Received: 9 April 2019 | Revised: 21 October 2019 | Accepted: 11 November 2019

DOI: 10.4218/etrij.2019-0185

ORIGINAL ARTICLE

ETRI Journal WILEY

Accurate and efficient GPU ray-casting algorithm for volume rendering of unstructured grid data

Gibeom Gu[1] | **Duksu Kim**[2]

[1]Center for Development of Supercomputing System, Korea Institute of Science and Technology Information, Daejeon, Rep. of Korea

[2]School of Computer Engineering, Korea University of Technology and Education, Cheoan, Rep. of Korea

Correspondence
Duksu Kim, HPC Lab., School of Computer Engineering, Korea University of Technology and Education, Cheoan, Rep. of Korea.
Email: bluekdct@gmail.com

Funding information
This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT & Future Planning (2018R1C1B5045551), the National Research Council of Science & Technology (NST) grant by the Korea government (MSIP) (CMP-16-03-KISTI), the Korea Institute of Science and Technology Information (KISTI), and the research fund for a newly appointed professor of Korea University of Technology and Education (KOREATECH) in 2018.

We present a novel GPU-based ray-casting algorithm for volume rendering of unstructured grid data. Our volume rendering system uses a ray-casting method that guarantees accurate rendering results. We also employ the per-pixel intersection list concept in the Bunyk algorithm to guarantee an accurate result for non-convex meshes. For efficient memory access for the lists on the GPU, we represent the intersection lists for all faces as an array with our novel construction algorithm. With the intersection lists, we perform ray-casting on a GPU, and a GPU thread handles each ray. To increase ray-coherency in a thread block and improve memory access efficiency, we extend a prior image-tile-based work distribution method to fit modern GPU architectures. We also show that a prior approach using a per-thread local buffer to reduce redundant computation is not appropriate for modern GPU architectures. Instead, we take an on-demand calculation strategy that achieves better performance even though it allows duplicate computations. We applied our method to three unstructured grid datasets with different characteristics. With a GPU, our method achieved up to 36.5 times higher performance for the ray-casting process and 19.7 times higher performance for the whole volume rendering process compared with the Bunyk algorithm using a CPU core. Also, our approach showed up to 8.2 times higher performance than a GPU-based cell projection method while generating more accurate rendering results. These results demonstrate the efficiency and accuracy of our method.

KEYWORDS
GPU, ray-casting, scientific visualization, unstructured grid, volume rendering

1 | INTRODUCTION

Direct volume rendering (DVR) is one of the most fundamental visualization methods and is widely used in various fields, including medical imaging and scientific simulations. Ray-casting is the most popular algorithm for DVR (hereafter referred to as "volume rendering") due to its generality and accuracy. For a uniform grid with a regular structure, ray-casting is relatively simple to implement, and various acceleration techniques have been well studied, including parallel algorithms using multi-core CPUs or GPUs [1]. Also, texture-based volume rendering methods efficiently utilizing modern GPU architectures have been proposed [2,3].

Unlike uniform grids, unstructured grids have irregular structures. Both the topology and geometry are completely unstructured, which makes it more complicated to perform

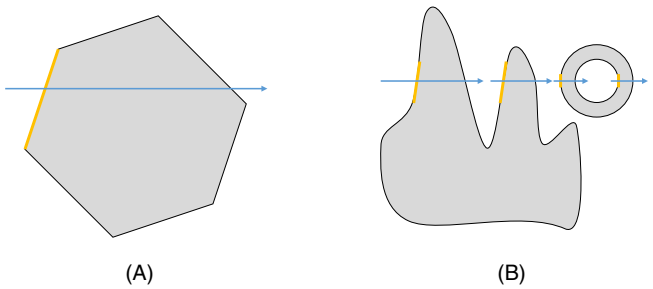

FIGURE 1 This figure illustrates the difference between the two types of meshes for the ray-casing process. Different from convex meshes, with non-convex meshes, a ray can have multiple entry points and pass through empty space in-between them: (A) Convex mesh and (B) non- convex mesh

ray-casting, requiring a significant amount of computation [4,5]. To handle such high computational overhead, cell projection methods have been proposed [6–10]. Cell projection methods can directly utilize the high-performance polygon rasterization function of the GPU, and they show much higher performance than CPU ray-casting methods. However, the rendering results can include artifacts depending on the accuracy of visibility ordering. There have been attempts to implement ray-casting algorithms on the GPU to perform accurate volume rendering while exploiting the GPU's computing power [11,12]. One recent work [13] proposed a GPU ray-casting algorithm (called VF-GPU) for a modern general purpose GPU (GPGPU) architecture, and it achieved up to five times higher performance than CPU-based ray-casting. It casts rays to each visible face (ie, the entry points of rays) and processes them concurrently. However, their approach does not guarantee accurate results for non-convex meshes (Figure 1). This is because multiple visible faces can be projected to the same pixels (eg, bold yellow lines in Figure 1), and more than one ray per pixel can be processed concurrently without considering the order among them. Also, we found that VF-GPU achieves limited performance improvement on current GPUs with different characteristics (eg, L1/L2 caches and much more cores) compared with GPUs used when it was developed.

1.1 | Main contribution

We present a novel GPU ray-casting algorithm that generates accurate volume rendering results for both convex and non-convex meshes. To support non-convex meshes, we employ the per-pixel intersection list concept proposed by Bunyk and others [14]. In our method, we represent the intersection lists as an array-style data structure preferred by the GPU architecture. To efficiently generate the array-style intersection lists, we propose a novel construction algorithm (Section 4). By extending an image-tile-based work decomposition approach by Kim [15], we also introduce a novel work distribution method that improves the memory and cache access efficiency on modern GPU architectures (Section 5.1). Also, we show that the on-demand view-dependent face information (VDFI) calculation approach fits the GPU better than using the per-thread local buffer employed by Maximo and others [13] to reduce duplicate computations, even though the on-demand strategy leads to duplicated work (Section 6.2). Therefore, our GPU ray-casting method adopts this on-demand VDFI computing strategy (Section 5.2).

