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

Rasterization with Data-Parallel Primitives

Makani Buckley,[1] Kenneth Moreland,[2] and Hank Childs[1]

[1]University of Oregon, USA
[2]Oak Ridge National Laboratory, USA

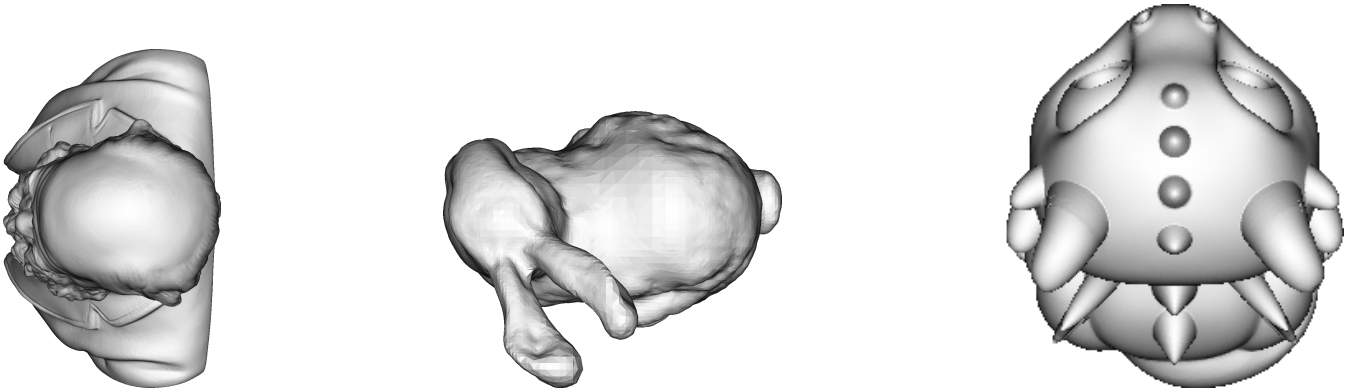

Figure 1: *Sample images rendered by DPP rasterizer.*

Abstract

Parallel rasterization can suffer from race conditions during fragment generation, which is traditionally addressed by using specialized hardware accessible via vendor graphics APIs. Unfortunately, graphics APIs are increasingly problematic on high-performance computers, either because they are not provided or because of concerns about dependencies with in situ visualization. In response, we present a hardware-agnostic rasterization algorithm that handles race conditions using only data-parallel primitives (DPPs), enabling efficient rendering on HPC systems without graphics API dependencies and aligning with recent efforts to deliver visualization software with DPPs. Our evaluation consists of three phases: (1) evaluating portability across different CPU and GPU architectures, (2) evaluating competitiveness with a community standard, and (3) evaluating performance across varying workloads and available parallelism. The supporting experiments run on both AMD and NVIDIA GPUs, considering data sets as large as 460 million triangles and 160 million pixels. While performance generally falls short of graphics API baselines, it achieves interactive frame rates on most workloads. As a result, we conclude our approach is a viable solution for rasterization on high-performance computers since our approach is portably performant across different architectures without the need for specialized vendor support.

1. Introduction

Rasterization is a foundational approach for rendering in computer graphics. It has been the dominant approach implemented in the hardware of Graphics Processing Units (GPUs), and has been accessible through APIs like OpenGL, Vulkan, and DirectX, among others. Broadly speaking, the rasterization algorithm has been deployed in applications where performance and interactivity are required.

Rasterization naturally lends itself to parallelization, but challenges occur when two cores generate fragments for the same pixel at the same time. The correct solution is to have the final image contain the fragment that is closest to the camera, but, for a naive implementation, the process of comparing depths and selecting color can result in incorrectly using a fragment that is further away, due to decisions made based on old information. Traditional GPU architectures address this through specialized hardware units and atomic operations.

The main assumption of this work is that there is a growing need

DOI: 10.2312/egpgv.20261000

for hardware-agnostic parallel rasterization approaches to support scientific visualization on supercomputers. While supercomputers are regularly made up of very high-end GPUs, in some cases vendors are showing reluctance in providing graphics APIs like OpenGL. This is a reflection of the shifting role of GPUs from graphics-focused hardware to general-purpose high-end compute devices. Further, even when these libraries are provided, additional complications can arise with in situ visualization in the form where the visualization software is linked directly into a simulation code [AKC*22]. By using OpenGL for graphics in an in situ scenario, the simulation code would gain additional dependencies, which is viewed as problematic by many simulation teams. Finally, interop mechanisms for sharing simulation data with OpenGL (e.g., CUDA/OpenGL interop) introduce additional synchronization requirements, driver dependencies, and vendor-specific complications. These challenges motivate the development of rendering approaches that leverage only the computational infrastructure simulation codes already require.

We introduce a new algorithm for rasterization composed of data-parallel primitives (DPPs). Of note, other works, such as that by Laine and Karras [LK11], have also considered software rasterization; our work focuses on DPPs because of the momentum for delivering DPP-based visualization solutions on supercomputers, particularly Viskores (formerly called VTK-m) [MSU*16]. This is aligned with our goal of minimizing dependencies for simulation codes in an in situ setting, as Viskores is already part of their software stack in many instances. In terms of algorithm specifics, our approach computes all fragments and sorts them in memory, avoiding the need for hardware atomics. Thus, our algorithm reduces hardware specificity at a cost of increased memory usage, i.e., storing fragments in this way can be memory intensive when depth complexity is high.

