GPUs. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, eds., *Parallel Processing and Applied Mathematics*, 2012.

[7] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. ACM, 2009.

[8] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. High-speed Marching Cubes using HistoPyramids. *Computer Graphics Forum*, 2008.

[9] L. Dyken, P. Poudel, W. Usher, S. Petruzza, J. Y. Chen, and S. Kumar. GraphWaGu: GPU Powered Large Scale Graph Layout Computation and Rendering for the Web. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2022.

[10] T. Dykes, A. Hassan, C. Gheller, D. Croton, and M. Krokos. Interactive 3D Visualization for Theoretical Virtual Observatories. *Monthly Notices of the Royal Astronomical Society*, 2018.

[11] K. Engel. CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs. In *2011 IEEE Symposium on Large Data Analysis and Visualization*. IEEE, 2011.

[12] T. Fogal, A. Schiewe, and J. Krüger. An analysis of scalable GPU-based ray-guided volume rendering. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization*, 2013.

[13] N. Fout, H. Akiba, K.-L. Ma, A. E. Lefohn, and J. Kniss. High-Quality Rendering of Compressed Volume Data Formats. *EUROVIS 2005: Eurographics / IEEE VGTC Symposium on Visualization*, 2005.

[14] N. Fout and K.-L. Ma. Transform Coding for Hardware-accelerated Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[15] E. Gobbetti, J. A. Iglesias Guitián, and F. Marton. COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks. *Computer Graphics Forum*, 2012.

[16] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister. SparseLeap: Efficient Empty Space Skipping for Large-Scale Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2018.

[17] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach. *IEEE Transactions on Visualization and Computer Graphics*, 2012.

[18] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum*, 2005.

[19] J. Hasselgren, J. Munkberg, M. Salvi, A. Patney, and A. Lefohn. Neural temporal adaptive sampling and denoising. *Computer Graphics Forum*, 2020.

[20] L. Herzberger, M. Hadwiger, R. Krüger, P. Sorger, H. Pfister, E. Gröller, and J. Beyer. Residency octree: A hybrid approach for scalable web-based multi-volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2024.

[21] D. Hoang, B. Summa, H. Bhatia, P. Lindstrom, P. Klacansky, W. Usher, P.-T. Bremer, and V. Pascucci. Efficient and Flexible Hierarchical Data Layouts for a Unified Encoding of Scalar Field Precision and Resolution. *IEEE Transactions on Visualization and Computer Graphics*, 2021.

[22] A. Horé and D. Ziou. Image quality metrics: PSNR vs. SSIM. In *2010 20th International Conference on Pattern Recognition*, 2010.

[23] H. Jacinto, R. Kéchichian, M. Desvignes, R. Prost, and S. Valette. A web interface for 3D visualization and interactive segmentation of medical images. In *Proceedings of the 17th International Conference on 3D Web Technology*, 2012.

[24] S. Jourdain, U. Ayachit, and B. Geveci. ParaViewWeb: A Web Framework for 3D Visualization and Data Processing. *International Journal of Computer Information Systems and Industrial Management Applications*, 2011.

[25] A. S. Kaplanyan, A. Sochenov, T. Leimkühler, M. Okunev, T. Goodall, and G. Rufo. Deepfovea: neural reconstruction for foveated rendering and video compression using learned statistics of natural videos. *ACM Trans. Graph.*, 2019.

[26] A. Kreskowski, G. Rendle, and B. Froehlich. Efficient Direct Isosurface Rasterization of Scalar Volumes. *Computer Graphics Forum*, 2022.

[27] J. K. Li and K.-L. Ma. P4: Portable Parallel Processing Pipelines for Interactive Information Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2018.

[28] P. Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, 2014.

[29] B. Liu, G. J. Clapworthy, F. Dong, and E. Wu. Parallel Marching Blocks: A Practical Isosurfacing Algorithm for Large Data on Many-Core Architectures. *Computer Graphics Forum*, 2016.

[30] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *Proceedings Visualization '98*. IEEE Computer Society Press, 1998.

[31] Y. Livnat, H.-W. Shen, and C. R. Johnson. A near optimal isosurface extraction algorithm using the span space. *IEEE Transactions on Visualization and Computer Graphics*, 1996.

[32] W. E. Lorenson and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Computer Graphics*, 1987.

[33] G. Marmitt, A. Kleer, I. Wald, H. Friedrich, and P. Slusallek. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *9th International Fall Workshop on Vision, Modeling, and Visualization*, 2004.