To demonstrate the benefits of our method, we applied it to three unstructured grid datasets with different sizes and characteristics (Figures 2 and 3). Our GPU ray-casting algorithm showed up to 36.5 times higher performance for the ray-casting process and 19.7 times higher performance for the entire volume rendering process compared to a CPU ray-casting method (the Bunyk algorithm [14]). Also, it generated accurate rendering results for non-convex meshes, as with the Bunyk algorithm. Our method also achieved up to 8.2 times higher performance while generating more accurate results compared with hardware assistant visibility sorting (HAVS) [9], which is a well-known GPU-based cell projection algorithm. These results demonstrate the accuracy and efficiency of our method.

2 | RELATED WORK

Cell projection is one of the widely used approaches for volume rendering of unstructured grid data. To perform volume rendering, it decomposes the cells into a set of polyhedrons (eg, tetrahedrons) and projects the faces of the polyhedrons to an image plane. The projection process can exploit the triangle rasterization performance of graphics hardware (ie, a GPU), and GPU-based cell projection algorithms have been actively proposed [6–9]. In cell projection methods, the rendering quality depends on the accuracy of visibility sorting of the faces projected on the same pixel. Callahan and others [9] proposed the HAVS algorithm, which accelerates visibility sorting by using the GPU, and it showed much higher performance than CPU-based sorting. However, the sorting accuracy was limited by the size of the k-buffer, and it also had a race condition problem for simultaneous texture memory access. Therefore, the rendering results of HAVS can contain artifacts.

Unlike cell projection methods, the ray-casting algorithm guarantees accurate rendering results, but it requires a large amount of computation. Garrity [16] implemented a ray-casting method for unstructured grid data based on cell connectivity information. Bunyk and others [14] improved the performance by computing VDFI for all faces in the preprocessing step and using them during the ray-casting process. As a result, they could remove duplicated computations, which dramatically improved the ray-casting performance. However, it required lots of memory and was difficult to apply to a large-scale dataset. Riberio and others [17] solved the

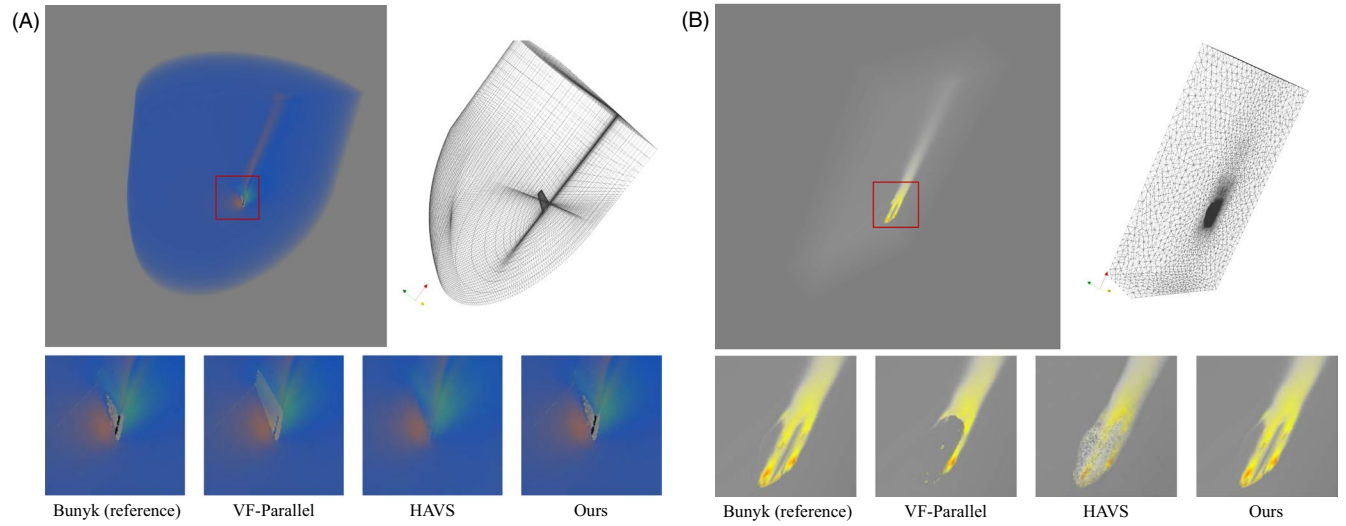

FIGURE 2 Top images show the volume rendering results of our method and shapes of the grid structure for two benchmark datasets. The bottom images compare the rendering results of four different algorithms: (A) Dataset 1 and (B) Dataset 3

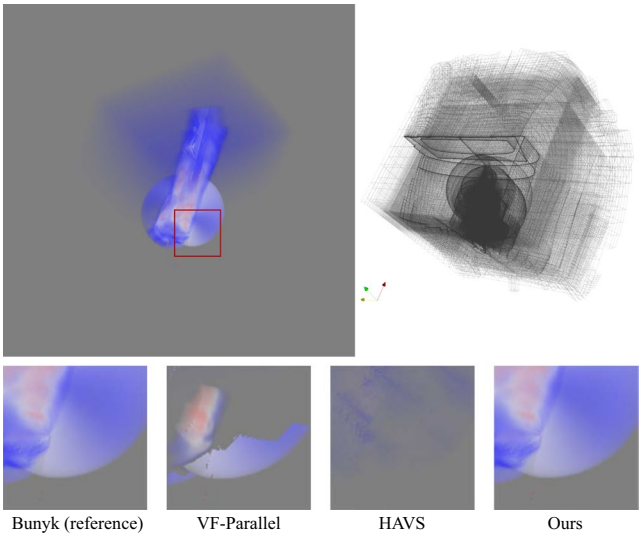

FIGURE 3 Top images show the rendering results of our method and the shape of the grid structure for Dataset 2. The bottom images compare the rendering results of four different algorithms

memory overhead by using a fixed-size VDFI buffer instead of maintaining VDFIs for all faces. Their Visible Face Driven Ray-Casting (VF-Ray) method achieved a performance close to that of Bunyk and others by exploiting ray-coherency, where nearby rays tend to visit similar faces. Recently, Kim [15] further reduced memory usage. They proposed a novel hash function based on face ID, which improves the hit ratio for a small-sized VDFI buffer. That work also presented a parallel ray-casting algorithm on multi-core CPUs. Different from VF-Ray, Kim's method comprised the ray group based on the image plane. More specifically, it divides the image plane as a set of image-tiles and distributes tiles to threads, where each thread processes all rays in the given tiles. As a result, it achieved a five-fold performance improvement with eight CPU cores over a serial algorithm.