The contribution of this work is as follows:

- A DPP-based description of the rasterization algorithm for opaque surfaces;
- An analysis of how the DPP-based rasterizer performs across different platforms;
- A comparison to a community standard for rasterization; and
- An exploration of how the rasterizer performs across varying workloads and available parallelism.

The source code is openly available (see Appendix B).

2. Related Works

2.1. Rasterization

Pineda established the edge-function algorithm for rasterization which serves as the foundation of modern parallel rasterization [Pin88]. Pineda's method offers inherent support for parallel computing and attribute interpolation, which makes it highly appealing. The edge-function is linear, allowing Pineda's algorithm to be conveniently computed in parallel. Moreover, the value of the edge-function is proportional to the distance of a point to an edge, which allows for the computation of the barycentric coordinates used in attribute interpolation. These twin benefits explain the method's enduring dominance as a rasterization algorithm. Our approach integrates the edge-function algorithm with the DPP paradigm.

The Larrabee architecture represented a major landmark in the exploration of software rasterizers. Whereas most contemporary GPUs performed rasterization and other rendering operations with fixed-function logic, Larrabee performed these operations entirely in software [SCS*08]. Indeed, the architecture eschewed the traditional hardware graphics pipeline in favor of a fully programmable software graphics rendering pipeline. While the Larrabee software rasterizer did not achieve the performance of dedicated hardware rasterizers, it was efficient enough to be viable and offered the additional benefit programmability [Abr09]. Thus, the Larrabee architecture and rasterizer demonstrated that flexible and efficient software graphics pipelines are possible to build.

Modern approaches to GPU rasterization have focused on achieving competitive performance with fixed-function hardware while maintaining software flexibility. Laine and Karras constructed a software graphics pipeline optimized for rendering on the NVIDIA Fermi architecture, specifically the GeForce GTX 480 model [LK11]. With these optimizations, their software rasterizer was able to achieve rendering times only 1.5-8x slower than hardware rasterizers. Such results showed that software rasterizers can achieve competitive performance when optimized for the target architecture. However, their rasterizer is hardware-specific, as they make use of hardware atomics and encode the structure of the GTX 480 into their algorithm. We design a hardware-agnostic rasterizer which avoids the usage of hardware atomics and encodes no hardware-specific structure in its algorithm. Thus, this work extends the work of Laine and Karras by presenting a hardware-agnostic software rasterizer. That said, we do not select this work as a comparator. Our goal is providing an alternative to a graphics hardware API, making one the natural choice for comparison, as we describe below.

Researchers have also explored generalized programming models for graphics pipelines beyond rasterization. GRAMPS defined a "programming model for expressing rendering pipelines and other parallel applications," demonstrating that many graphics algorithms, such as rasterization and ray-tracing, can be expressed as graphs of stages connected through queues and buffers [SFB*09]. Both GRAMPS stages and DPPs seek to create performant and flexible algorithms by decomposing those algorithms into parallel building blocks. Since DPPs are finer-grained, our work pushes this decomposition to a more granular level, enabling rasterization to be built with reusable primitives and improving its portability.

OpenGL is a graphics hardware API allowing programmers to specify the production of graphical images, specifically color images of 3D objects [SA22]. The abstract interface OpenGL provides can be implemented on any GPU or CPU, so programs written with OpenGL can run performantly across a wide number of devices and architectures. This work uses OpenGL through the VisIt project. VisIt is a project designed for visualizing and analyzing the extremely large data sets produced by HPC systems [CBW*11], and is heavily used in the supercomputing community. For this work, VisIt serves as a community standard for rasterization against which the DPP-based rasterizer is compared.

2.2. Visualization with Data-Parallel Primitives

As parallelism within a node, whether from multi-core CPUs or from GPUs, were introduced into supercomputers, several efforts were launched exploring portable performance for scientic visualization. The US Department of Energy produced three such efforts, DAX [MAGM11], EAVL [MAPS12], and PISTON [LSA12], which ultimately merged to form the Viskores project [MSU*16]. PISTON, in particular, embraced data-parallel primitives [Ble90] in the style of Thrust [BH12], which carried forward into Viskores. Viskores' DPPs are implemented using "device adapters" over multiple technologies, including Kokkos [TLGA*21], CUDA [Lue08], and others. Viskores expanded the set of DPPs they considered for the patterns they encountered with visualization of mesh data structures [MMP*21]. In this same work, they also studied performance in a meta-study that aggregated over many previously published works, demonstrating that performance was generally comparable with hardware-specific implementations. Of note, Viskores was used to visualize 80 trillion cells on 74,000 GPU [MAB*24]. Finally, Viskores is deployed in Ascent [LBCH22], ParaView, and VisIt, as well as being used for the in situ modules within DOE's ALPINE project [A*25], representing a significant user community making use of DPP-based visualization algorithms.