[34] S. Martin, H.-W. Shen, and P. McCormick. Load-Balanced Isosurfacing on Multi-GPU Clusters. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2010.
[35] J. Mensmann, T. Ropinski, and K. Hinrichs. A GPU-Supported Lossless Compression Scheme for Rendering Time-Varying Volume Data. *IEEE/EG Symposium on Volume Graphics*, 2010.
[36] M. M. Mobeen and L. Feng. High-Performance Volume Rendering on the Ubiquitous WebGL Platform. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012.
[37] A. Mustafa, A. Mikhailiuk, D.-A. Iliescu, V. Babbar, and R. K. Mantiuk. Training a task-specific image reconstruction loss. *2022 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2021.
[38] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive Ray Tracing for Isosurface Rendering. In *Proceedings Visualization '98*, 1998.
[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.
[40] F. Perez and B. E. Granger. IPython: A System for Interactive Scientific Computing. *Computing in Science & Engineering*, 2007.
[41] M. Raji, A. Hota, T. Hobson, and J. Huang. Scientific Visualization as a Microservice. *IEEE Transactions on Visualization and Computer Graphics*, 2018.
[42] M. Raji, A. Hota, and J. Huang. Scalable Web-Embedded Volume Rendering. In *IEEE 7th Symposium on Large Data Analysis and Visualization*. IEEE, 2017.
[43] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Cham, 2015.
[44] J. Schneider and R. Westermann. Compression domain volume rendering. In *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*. IEEE, Seattle, WA, USA, 2003.
[45] W. Schroeder, R. Maynard, and B. Geveci. Flying edges: A high-performance scalable isocontouring algorithm. In *IEEE 5th Symposium on Large Data Analysis and Visualization*. IEEE, 2015.
[46] M. Schütz. *Potree: Rendering Large Point Clouds in Web Browsers*. PhD thesis, Vienna Univ. of Technol., Vienna, Austria, 2016.
[47] M. Schütz, B. Kerbl, and M. Wimmer. Software Rasterization of 2 Billion Points in Real Time. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 2022.
[48] P. Seetharaman, G. Wichern, B. Pardo, and J. Le Roux. AutoClip: Adaptive gradient clipping for source separation networks. In *2020 IEEE 30th International Workshop on Machine Learning for Signal Processing (MLSP)*. IEEE, 2020.
[49] T. Sherif, N. Kassis, M.-Ã. Rousseau, R. Adalat, and A. C. Evans. BrainBrowser: Distributed, web-based neurological data visualization. *Frontiers in Neuroinformatics*, 2015.
[50] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel, S. M. Bileschi, M. Terry, C. Nicholson, S. N. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. S. Corrado, F. B. Viégas, and M. Wattenberg. Tensorflow.js: Machine learning for the web and beyond. *ArXiv*, 2019.
[51] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. In *Volume Graphics*. IEEE, 2005.
[52] S. K. Suter, J. A. Iglesias Guitian, F. Marton, M. Agus, A. Elsener, C. P. E. Zollikofer, M. Gopi, E. Gobbetti, and R. Pajarola. Interactive Multiscale Tensor Reconstruction for Multiresolution Volume Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2011.
[53] M. M. Thomas, K. Vaidyanathan, G. Liktor, and A. G. Forbes. A reduced-precision network for image reconstruction. *ACM Trans. Graph.*, 2020.
[54] W. Usher and V. Pascucci. Interactive Visualization of Terascale Data in the Browser: Fact or Fiction? In *IEEE 10th Symposium on Large Data Analysis and Visualization*, 2020.
[55] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel. Faster isosurface ray tracing using implicit KD-trees. *IEEE Transactions on Visualization and Computer Graphics*, 2005.
[56] C. Wang, Hongfeng Yu, and Kwan-Liu Ma. Application-Driven Compression for Visualizing Large-Scale Time-Varying Data. *IEEE Computer Graphics and Applications*, 2010.
[57] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 2004.
[58] S. Weiss, M. Chu, N. Thuerey, and R. Westermann. Volumetric Isosurface Rendering with Deep Learning-Based Super-Resolution. *IEEE Transactions on Visualization and Computer Graphics*, 2021.
[59] S. Weiss, P. Hermüller, and R. Westermann. Fast neural representations for direct volume rendering. *Computer Graphics Forum*, 2022.
[60] S. Weiss, M. IşIk, J. Thies, and R. Westermann. Learning adaptive sampling and reconstruction for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2022.
[61] L. Wright. Ranger - a synergistic optimizer. https://github.com/lessw2020/Ranger-Deep-Learning-Optimizer, 2019.
[62] Q. Wu, D. Bauer, M. J. Doyle, and K.-L. Ma. Interactive volume visualization via multi-resolution hash encoding based neural representation. *IEEE Transactions on Visualization and Computer Graphics*, 2023.
[63] H. Zhao, O. Gallo, I. Frosio, and J. Kautz. Loss functions for image restoration with neural networks. *IEEE Transactions on Computational Imaging*, PP, 2016.

Landon Dyken is a student and graduate research assistant at the University of Illinois Chicago. He is currently pursuing a PhD degree in Computer Science under Sidharth Kumar as a member of the Electronic Visualization Lab (EVL) and High Performance Computing (HPC) groups. His focus is on the subjects of data visualization and parallel computing, specifically building web-based graphics tools and GPU-accelerated systems. Before beginning his PhD, he received a dual B.S. in Mathematics and Computer Science from the University of Alabama at Birmingham in 2021, and an M.S. in Computer Science from the University of Alabama at Birmingham in 2023.

Will Usher is a Scientific Visualization Engineer at Luminary Cloud, where he works on a mix of challenging problems in computer graphics, spatial data processing, parallel computing, and rendering large data sets in the browser. He completed his Ph.D. in Computer Science at the Scientific Computing and Imaging Institute at the University of Utah, advised by Valerio Pascucci. His research interests cover a range of areas in scientific visualization and computer graphics including: distributed rendering, virtual reality, in situ visualization and ray tracing.

Sidharth Kumar is an assistant professor in the Department of computer science at the University of Illinois at Chicago. He received his Ph.D. in 2016 from the University of Utah, where he also conducted his postdoctoral research. His research interests include high-performance computing, large-scale data management, and big data analytics and visualization. His research in parallel I/O framework has been deployed at the highest scale (768K cores) and also put into production runs (at 260K cores) at some of the fastest supercomputers in the world (Theta, Mira, Polaris, Shaheen, Edison, and Hopper). He has received best paper awards at premium HPC and vis conferences such as ISC, HiPC and LDAV. His current research is supported by four NSF grants.