Our ray-casting method is also based on the Bunyk algorithm. However, unlike the CPU-based ray-casting methods, we employ the computing power of the GPU for the ray-casting process. Also, our method employs the image-tile concept [15] for work distributions, but we extend it to utilize modern GPU architectures efficiently (Section 5.1).

Weiler and others [11] implemented the ray-casting algorithm of Garrity [16] on a GPU. However, the method does not support non-convex meshes. Bernardon and others [12] solved this problem with a depth-feeling algorithm, and then Espinha and Celes [18] improved the rendering quality. These early works implemented their methods based on fragment shader languages, and we can generally apply them to most GPUs. However, they must take a rendering pipeline, and this limits the utilization efficiency of modern GPU architecture developed for GPGPU. Maximo and others [13] extended VF-Ray [17] to a GPU-based parallel algorithm (called VF-GPU) while using CUDA [19], which is a programming API for GPGPU. Like VF-Ray, it forms ray groups based on visible faces and distributes them to the thread blocks of a GPU. Each thread block processes the given ray groups, and each thread in a block handles multiple rays. Also, VF-GPU reduces duplicate VDFI computations by employing the per-thread VDFI buffer used in VF-Ray. As a result, it achieved approximately 3–5 times higher volume rendering performance than CPU methods. To the best of our knowledge, VF-GPU is still one of the state-of-the-art algorithms for GPU-based ray-casting for unstructured grid data from the perspective of performance. However, we found that using the VDFI buffer is not a suitable approach for modern GPU architectures, which have improved over the last decade (Section

5.2). Also, Kim [15] showed that the visible face-based ray distribution can generate artifacts for non-convex meshes.

We designed our algorithm to efficiently utilize modern GPU architecture with our GPU-based ray-casting algorithm (Section 5). Also, our method guarantees correct rendering results for non-convex meshes by employing the per-pixel intersection list concept of Bunyk and others [14] (Section 4).

For the uniform grid, texture-based volume rendering methods have actively employed GPUs [2,3,20,21]. The uniform grid can be stored directly in the GPU as a texture, and a dedicated cache is available for texture on modern GPUs. To efficiently utilize the texture cache, Sugimoto and others [2] localized texture memory reference according to the viewpoint. They also dynamically adjusted the thread block layout and incorporated transposed thread indexing for more optimal access to the texture cache. Wang and others [3] presented a novel sampling strategy called Warp Marching, which improves the hit ratio for both texture and L2 caches. As a result, Warp Marching outperformed empty space skipping approaches while showing stable performance for viewing direction changes. Unfortunately, such prior cache-friendly methods for texture-based volume rendering cannot be directly applied to unstructured grid data. Different from the uniform grid, it is difficult to expect which cells are visited by a ray because both topology and geometry are completely unstructured. Also, the unstructured grid consisting of meshes (ie, tetrahedrons) is stored in the device memory of the GPU, not in the texture memory.

Instead, we exploit ray-coherency to improve L1/L2 cache utilization efficiency for device memory access (Section 5.1).

3 | OVERVIEW

We first provide background on the Bunyk algorithm and modern GPU architecture with its programming model. Then, we describe our volume rendering system.

3.1 | Bunyk Algorithm

There are two categories of information required for the Bunyk algorithm [14]: view-independent and view-dependent information. View-independent information includes geometric information, such as cell connectivity, tetrahedrons in each cell, and faces composing each tetrahedron. These are essential pieces of information for ray-casting for unstructured grids, and most ray-casting algorithms commonly maintain this geometric information. Unless the geometry is changed, it does not require re-computation for viewpoint change.

View-dependent information includes per-pixel intersection lists and view-dependent face information. For each pixel, the Bunyk algorithm maintains an intersection list that is a set of boundary faces intersected by a ray from the pixel. Boundary face means a face included by only one tetrahedron and exposed outside. Each list is also sorted by the distance between faces and the pixel (ie, depth). This information is used to identify entry-points (ie, boundary faces) of a ray cast from the pixel, and the ray can find the next entry-point once it exits from a non-convex mesh. Thanks to this intersection list, the Bunyk algorithm guarantees accurate volume rendering for non-convex meshes.

The VDFI includes a set of coefficients used for the ray-face intersection test. The ray-face intersection test is a primary operation for ray traversal on an unstructured grid, and it can be calculated multiple times because many rays can visit the same face. The Bunyk algorithm computes the VDFIs for all faces for a given viewpoint and maintains them in the memory. With the precomputed VDFIs, the Bunyk algorithm avoids duplicate computation and improves the ray-casting performance.

3.2 | GPU architecture and programming model

Although the architecture of GPUs can differ depending on vendors, recent GPUs share many basic architecture concepts and programming models. We briefly introduce the architecture and programming model (ie, CUDA) of modern GPUs based on NVIDIA's GPU, which is one that is widely used for GPGPU.

The basic building block of the GPU is a streaming multiprocessor (SM). A GPU consists of multiple SMs (eg, 20–80 SMs), and each SM includes a set of GPU cores (eg, 32–128 cores). In the CUDA programming model, a set of threads compose a thread block, and an SM handles a thread block. The current GPU architecture is called SIMT (Single Instruction, Multiple Thread). In SIMT, an instruction controls thirty-two threads (ie, a warp). When the threads in a warp are diverged by a conditional branch, the work of the threads is serialized. Such warp divergence is a well-known performance bottleneck for GPU algorithms [19].

Each SM has its own on-chip resources, and thread blocks share them. Some of the resources (eg, registers) are disjointly allocated to thread blocks. When the resources are not sufficient to support all thread blocks, some of them work first. A thread block that gets all required resources and can do its work is called an active block. The number of active blocks is one of the critical factors for improving the computation efficiency (eg, instruction throughput) of a GPU.

The GPU stores data in the device memory (global memory of the GPU). Because device memory access is slow, GPU algorithms usually launch a massive number of threads (eg, tens or hundreds of thousands of threads) to hide the latency [19]. Recent GPUs have an L2 cache, and they access the device memory through the L2 cache. If the memory

access region of all threads in a warp belongs to the same cache line, only one L2 transaction is required. However, it requires multiple L2 transactions (eg, up to 32) for an irregular memory access pattern in a warp. Therefore, localizing the memory access regions of threads in a warp is also essential to achieve high performance [22].