2.3. Position with the HPC Rendering Landscape

Ray tracing has played a significant role in HPC rendering in recent years. Significant vendors efforts like Embree [WWB*14], OSPray [WJA*16], OptiX [PBD*10], provided highly-optimized support for Intel and NVIDIA. Further, a DPP-based version of ray tracing was contributed to EAVL, and later ported to Viskores [LMNC15]. Other projects, such as Visionaray [ZWL17], also considered portability for ray tracing. While these projects are seeing significant success, the premise of this paper is that there is still a role for rasterization in the HPC ecosystem. Indeed, the work by Larsen on performance modeling for rendering on HPC systems shows that rasterization is expected to outperform raytracing for some use cases [LHK*16]. In all, we do not endeavor to re-explore the differences between ray tracing and rasterization; rather our experiments are designed to inform the efficacy of rasterization when using DPPs. Finally, aligned with the motivation for this work, the ANARI project enables reduced reliance on low-level, vendor-specific graphics APIs through a standardized API layer [SGA*22]. Our approach has the potential to ultimately be an engine in a framework like ANARI, provided it demonstrates sufficient capability and performance.

3. Algorithm Overview

This section describes our DPP-based rasterization algorithm. The algorithm takes as input (1) the total number of triangles, (2) the vertex positions for each triangle, and (3) the color of each triangle, and returns a final image as a buffer of RGB values. Our rendering algorithm consists of four main phases:

- **Rasterization:** computes the fragments for each triangle.
- **Sorting:** groups the fragments that overlap.
- **Selecting:** uses groupings to determine which fragments will appear in the final image.
- **Writing:** collects the fragments chosen by the selecting phase and writes them to the final image buffer.

The remainder of this section describes each of the four phases in its own subsection. Its discussion assumes familiarity with data-parallel primitives, and readers lacking this familiarity are encouraged to first review Appendix A. Further, our appendix defines two non-standard helper functions used in our algorithm: expand and indexInSubGroup. Finally, our pseudocode convention has DPPs written as:

```
primitive<functor>((*inputs),*outputs)
```

with a default functor for that primitive if none is specified and all local and output arrays and variables initialized to zero unless otherwise specified.

3.1. Rasterization

Algorithm 3a:

```
1  /* INPUT */
2  integer: N
3  real array: triangles[N][3][3]
4  character array: colors[N][3]
5  /* OUTPUT */
6  integer: fragments
7  integer array: fragment_positions[fragments][2]
8  real array: fragment_depth[fragments]
9  character array: fragment_color[fragments][3]
10 /* LOCAL */
11 integer array: fragment_count[N], write_index[
     N]
12 integer array: fragment_triangle[fragments]
13 integer array: row_count[N], row_offset[N]
14 integer: rows
15 integer array: row_triangle[rows], row_index[
     rows]
16 integer array: column_count[rows],
     column_offset[rows]
17 integer array: fragment_row[fragments],
18 integer array: fragment_column[fragments]

19 transform<fragCount>((triangles),
     fragment_count)
20 fragments = exclusiveScan<>((fragment_count),
     write_index)
21 expand(write_index, fragment_count,
     fragment_triangle, N)
22 transform<rowCount>((triangles), row_count)
23 rows = exclusiveScan<>((row_count), row_offset
     )
24 expand(row_offset, row_count, row_triangle, N)
25 indexInSubGroup(row_triangle, row_offset,
     row_index, rows)
26 transform<colCount>((triangles, row_index),
     column_count)
27 exclusiveScan<>((column_count), column_offset)
28 expand(column_offset, column_count,
     fragment_row, rows)
29 indexInSubGroup(fragment_row, column_offset,
     fragment_column, fragments)
```

```
30 transform<minus>((fragment_row, permute(
    row_offset, fragment_triangle)), fragment_row
    )
31 transform<rasterize>((permute(triangles,
    fragment_triangle), fragment_row,
    fragment_column), fragment_position,
    fragment_depth)
32 gather<>((fragment_triangle, colors),
    fragment_color)
```

This phase of the algorithm rasterizes each input triangle, producing a vector containing the position, depth, and color of each fragment. The algorithm exhibits a classic DPP pattern, namely that each element of the input (each triangle) generates a variable number of outputs (fragments). We address this by splitting the phase into two sub-phases. The first sub-phase (lines 19-30) counts the number of fragments produced by each triangle (line 19) and generates arrays assigning each fragment to a triangle (lines 20-21) and a row-column index within that triangle (lines 22-30). The second sub-phase (lines 31-32) uses these triangle assignments and row-column indices to generate each fragment's data.

3.2. Sorting

Algorithm 3b: Sort

```
1 /* INPUT */
2 integer: fragments
3 integer array: fragment_positions[fragments][2]
4 real array: fragment_depth[fragments]
5 character array: fragment_color[fragments][3]
6 /* OUTPUT */
7 real array: sorted_fragment_depth[fragments]
8 character array: sorted_fragment_color[
    fragments][3]
9 / * LOCAL */
10 integer array: sorted_indices[fragments]

11 sorted_indices = counting(fragments)
12 sortByKey<>((fragment_positions,
    sorted_indices)) //In-place
13 gather<>((sorted_indices, fragment_colors),
    sorted_fragment_colors)
14 gather<>((sorted_indices, fragment_depth),
    sorted_fragment_depth)
```

This phase of the algorithm sorts fragments by their position to find the fragments that are in the same location. The algorithm first sorts an array of fragment indices by position (lines 11-12) and then uses those indices to gather fragment data to the sorted locations (lines 13-14). Once this phase been completed, the fragment position, color, and depth vectors have been grouped by position.