An SM also has a fast but small (eg, 64 KB) memory space that is shared by all threads in a thread block (ie, shared memory). We also call this a user-managed cache, and the most popular usage puts data used frequently by threads in a thread block to decrease memory access latency. However, it is not easy to apply this usage when it is difficult to predict frequently used data like the ray-casting algorithm. Fortunately, we can configure the shared memory space as a hardware-managed L1 cache in a recent GPU architecture [22]. Therefore, we can improve the performance by guiding threads in a block to access adjacent memory areas because this increases memory access locality.

3.3 | System overview

Figure 4 shows an overview of our volume rendering system. Our method employs the per-pixel intersection list concept of the Bunyk algorithm [14] to guarantee accurate rendering results for non-convex meshes. We construct the intersection lists on the CPU-side. For efficient data transfer to the device memory and efficient memory access in the GPU, we make the intersection lists as an array-style data structure using our construction algorithm (Section 4). Once the GPU gets view-dependent information (eg, intersection lists), it performs ray-casting for all rays in parallel (Section 5). To improve ray-coherency among rays in a thread block, we make ray groups based on an image-tile. We distribute the ray groups to available thread blocks, and the threads in a block handle the given rays (Section 5.1). The ray-casting process for a ray follows the Bunyk algorithm, but we do not maintain VDFIs for all faces. Instead, we take an on-demand VDFI calculation strategy because we found that it is the appropriate approach for recent GPU architecture (Section 5.2).

4 | ARRAY-BASED INTERSECTION LIST CONSTRUCTION

To identify the boundary faces intersected by each ray, we project all boundary faces to the image plane. A straightforward approach for building per-pixel intersection lists is to use a linked list for each pixel. For a given boundary face, it adds the face ID to the lists of the pixels the face projected, while sorting the lists by depth with the insertion sort. However, the linked list is not a preferred data structure for a GPU because it leads to an irregular memory access pattern (Section 3.2). Also, transferring linked lists to the device memory requires a complicated process, like allocating memory and copying data for each element one-by-one. Therefore, it is better to convert the linked lists to a set of arrays. However, the conversion process is also complicated because all nodes must be visited one-by-one.

To efficiently build per-pixel intersection lists in the form of arrays, we propose a novel intersection list construction algorithm (Figure 5). Our method uses two arrays: $\text{Array}_{\text{pixel}}$ containing the pixel's ID (eg, x and y) and $\text{Array}_{\text{intersect}}$, whose elements consist of the face ID and depth from a pixel. Our construction algorithm follows three steps. (a) In the first step, we project all boundary faces to the image plane sequentially. After projecting a face, for all pixels projected by the face, we add the pixel's ID to $\text{Array}_{\text{pixel}}$ and the face ID and depth to $\text{Array}_{\text{intersect}}$. After this step, the two arrays hold all intersection information for all pixels, but they are not ordered yet. (b) In the second step, we sort the arrays by pixel ID, and this gathers intersection information for each pixel in one place. (c) Then, we make a new array called $\text{Array}_{\text{list}}$, whose size is the same as the number of pixels. Each element in $\text{Array}_{\text{list}}$ has meta-information to access the intersection list for a pixel directly. The meta-information includes the start index of $\text{Array}_{\text{intersect}}$ and the length of the intersection list of a pixel. Finally, we sort the intersection lists for each pixel by depth, and we get the complete intersection lists $\text{Array}_{\text{intersect}}$ and $\text{Array}_{\text{list}}$. We can do the last sorting in parallel because the lists for each pixel are independent of each other.

4.1 | Construction performance

Compared with the linked list-based method, our construction algorithm took about 86%, 14%, and 2% more time for the construction step in the three benchmarks, respectively. However, we found that the performance gap decreased as the size of the dataset increased, and the difference was tens of milliseconds numerically (eg, 33.5 milliseconds for Dataset 1). Also, a CPU ray-casting algorithm (eg, Bunyk algorithm) with our array-based intersection list showed better performance than using a linked list. This is because accessing elements in a linked list requires additional memory access for reading the link compared

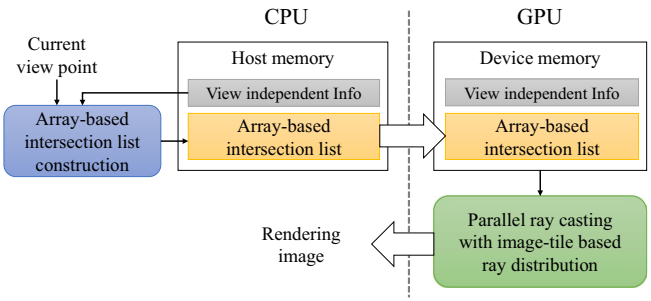

FIGURE 4 Overview of our volume rendering system

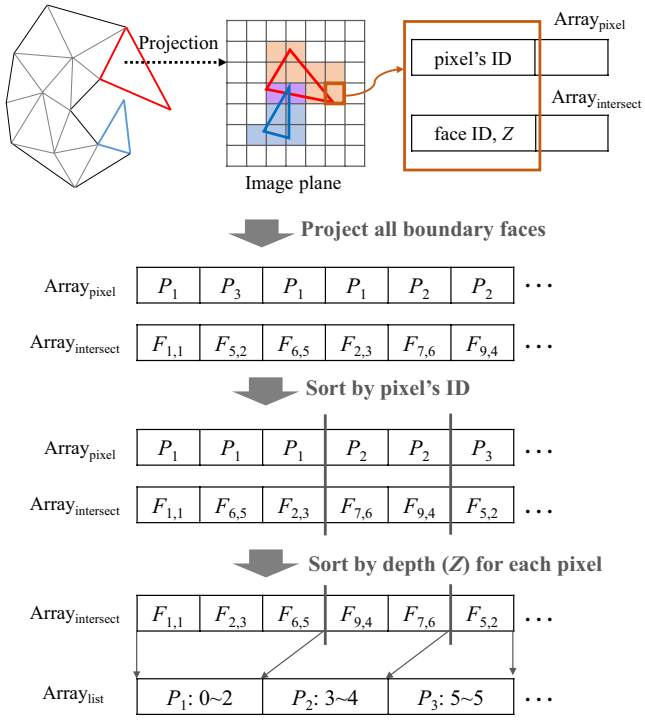

FIGURE 5 This figure shows the process of our array-based intersection list construction algorithm. P_i represents the pixel's ID, and $F_{j,k}$ is the face's ID and depth value

with an array-based list. As a result, it achieved a performance (±1%) that was compatible with using a linked list for the entire volume rendering process.