3.3. Selecting

Algorithm 3c: Select

```
1 /* INPUT */
2 integer: fragments
3 integer array: fragment_positions[fragments][2]
4 real array: sorted_fragment_depth[fragments]
5 /* OUTPUT */
6 boolean array: write_fragment[fragments]
7 /* LOCAL */
8 integer: unique_positions
9 integer array: positions[unique_positions]
10 real array: closest_depth[unique_positions]
11 integer array: position_multiplicity[
    unique_positions]
12 integer array: position_start_index[
    unique_positions]
13 integer array: depth_map[fragments]
14 real array: expanded_closest_depth[fragments]

15 unique_positions = len(unique_copy<>(
    fragment_positions))
16 reduceByKey<maximum>((fragment_positions,
    sorted_fragment_depth), positions,
    closest_depth)
17 reduceByKey<>((fragment_positions, constant(1,
     fragments)), discard(fragments),
    position_multiplicity)
18 exclusiveScan<>((position_multiplicity),
    position_start_index)
19 expand(position_start_index,
    position_multiplicity, depth_map,
    unique_positions)
20 gather<>((depth_map, closest_depth),
    expanded_closest_depth)
21 transform<equalTo>((expanded_closest_depth,
    sorted_depth), write_fragment)
```

In this phase the algorithm selects which fragments will be written to the image. It is this phase that allows our algorithm to execute without race conditions on the writing of fragments. Fragments are selected such that only one fragment, the fragment with minimum depth, will be written to any position in the image. This is done by generating an array associating each fragment to the minimum depth at its position (lines 15-20) and an array which assigns each fragment a boolean value which is true if its depth is equal to that minimum depth and false otherwise (line 21). As a result, there is no case where multiple cores will write fragments to the same image position during the writing phase, preventing race conditions. The phase takes the sorted position and depth of each fragment and returns a vector indicating whether each fragment is written to the final image or not. Once this phase been completed, the algorithm has produced a vector of booleans indicating whether or not each fragment will be written in the final image.

3.4. Writing

Algorithm 3d: Write

```
1 /* INPUT */
2 integer: fragments, width, height
3 integer array: fragment_positions[fragments][2]
4 character array: sorted_fragment_color[
    fragments][3]
5 boolean array: write_fragment[fragments]
6 /* OUTPUT */
7 character array: final_image[width * height *
    3]
8 /* LOCAL */
9 integer array: row_major_position[fragments]
```

```
10 character array: image[width * height][3]
11 integer array: channel_to_pixel_map[width *
    height * 3]
12 integer array: channel_to_pixel_color[width *
    height * 3]
13 integer array: channel_type[width * height *
    3]

14 transform<toRowMajor(width)>((
    fragment_position), row_major_position)
15 scatterIf<>((sorted_fragment_colors,
    row_major_position, write_fragment), image)
16 transform<groupThree>((counting(width * height
     * 3)), channel_pixel_map)
17 channel_pixel_color = permute(image,
    channel_pixel_index)
18 // R = 0, G = 1, B = 2
19 transform<indexThree>((counting(width * height
     * 3), channel_type)
20 transform<channelValue>((channel_type,
    channel_pixel_color), final_image)
```

In the final phase of this algorithm, the fragments are written to the final image. The phase takes the sorted position and color of each fragment and the vector of booleans indicating if each fragment is written and returns the final image as a vector of colors. Our algorithm returns the final image as a flat array of characters which can be grouped in threes to get the RGB values of each pixel. It does this by assigning each group of three elements to a pixel and its color (lines 16-17), assigning each of those elements to the red, green, or blue or green channel (line 19), and then storing the appropriate value in them (line 20). Once this phase has been completed, the color of each written fragment has been placed at that fragment's position in the final image. The final image is thus populated by the visible fragments of the initial triangles. The algorithm is then complete and the final image is returned.

4. Experiment Overview

Our study consists of three campaigns of experiments, covering the Portability, Workload Scaling, and Competitiveness of our rasterization program. We run our experiments over the following axes:

- Software: 5 options
- Hardware: 4 options
- Data Sets: 6 options
- Concurrency Level: 5 options
- Image Resolution: 8 options
- Mesh Resolution: 7 options

We run a total of 48 (Portability) + 61 (Workload Scaling) + 6 (Competitiveness) = 115 tests over these axes, the details of which are described in the remainder of this section.

4.1. Software

We wrote two versions of our rasterization program, one using the NVIDIA Thrust library, and the other using the Viskores library. Both libraries provide access to performant implementations of DPPs written for C++, as well as support for a number of different host and device platforms for execution. Their backends are as follows:

Implementation	CPU	NVIDIA GPU	AMD GPU
Thrust	OpenMP	CUDA	—
Viskores	TBB	CUDA	Kokkos

We also compared to a community standard, VisIt, using OpenGL for rendering on a GPU.

4.2. Hardware

We ran our experiments on three different supercomputers: Talapas, Perlmutter, and Frontier. We ran both CPU and GPU tests on Talapas. Perlmutter and Frontier were used only for GPU tests. Finally, we test our community standard VisIt on a desktop computer. We summarize our hardware architectures below:

Device	Machine	Hardware
INTEL	Talapas	Intel Xeon E5-2690 v4
NVIDIA 1	Talapas	NVIDIA A100 PCIe
NVIDIA 2	Perlmutter	NVIDIA A100 SXM
NVIDIA 3	Desktop	NVIDIA Geforce GTX 1070 Ti
AMD	Frontier	AMD Instinct MI250X Accelerator

4.3. Data Sets and Images

We use six STL models sourced from thingiverse.com to test the performance of our program. Figure 2 has renderings of these data sets, and their triangle count, default image resolution, and the number of fragments generated at this resolution is listed below. All aspects of this table change in the Workload Scaling portion of the study. With Mesh Resolution, triangles are subdivided into four repeatedly. For example, a Mesh Resolution option of two subdivisions on Bunny means each triangle is split into four (first subdivision) and each of the resulting four is split into another four (second subdivision), for a total of $112,402 \times 16 = 1,798,432$ triangles. For Image Resolution, each subdivision replaces a single pixel with a 2x2 array of pixels, meaning an Image Resolution option of three subdivisions on Bunny would be 864 x 704 pixels (width = 108×8, height = 88×8). For fragment count, the number of fragments is approximately proportional to the increase in pixels as the image becomes higher resolution.