5 | GPU-BASED RAY-CASTING ALGORITHM

Rays are independent of each other, and the GPU processes rays in parallel. In this section, we explain how we distribute rays to threads while considering ray coherency for more efficient memory access (Section 5.1). Then, we explain why we employ the on-demand VDFI computation strategy (Section 5.2).

5.1 | Image-tile-based ray distribution

To increase the utilization efficiency of the L1 and L2 caches, we need to improve the memory access locality of threads in a thread block (Section 3.2). For the ray-casting process, to get a high locality, we can guide by increasing ray-coherency among the rays in a group. VF-GPU [13] distributes rays based on visibility faces to increase the ray-coherency processed by a thread block. However, this approach can generate artifacts with non-convex meshes

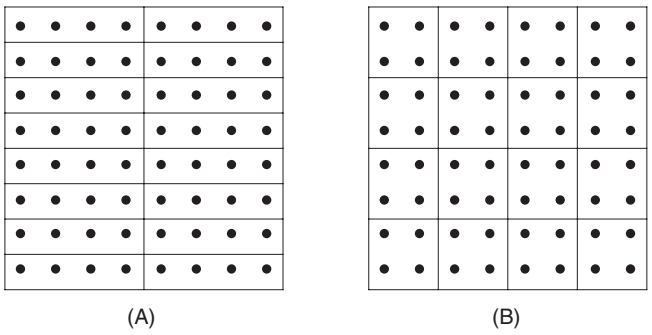

FIGURE 6 These figures show the two ray distribution methods when a ray group consists of four rays. (A) Pixel-order ray distribution and (B) Image-Tile based distribution

because multiple visible faces can be projected to the same pixel [15].

To guarantee accurate results for non-convex meshes, we make ray groups based on the image plane. One simple ray distribution scheme using the image plane is to allocate rays to threads according to the order of pixels (Figure 6A). However, rays processed by a thread block produce low coherence and lower cache utilization efficiency.

To increase ray-coherency among rays in a thread block, we employ the image-tile used by Kim [15]. An image-tile is a square image with a certain size (eg, 8 by 8), as shown in Figure 6B. In our method, rays from an image-tile comprise a ray group similar to the work by Kim. However, we distribute them to thread blocks evenly, unlike Kim, who allocated an image-tile to a thread. To exploit parallelism more, we also use a 2D thread block whose size is the same as the size of an image-tile, and each thread in a block handles a ray. We found that our image-tile-based ray distribution method requires fewer cache transactions compared with using a simple pixel-order ray distribution approach (Section 6.1).

5.2 | On-demand VDFI computation

To solve the high memory overhead of the Bunyk algorithm [14] while reducing duplicate VDFI calculations, VF-GPU [13] employed a thread-local VDFI buffer. In VF-GPU, a thread handles multiple rays and reuses the VDFI of a face if it has already computed it for prior rays. However, we found that this approach does not meet the properties of recent GPU architectures from two perspectives.

First, a thread should handle multiple rays to take advantage of the VDFI buffer, but it must launch a large number of threads (eg, a ray per thread) to fully utilize the computing power of a modern GPU. Secondly, GPU threads run in a unit of a warp, and their work is serialized if there is divergence (Section 3.2). We can take advantage of the VDFI buffer only when all threads in a warp find the required information in

the buffer (ie, buffer hit). If one or more threads in a warp cannot find the necessary information in the buffer (ie, buffer miss), warp serialization invalidates the effect of the buffer hits for other threads. In this case, checking the buffer becomes overhead. Therefore, it is difficult to achieve any performance improvement with a thread-local buffer unless the size of the buffer is large enough to show a high buffer hit ratio for all threads.

Unlike the legacy GPUs used in the days when VF-GPU was proposed (eg, GeForce 8800 Ultra, 128 cores, 768 MB), recent GPUs have thousands of cores (eg, Nvidia GTX 1080, 2560 cores, 8 GB). To fully exploit the massive parallelism of current GPUs, we need to launch tens or hundreds of thousands of threads. Although the size of the device memory also has increased, it does not meet the incremental number of cores, while the size of unstructured grid data has also increased. In the implementation of VTK [5], the VDFI for a face takes 72 bytes. If we allocate 256 slots for each buffer, which is the size that achieved a hit ratio of approximately 70% for Kim [15], it would require a few gigabytes (eg, 4 GB) when we launch hundreds of thousands of threads (eg, 262 144 threads). Therefore, the possible buffer size is limited to a few tens of slots, and it is hard to expect a high hit ratio for all threads.

In our experiments, we also found that calculating VDFI whenever a ray meets a face achieves better performance than using a thread-local buffer, even though it can result in duplicate computation (Section 6.2). Based on these theoretical and experimental observations, we use the on-demand VDFI calculation strategy in our GPU ray casting algorithm.

6 | IMPLEMENTATION AND RESULTS

We implemented our volume rendering system on a Windows system consisting of a GPU (Nvidia GTX 1080, 2560 cores, 8 GB) and a quad-core CPU (Intel i7-8700) with 32 GB main memory. We used OpenMP [23] and CUDA [19] to implement parallel algorithms on the CPU and GPU, respectively. We implemented our GPU ray-casting algorithm in two versions, depending on the ray distribution method:

- *Ours-Pixel* is one implementation of our approach. In this method, we take the on-demand VDFI computation strategy and use the pixel-order to make ray groups (Figure 6A). We composed a thread block of 64 threads, and we used 262 144 threads (4096 thread blocks).
- *Ours-Tile* was implemented by replacing the ray distribution method used in *Ours-Pixel* with our image-tile-based approach (Figure 6B) with tiles of size 8 by 8.

To compare the efficiency and accuracy of our approach, we also implemented four alternative methods based on prior work:

- *Bunyk* is an implementation of Bunyk and others [14] in VTK. We used a single CPU core for this algorithm.
- *Bunyk-Parallel* is an implementation of a CPU-based parallel Bunyk algorithm based on Kim [15]. It computes VDFIs for all faces as a pre-processing step and uses them during the ray-casting process. We used four CPU threads for this method.
- *VF-Parallel* is a CPU version of Maximo and others [13]. It shoots rays for each visible face while using the face as the entry-point. We used four CPU threads for this method, and it processes multiple visible faces in parallel.
- *HAVS* is one of the most well-known GPU-based cell projection methods proposed by Callahan and others [9]. We employed the implementation of VTK 6.3 because later VTK versions do not include this method.

All of the algorithms were implemented as VTK filters and were run in the VTK visualization framework [5].

Benchmarks: We tested different volume rendering methods against three different benchmarks (Table 1). Datasets 1, 2, and 3 are CFD simulation results of fluid flow around an airplane's empennage, a ship's propeller, and a car, respectively. We used the velocity magnitude for Dataset 1 and 2 and the turbulence kinetic energy for Dataset 3 as the scalar value for volume rendering. For all benchmarks, we used 1024 by 1024 image resolution and measured the volume rendering time for 72 viewpoints by rotating five degrees around the given axis of rotation.

6.1 | Results and analysis

Table 2 lists the average volume rendering times per frame of the six different algorithms. Please note that the volume rendering times in the table includes times for both the intersection list construction and ray-casting process. *Ours-Pixel* and *Ours-Tile* respectively showed up to 15.4 and 19.7 times higher performance than *Bunyk*, which used a CPU core. Compared with *Bunyk-Parallel*, *Ours-Pixel* and *Ours-Tile* achieved up to 4.5 and 5.8 times (3.0 and 3.6 times on average) higher performances, respectively. For Dataset 3, our methods showed relatively lower performance improvements because the intersection list construction running on CPU takes about 0.9 seconds, which is about 35% of the entire volume rendering time of *Bunyk-Parallel*. *VF-Parallel* showed a comparable or slightly higher performance compared with *Bunyk-Parallel*.

HAVS, which uses the same GPU as our methods, also showed up to 4.1 and 1.2 times higher performances

TABLE 1 This table shows the different statics of each benchmark. Each column shows the number of vertices, faces, tetrahedrons, and boundary faces along with the required memory space for storing the information

		Vertices	Faces	Tetras	Boundaries
Dataset 1 Figure 2A	#	647 073	7 583 488	3 773 943	71 614
	Mem. (MB)	5	364	121	0.6
Dataset 2 Figure 3	#	7 320 994	83 385 432	41 257 368	1 741 392
	Mem. (MB)	56	3,817	1,259	13
Dataset 3 Figure 2B	#	7 659 517	85 051 683	41 939 085	2 347 026
	Mem. (MB)	61	4,082	1,342	19

TABLE 2 This table shows the average rendering time (in seconds) per frame for three benchmark datasets

(seconds)	Bunyk	Bunyk-Parallel	VF-Parallel	HAVS	Ours-Pixel	Ours-Tile
Dataset 1	7.21	2.12	2.25	0.51	0.47	0.37
Dataset 2	50.65	15.52	14.46	12.44	5.57	4.83
Dataset 3	5.31	2.53	2.03	10.62	1.33	1.29

than *Bunyk-Parallel* for Datasets 1 and 2, respectively. However, our methods generally achieved better performance than *HAVS*. For example, *Ours-Tile* had 1.4, 2.6, and 8.2 times higher performance for Datasets 1, 2, and 3, respectively. These results demonstrate the benefits of our approach. We were able to achieve such high volume rendering performance because our GPU ray-casting algorithm properly considers the characteristics of modern GPU architecture. In Dataset 3, cells are crowded in specific regions. During the ray-casting process, we can terminate the process for a ray when the opacity of the pixel reaches its max by employing the early ray termination method [24]. Therefore, for Dataset 3, even *Bunyk-Parallel* showed better performance than HAVS, which had to consider all faces.

6.1.1 | Rendering accuracy

The bottom images in Figures 2 and 3 compare the rendering results for *Bunyk* (reference), *VF-Parallel*, *HAVS*, and our methods on the three benchmark datasets. *VF-Parallel* failed to handle the non-convex region and made some false holes because multiple rays for the same pixel with different entry-points were processed simultaneously without any synchronization. Although *HAVS* sorts faces projected to a pixel by depth, the sorting results can be incorrect depending on the size of the k-buffer and the synchronization issue on texture memory access, as mentioned by Callahan and others [9]. As a result, we can observe artifacts in the rendering results, especially for the dense and complex area. On the other hand, our methods showed the same results as the *Bunyk* (reference) by implementing an accurate ray-casting algorithm on the GPU.

6.1.2 | Ray-casting performance

To check the benefits of our GPU ray-casting algorithm, we measured the processing times just for the ray-casting process, while excluding common processing times on the CPU, such as the intersection list construction time. For *Bunyk*, *Bunyk-Parallel*, and *VF-Parallel*, we included the preprocessing time in which VDFIs are computed for all faces. We included the data communication time between the CPU and GPU for our methods. We did not consider *HAVS* for this comparison because it is a cell projection method, not a ray-casting-based algorithm. Figure 7 shows the performance improvement that each algorithm achieved over *Bunyk* (baseline). *Ours-Tile* achieved up to 36.5 times (27.1 times on average) higher ray-casting performance than *Bunyk*. It also had 7.1, 3.7, and 12.4 times higher performance than *Bunyk-Parallel* for Datasets 1, 2, and 3, respectively. *VF-Parallel* showed comparable performance with *Bunyk-Parallel*, and *Ours-Tile* achieved 7.3, 3.6, and

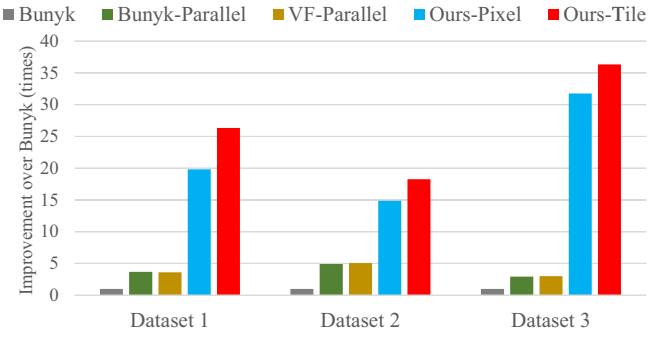

FIGURE 7 This figure shows the performance improvements for the ray-casting process of five algorithms over *Bunyk* (baseline)

Tile size	(8, 4)	(8, 8)	(16, 8)	(16, 16)	(32, 16)
Dataset 1	283.06	**272.17**	279.84	277.93	387.17
Dataset 2	3872.34	3813.82	3896.33	**3802.77**	4684.05
Dataset 3	117.46	**115.73**	119.06	120.72	157.01

Note: The bold font indicates the best results.