Data Set	Triangles	Resolution	Fragments
Bunny	112,402	108 x 88	14,207
Jeff the Landshark	3,750,442	77 x 104	30,690
Holding Cat	533,180	164 x 150	26,531
Dragon Sturdier	311,572	22 x 23	1,175
Darwin Bust	2,132,934	78 x 122	22,795
Bag Holder	12,138	134 x 124	13,378

4.4. Timing

To measure the run time of our rasterization function, we time the execution of each of the four phases of our algorithm, using CUDA Events for Thrust and the native Viskores Timer for Viskores. For

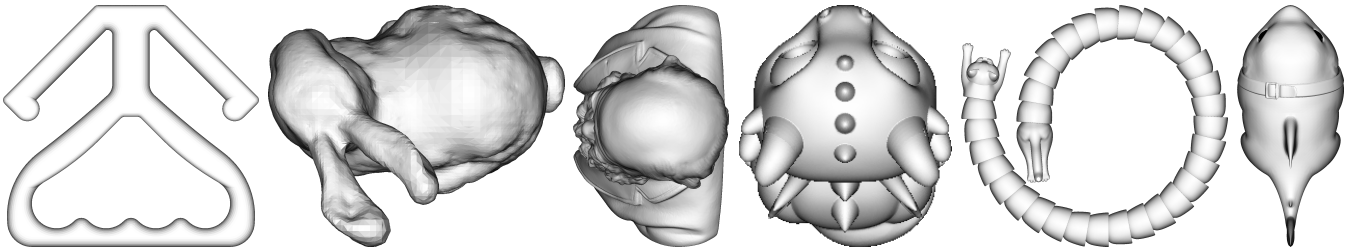

Figure 2: *The data sets rendered in this study. From left to right: Bag Holder, Bunny, Darwin Bust, Dragon Sturdier, Holding Cat, and Jeff the Landshark.*

VisIt, we use the built-in timing instrumentation, which simply reports a per-frame render time.

4.5. Campaign Details

Our study consists of three campaigns: Portability, Scalability & Workload Size, and Competitiveness. Our Portability tests evaluate the performance of our software across different hardware architectures. Our Workload Scaling tests evaluate as image and model sizes change, as well as available parallelism. Finally, our Competitiveness tests compare our algorithm to community standards for rasterization. For all tests, data sets were rendered from the viewpoint shown in Figure 2.

4.5.1. Portability

These tests varied three factors:

- Software: 5 options
- Hardware architecture (CPU and GPU): 4 options
- Data set: 6 options

We measured the total runtime of our algorithm on the cross product of 8 hardware-software combinations and 6 data sets for a total of 8 x 6 = 48 tests. The hardware and software combinations selected for these tests were Thrust using OpenMP and Viskores using TBB on INTEL, Thrust and Viskores using CUDA on NVIDIA 1, Thrust using CUDA and Viskores using CUDA and Kokkos on NVIDIA 2, and Viskores using Kokkos on AMD. Each model was rendered at 10x its original image resolution. For example, Bunny was rendered at 1080x880. CPU tests were executed using 14 threads.

4.5.2. Workload Scaling

These tests varied in three factors:

- Concurrency Level: 5 options (1, 2, 4, 8, 14 threads)
- Image Resolution: 8 options (1x, 2x, 4x, 8x, 16x, 32x, 64x, 128x scale)
- Mesh Resolution: 7 options (0, 1, 2, 3, 4, 5, 6 subdivisions)

We measured the runtime of the four phases of our algorithm on each concurrency level and on the cross product of each scale and subdivision level for a total of 5 + 8 x 7 = 61 tests. Concurrency tests were performed on INTEL and scale and subdivision tests were conducted on NVIDIA 2. The Jeff the Landshark data set is used for concurrency tests, and the Bunny data set is used for scale and subdivision tests.

4.5.3. Competitiveness

These tests vary one factor:

- Data Set: 6 options

We measure the frames per second (FPS) of VisIt on each data for for a total of 6 tests and compare the results with those of the Portability tests (48 tests). The architecture options used for the community standard is NVIDIA 3.

5. Results

5.1. Portability

Table 1 shows the frames per second (FPS) achieved for each data set by both the Thrust and Viskores rasterizers on the INTEL, NVIDIA 1, and AMD architectures. The remainder of this section compares these runtimes, first between GPU architectures, and then between GPU and CPU architectures.

The results of Table 1 demonstrate that our program can be ported across different platforms. The Viskores implementation of our program is able to run on INTEL, NVIDIA 1, NVIDIA 2, and AMD without modification to its source code. The Thrust implementation of our program cannot run on AMD, but can run on INTEL, NVIDIA 1, and NVIDIA 2 without modification to its source code as well. Moreover, the Viskores and Thrust implementations of our program have comparable performance across all platforms that they share.