TABLE 3 This table shows the ray-casting time (in milliseconds) according to the image-tile size

12.1 times higher performance than *VF-Parallel* Datasets 1, 2, and 3, respectively. More specifically, our methods showed much higher performance improvement for Dataset 3. This is because rays visit a small ratio of faces during the actual ray-casting process, and the preprocessing time for computing VDFIs for all faces becomes overhead.

Compared with *Ours-Pixel*, *Ours-Tile* achieved 33%, 23%, and 14% better ray-casting performance for Datasets 1, 2, and 3, respectively. We profiled two ray-casting kernels with the Nsight Profiler [22] and found that *Our-Tile* took less L2 cache transactions than *Our-Pixel* (eg, 21% fewer transactions for Dataset 1). This result demonstrates that our image-tile-based ray distribution approach increases ray-coherency in a thread block and actually improves cache utilization efficiency.

6.1.3 | Image-tile size

We also analyzed the effect of the size of the tile using the Nsight Profiler. Table 3 is the average ray-casting time with different tile sizes. As shown in the results, the 8 by 8 tile generally shows good performance. We found that a large image-tile size lowers the degree of parallelism (eg, instruction throughput) because it requires a large number of resources (eg, register) for a thread block, and it decreases the number of active blocks. For the 8 by 4 (or smaller) image-tiles, on the other hand, each thread block includes only a warp, and it did not fully utilize the available resources in an SM.

6.2 | Benefits of on-demand VDFI computation

Dataset 1 is small enough to load VDFIs for all faces with the device memory of the GTX 1080. To validate the efficiency of our on-demand VDFI computation strategy compared with possible alternative GPU algorithms, including VF-GPU [13], we also implemented the following two algorithms:

- *GBunyk-Buffer* uses a VDFI buffer for each thread. Because a thread should handle multiple rays to take advantage of the VDFI buffer, we allocated an image-tile to a thread. However, because a large number of rays for a thread lowers the parallelism, we used a 4 by 4 tile. Please note that this also represents an improved version of VF-GPU [13] to support non-convex meshes.
- *FullVDFI* is a modified version of *Our-Tile*. It pre-computes VDFIs for all faces and uses them during ray-casting. It also performs VDFI computation on the GPU and maintains the results in the device memory. The VDFI computation took approximately 23 ms for Dataset 1. Please note that this method does not work for Datasets 2 and 3 because the device memory is not sufficient to handle these large datasets.

We measured the ray-casting time for the two algorithms and *Ours-Tile*. Figure 8 shows the results. For *GBunyk-Buffer*, we varied the number of slots of each buffer from 4 to 256 (the largest the device memory allows). Because *FullVDFI* removes redundant computation for VDFI, it showed about 14% better performance than *Ours-Tile*. However, *GBunyk-Buffer* exhibited much lower performance compared with *Ours-Tile*. Moreover the performance degraded as the size of the buffer increased. Table 4 lists the performance profiling results of the two algorithms for a viewpoint. We found that *GBunyk-Buffer* with 256 slots had approximately 1.5 times more memory access requests and up to 2.7 times more L2 cache transactions compared with *Ours-Tile*. The lower control-flow efficiency (one of the branch statics) for *GBunyk-Buffer* shows that the size of the buffer was not sufficient to achieve a high hit ratio for all threads. Therefore, checking the buffer becomes additional overhead, which leads to a large number of memory (or cache) transactions. We found that the L1 and L2 cache hit ratios also decreased by about 30.9% and 11.9% compared with our on-demand strategy,

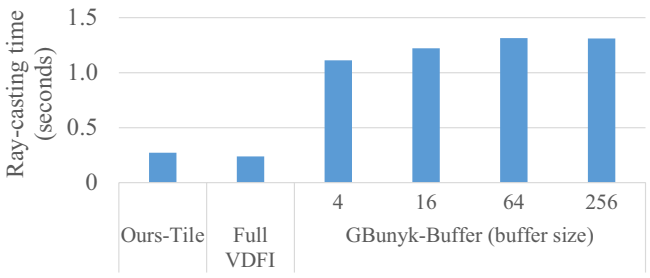

FIGURE 8 This figure shows the ray-casting time on Dataset 1 by *Ours-Tile*, *FullVDFI*, and *GBunyk-Buffer* with different buffer sizes

TABLE 4 This table shows the performance profiling results of two algorithms for Dataset 1. Detailed definitions of each metric are available in the Nsight Profiler user guide [25]

	L1 cache hit ratio	L2 cache hit ratio	Control-flow efficiency	Active warps per SM
Ours-Tile	38.49%	75.70%	78.20%	15.04
Gbunyk-Buffer (256 slots)	7.54%	63.80%	48.50%	13.75

respectively. This is because as the size of the buffer per thread increased, the cache utilization efficiency decreased. Such memory access efficiency affects the computational efficiency too, and *Ours-Tile* showed 1.3 times more active warps per SM than *GBunyk-Buffer*. These results confirm that the on-demand VDFI computation strategy is a better choice for modern GPU architectures than using VDFI buffers.

7 | CONCLUSIONS AND FUTURE WORK

We have presented a GPU ray-casting algorithm for volume rendering of unstructured grid data. Our method employs per-pixel intersection lists to guarantee accurate rendering results for non-convex meshes. We have presented an array-based intersection list construction algorithm to access the lists on the GPU efficiently. To increase ray-coherency in a thread block and improve cache utilization efficiency, we have proposed an image-tile-based ray distribution method. We have also shown that for a modern GPU architecture, our on-demand VDFI computation strategy achieves better performance than using thread-local VDFI buffers to reduce duplicate calculations. For three different unstructured grid datasets, our method achieved up to 19.7 times higher volume rendering performance than a serial ray-casting method running on a CPU. Our method also showed up to 8.2 times higher performance than a GPU-based cell projection method while generating more accurate results. These results demonstrate the benefits of our approaches.

However, our method exhibited limited performance improvement when the intersection list construction took a large portion of the entire volume rendering time (eg, Dataset 3) because the CPU is responsible for this. As future work, we would like to design a more scalable algorithm while making the GPU build the intersection list. We also would like to extend our method to an out-of-core algorithm to handle a large dataset whose size is larger than the device memory.

ORCID

Duksu Kim https://orcid.org/0000-0002-9075-3983

REFERENCES

1. K. Brodlie and J. Wood, *Recent advances in volume visualization*, Comput. Graph. Forum **20** (2001), no. 2, 125–148.
2. Y. Sugimoto, F. Ino, and K. Hagihara, *Improving cache locality for gpu-based volume rendering*, Parallel Comput. **40** (2014), no. 5–6, 59–69.
3. J. Wang, F. Yang, and Y. Cao, *A cache-friendly sampling strategy for texture-based volume rendering on gpu*, Visual Inf. **1** (2017), no. 2, 92–105.
4. C. T. Silva et al., *A survey of gpu-based volume rendering of unstructured grids, Revista de informática teórica e aplicada*. Porto Alegre, RS. **12** (2005), no. 2, 9–29.
5. W. Schroeder, K. M. Martin, and W. E. Lorensen, The visualization toolkit (4th ed.): an object-oriented approach to 3d graphics, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
6. B. Wylie et al., *Tetrahedral projection using vertex shaders*, in Proc. IEEE Symp. Volume Visualization Graph. (Boston, MA, USA), Oct. 2002, pp. 7–12.
7. R. Marroquim et al., *GPU-based cell projection for interactive volume rendering*, in Proc. Brazilian Symp. Comput. Graph. Image Process. (Manaus, Brazil), 2006, pp. 147–154.
8. M. Weiler et al., *Hardware-based view-independent cell projection*, IEEE Trans. Visualization Comput. Graph. **9** (2003), no. 2, 163–175.
9. S. P. Callahan et al., *Hardware-assisted visibility sorting for unstructured volume rendering*, IEEE Trans. Visualization Comput. Graph. **11** (2005), no. 3, 285–295.
10. J. S. B. Mitchell, C. T. Silva, and R. Farias, *ZSWEEP: an efficient and exact projection algorithm for unstructured volume rendering*, in Proc. IEEE Symp. Volume Visualization (Salt Lake City, UT, USA), Oct. 2000, pp. 91–99.
11. M. Weiler et al., *Hardware-based ray casting for tetrahedral meshes*, in Proc. IEEE Visualization (Seattle, WA, USA), Oct. 2003, pp. 333–340.
12. F. F. Bernardon et al., *GPU-based tiled ray casting using depth peeling*, J. Graph. Tools **11** (2006), no. 4, 1–16.
13. A. Maximo et al., *Memory efficient GPU-based ray casting for unstructured volume rendering*, in Proc. Eurographics/IEEE VGTC Conf. Point-Based Graph. (Goslar, Germany), 2008, pp. 155–162.
14. P. Bunyk, A. Kaufman, and C. T. Silva, *Simple, fast, and robust ray casting of irregular grids*, in Proc. Scientific Visualization Conf. (Dagstuhl, Germany), June 1997, pp. 1–7.
15. D. Kim, *Memory efficient parallel ray-casting algorithm for unstructured grid volume rendering*, in Proc. Eurographics/IEEE VGTC Conf. Visualization: Posters (Goslar, Germany), 2017, pp. 13–15.
16. M. P. Garrity, *Raytracing irregular volume data*, ACM SIGGRAPH Comput. Graph. **24** (1990), no. 5, 35–40.
17. S. Ribeiro et al., *Memory-aware and efficient ray-casting algorithm*, in Proc. Brazilian Symp. Comput. Graph. Image Process. (Minas Gerais, Brazil), Oct. 2007, pp. 147–154.

18. R. Espinha and W. Celes, *High-quality hardware-based ray-casting volume rendering using partial pre-integration*, in Proc. Brazilian Symp. Comput. Graph. Image Process. (Rio Grande do Norte, Brazil), Oct. 2005, pp. 273–280.
19. NVIDIA, *CUDA programming guide 9.2*, NVIDIA, Santa Clara, CA, 2018.
20. J. Wang, F. Yang, and Y. Cao, *Cache-aware sampling strategies for texture-based ray casting on GPU*, in Proc. IEEE Symp. Large Data Anal. Visualization (Paris, France), Nov. 2014, pp. 19–26.
21. J. Wang, F. Yang, and Y. Cao, *Computation-to-core mapping strategies for iso-surface volume rendering on GPUs*, in Proc. IEEE Pacific Visualization Symp. (Hangzhou, China), Apr. 2015, pp. 153–157.
22. J. Cheng, M. Grossman, and T. McKercher, Professional CUDA C programming, John Wiley & Sons, New York, NY, 2014.
23. L. Dagum and R. Menon, *OpenMP: an industry standard API for shared-memory programming*, IEEE Comput. Sci. Eng. **5** (1998), 46–55.
24. M. Levoy, *Display of surfaces from volume data*, IEEE Comput. graphics Applicat. **8** (1988), no. 3, 29–37.
25. NVIDIA, *Nsight visual studio edition user guide*, NVIDIA, Santa Clara, CA.

AUTHOR BIOGRAPHIES

Gibeom Gu received his BS and MS degrees in computer science and engineering from the School of Engineering, SoGang University, Seoul, Rep. of Korea, in 1997 and 1999, respectively. He is currently a senior researcher at the Center for Development of Supercomputing System, Korea Institute of Science and Technology Information(KISTI). His main research interests are scientific visualization and high-performance computing.

Duksu Kim is currently an assistant professor at the School of Computer Engineering at KOREATECH (Korea University of Technology and Education). He received his BS from SungKyunKwan University in 2008. He received his PhD in Computer Science from KAIST (Korea Advanced Institute of Science and Technology) in 2014. He spent several years as a senior researcher at KISTI National Supercomputing Center. His research interest is designing heterogeneous parallel computing algorithms for various applications, including proximity computation, scientific visualization, and machine learning. He has received the distinguished paper award at Pacific Graphics 2009 and an ACM student research competition award in 2009 and was selected as the spotlight paper for the September issue of IEEE Transactions on Visualization and Computer Graphics (TVCG) in 2013. He is a young professional member of IEEE and a professional member of ACM.