However, the difference in runtimes between GPU architectures is non-trivial. Across all software options, the NVIDIA GPUs (NVIDIA 1 and NVIDIA 2) are 2.4x faster than the AMD GPU (AMD). However, this disparity can be accounted for by a difference in implementation. For the tests where Viskores is used with the Kokkos backend, the NVIDIA GPU (NVIDIA 2) performs only 1.4x faster than the AMD GPU (AMD). Indeed, on NVIDIA 2, the Thrust and Viskores rasterizers using the CUDA backend perform 3.1x faster than the Viskores rasterizer using the Kokkos backend. Thus, when the disparity in performance between the Kokkos and CUDA backends are accounted for, the NVIDIA and AMD GPUs perform similarly, showing that the rasterizer can be ported to different GPUs while maintaining its performance.

However, Table 1 also shows that NVIDIA 2 performs significantly better than NVIDIA 1. For the Thrust and Viskores rasterizers using the CUDA backend, NVIDIA 2 performs 2x faster than

Table 1: *This table shows the frames per second (FPS) the Thrust and Viskores rasterizers achieve while rendering a workload. Rows correspond to data sets, and columns correspond to hardware-software configurations. The results for the community standard VisIt on each data set are presented in the final column for comparison.*

Algorithm Backend Device	Thrust OpenMP INTEL	Viskores TBB INTEL	Thrust CUDA NVIDIA 1	Viskores CUDA NVIDIA 1	Thrust CUDA NVIDIA 2	Viskores CUDA NVIDIA 2	Viskores Kokkos NVIDIA 2	Viskores Kokkos AMD	VisIt OpenGL NVIDIA 3
Bag Holder	8	11	90	85	156	135	57	38	200
Bunny	7	10	92	88	173	153	57	38	200
Darwin	5	6	51	50	114	126	42	27	26
Dragon	57	56	352	337	449	376	98	88	250
Cat	4	6	55	52	122	105	38	25	200
Landshark	3	4	40	40	103	112	33	23	26

Table 2: *This table shows the speedup achieved through a GPU over CPU execution. The GPU and CPU runtimes are the average runtimes across all workloads in an experiment. Rows correspond to different algorithms. Rasterization is the proposed algorithm whereas Raytracing and Volume Rendering are reported by their respective publications. Note that the workloads varied across these studies, and so this does not establish which algorithm or implementation is faster. Rather, it demonstrates that the difference between CPU and GPU performance for three rendering algorithms is similar.*

Algorithm	GPU (s)	CPU (s)	Speedup
Rasterization	0.016	0.158	9.9
Raytracing [LMNC15]	0.042	0.460	10.9
Volume Rendering [LLN*15]	0.059	0.974	16.5

NVIDIA 1. This is unexpected because NVIDIA 1 and NVIDIA 2 both utilize the NVIDIA A100 GPU, and we would not expect there to be a performance difference when using the same architecture. However, it is also the case that GPU execution is often memory-bound rather than compute-bound. Since the DPP-based rasterization algorithm stores all fragments and triangles in memory, it is very memory intensive and more likely to be a memory-bound process. Thus, this disparity can be accounted for as a difference in the configuration of memory on Talapas and Perlmutter, and this conclusion is supported by the fact that the disparity is greatest on the Darwin Bust and Jeff the Landshark data sets, which are the largest data sets we test in terms of triangles.

Table 1 shows that INTEL is 9.9x slower than NVIDIA 1, 15x slower than NVIDIA 2, and 4.6x slower than AMD. Such results are in line with previous work comparing DPPs on GPUs and CPUs. Table 2 compares our GPU and CPU performance to the performance of the DPP-approaches to other proposed portable rendering algorithms: Raytracing [LMNC15] and Volume Rendering [LLN*15]. For the DPP-based rasterization algorithm, GPU execution achieves a 9.9x speedup over execution on a CPU. This speedup is similar to other studies in parallel rendering that Table 2 presents. Thus, the performance of the DPP-based rasterizer between GPU and CPU architectures is within the expected range for DPP-based software, which indicates that the rasterizer can be ported from GPUs to CPUs while remaining performant.

We conclude that the DPP approach for rasterization is portable. The algorithm we present runs on multiple architectures without modification, and it maintains its performance when transferred between different GPU and CPU architectures. Thus, DPPs are able to achieve the promise of portable performance for the rasterization algorithm.

5.2. Competitiveness

Table 1 shows that VisIt achieves substantially better performance on the tested data sets than the DPP-based rasterization program. The median framerate for VisIt is 200 FPS while the median framerate for the DPP-based rasterization program is 88 FPS on GPU architectures and 6.5 FPS on CPU architectures. Looking at the performance of the DPP rasterizer using the CUDA backend on the NVIDIA architectures alone, the median rises to 108.5 FPS, just over half the median framerate for VisIt.

There are a few cases where the DPP rasterizer outperforms VisIt. The median framerate of the DPP rasterizer on GPU architectures of the Darwin Bust and Jeff the Landshark data sets are 50.5 FPS and 40 respectively. However, VisIt achieves only 26 FPS on these data sets. Similarly, the median framerate of DPP rasterizer on GPU architectures on the Dragon Sturdier data set is 344.5 FPS while VisIt achieves 250 FPS. Then for these three data sets, the DPP rasterizer outperforms VisIt. These data sets represent the extreme cases in the testing of this study. Darwin Bust and Jeff the Landshark are very complex models, consisting of 2,132,934 triangles and 3,750,442 triangles respectively, and Dragon Sturdier is a very small image, being 22 x 23 pixels at base and 220 x 230 pixels at 10x scale. These results indicate that the DPP rasterizer may compete better with VisIt for workloads closer to the boundary of visualization use cases.

We conclude that the DPP rasterizer is slower than community standards for rasterization, but still fast enough to be useful. In particular, frame rates of above twenty are useful for most in situ scenarios. Of course, only limited conclusions should be drawn from the experiments, as they were run on different hardware, with VisIt being run on a desktop rather than a supercomputer. That said, there is no reason to think our algorithm would outperform a traditional

rasterizer; the contribution of our approach is on portability and our performance goal is simply to achieve similar frame rates.

5.3. Workload Scaling

This section considers the performance of our approach across varied workloads and available parallelism. In the context of CPUs, available parallelism means CPU cores, which we explicitly vary to do traditional scalability tests. In the context of GPUs, available parallelism refers to how well the workload engages the hardware, i.e., small workloads may not fully exploit GPU resources, while larger ones can. In terms of findings, the results of our tests indicate that our program performs well with different image sizes, triangle counts, and concurrency levels.

5.3.1. Concurrency (Scalability On CPU Cores)

Table 3 shows the time and cost of the rasterizer while running with different thread counts, with cost calculated as time multiplied by the number of threads used for execution. It also shows the speedup over the serial execution that the program achieves at each thread count. The remainder of this section discusses how it scales as it runs with greater concurrency.

Table 3: *This table shows the runtime and parallel cost of the program on the Jeff the Landshark data set as well as its speedup over serial execution while using different numbers of threads.*

Cores	Time (s)	Cost (core·s)	Speedup
1	0.143	0.143	1.00
2	0.135	0.27	1.059
4	0.081	0.324	1.765
8	0.05	0.4	2.86
14	0.036	0.507	3.972

The results of Table 3 show a clear speedup due to parallelism. However, this speedup is sublinear, with the speedup over serial execution falling behind as the number of threads increases, indicating inefficiences. The rasterizer with 14 threads is 4.0x faster than it is during serial execution, 3.5X below perfect scaling. Comparatively, the scalability test of the DPP-based volume renderer presented by Larsen, et al. is only 1.4x slower than linear scaling at 24 threads [LLN*15]. Similarly, the two DPP-based external facelist calculation algorithms presented by Lessley, et al. is 1.5x slower than linear scaling at 16 threads [LBMC16].

Figure 3 shows that, of the four phases, the select phase scales the worst as threads increase. However, all phases demonstrate sublinear speedups, which leads to the sublinear acceleration of the overall program. Thus, the rasterizer scales worse than other DPP-based algorithms, specifically those for external facelist calculation and volume rendering. However, we conclude that the DPP-rasterizer scales well enough as parallel resources grow, even if this scaling is not exceptional.

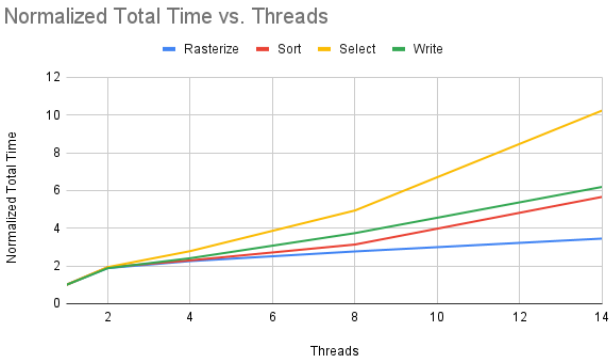

Figure 3: *This figure shows the normalized cost for each phase of the algorithm up to 14 threads. The normalized cost is computed as the cost for each thread count divided by the cost for serial execution.*

Table 4: *This table shows the expected FPS of our program on the Bunny data set as scale and subdivisions vary. Scales vary along the rows of this table, and subdivisions vary along its columns.*

	Triangle Subdivisions						
Image	0	1	2	3	4	5	6
1x	818	663	54	359	202	78	19
2x	693	553	416	307	185	75	22
4x	353	303	271	222	145	66	16
8x	209	196	150	131	98	52	19
16x	118	109	97	88	69	41	16
32x	47	46	20	42	34	10	6
64x	13	12	14	7	8	11	4
128x	3	4	4	3	3	2	—

5.3.2. Scaling Mesh and Image Resolution

Table 4 shows the FPS of our program as mesh resolution and image resolution varies. The results demonstrates that the rasterizer achieves interactive rates for many of the tested configurations. The two most challenging workloads to achieve 24 FPS or better are 32x image resolution and 4 triangle subdivisions and 16x image resolution and 5 triangle subdivisions. For the Bunny data set (112,402 triangles, 108 x 88 pixels) these configurations translate to a model of 28,774,912 triangles spanning an image of 3,456 x 2,816 pixels or a model of 115,099,648 triangles spanning an image of 1,728 x 1,408 pixels. The image area of the former is approximately equivalent to a 4k resolution image, and the latter a 1080p resolution image. The most challenging workload achieving 12 FPS or better is the combination of 16x image resolution and 6 triangle subdivisions, which yields a model of 460,398,592 triangles rendered to an image of 1,728 x 1,408 pixels. Further, Table 4 shows that all successful tests achieve a framerate higher than 2 FPS. While 2 FPS is poor performance when animation is required, this indicates that the program is capable of rendering models with very large triangle counts to static images with very high resolutions in a reasonable amount of time. In all, we conclude Table 4 demonstrates that the

DPP-based rasterizer scales to the model complexities and image sizes required for visualization software.

That said, our approach's property of explicitly storing fragments in an intermediate array caused the 128x image resolution and 6 triangle subdivision test to fail due to running out of memory. The overhead for each fragment is 51 bytes, with 15 of those bytes being for the fragment itself (four each for X, Y, and depth, and three for color), and the remaining 36 bytes in other arrays in support of the calculation. Assuming the number of fragments scales with image size, the 128X image for Bunny would have 232M fragments ($= 14,207 \times 128 \times 128$) and thus take almost 12GB. This was smaller than the memory footprint for the 6 triangle subdivision Bunny model (46M triangles and 25GB), but enough to cause our test to fail.

6. Conclusion

This work contributes a novel algorithm for rasterization using DPPs and evaluated its performance through a series of experiments. While the algorithm was slower than existing community standards for rasterization, it still provided generally strong performance, and performance that is suitable in the context of in situ visualization. Moreover, the DPP rasterizer achieves the promise of portable performance which gives it longevity and usability across many architectures and devices. In all, these experiments demonstrate that the DPP rasterizer is viable within the high-performance visualization space.

In terms of future work, alternative methods of selecting fragments to be written should be explored. The approach presented in this work keeps all fragments in memory from its rasterization step to its selecting step. A scene with a high depth complexity could stress available memory, causing the algorithm to fail. One alternative is to use hashing instead of sorting for fragment selection. Efficient implementations of hash tables have been developed with DPPs [LLC20], and Lessley et al. find that for external facelist calculation a hashing-based DPP algorithm outperforms a sorting-based DPP algorithm [LBMC16]. Using hashing may improve the competitiveness of the algorithm.

Another avenue of future work is support for transparent surfaces. While our approach of keeping all fragments is a liability with respect to memory footprint, it is an asset for transparent rendering, as we can sort the fragments and blend them.

Another area of future work includes an investigation of hierarchical visibility via DPPs. Hierarchical approaches to visibility and rasterization enable the management of high complexity, and especially high depth-complexity, scenes, a context in which the approach presented by this work struggles. Greene et al. introduce hierarchical Z-buffering, an algorithm combining an object-space octree and an image-space Z pyramid for efficiently culling hidden geometry [GKM93]. Adapting such an algorithm to the DPP paradigm and combining it with the DPP-based rasterization algorithm will ameliorate the weakness of our algorithm to high depth-complexities, enabling the rendering of more complex scenes and models.

Finally, to become a useful graphics engine, our code base needs additional rendering capabilities: colors, textures, and camera transformations. While these were not necessary to evaluate our research questions, they will be necessary for our approach to be a practical option within Viskores for rasterization-appropriate workloads. Further, while verifying the fitness of our algorithm in a distributed-memory rendering setting should be straightforward, this should be confirmed because a major premise of this work is in support of visualization on supercomputers.

7. Acknowledgments

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy User Facility (project m2956-2025). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a Department of Energy User Facility using NERSC award ASCR-ERCAP0035972. The authors acknowledge Research Advanced Computing Services (RACS) at the University of Oregon for providing computing resources that have contributed to the research results reported within this publication. This work was also supported in part by the U.S. Department of Energy (DOE) RAPIDS SciDAC project under contract number DE-AC05-00OR22725. The authors used Anthropic's Claude to provide feedback and suggestions on text written by the authors, and also to recommend related works for the authors to review.

Appendix A: DPP Preliminaries

Our algorithm uses the following DPPs:

- **`gather`**`<>((input, map), output)`: Copies elements from an input range to an output range according to a map. The map determines what elements of the input each element of the output copies from. `gather` is the inverse of `scatter`.
- **`reduceByKey`**`<functor>((keys, values), keys_output, values_output)`: Applies the binary operator defined by its functor to each element of its input that shares a key to reduce that input to a single value for each key.
- **`scan`**`<functor>((input), output)`: Applies the binary operator defined by its functor to each element of its input and stores the partial sum of the input range to each element of the output. `scan` may be `inclusive` (the element at an index is included in the partial sum for that index) or `exclusive` (the element at an index is not included in the partial sum for that index).
- **`scatterIf`**`<>((input, map, stencil), output)`: Conditionally copies elements of an input range into an output according to a map and predicated by a stencil. The map determines where elements of the input are copied to in the output. The stencil determines if each copy is executed, and its elements are treated as booleans. `scatter` is the inverse of `gather`.
- **`sortByKey`**`<comparator>((keys, values))`: Sorts

each element of the input by its key in-place using the comparator.
- **transform**<functor>((*inputs), output): Applies the operation defined by its functor to each element of its input and stores the result in its output. The number of inputs is the number of inputs required by the functor.

Our algorithm also includes some standard helper functions which are provided by the DPP library:

- **counting**(N) returns the sequence from 0 to N-1 as an integer array.
- **permute**(source,map) permutes the source array according to the map array and returns the result.
- **constant**(v,N) returns an array of length N containing only the value v.
- **discard**(N) returns a write-only array of length N which is discarded after use.
- **len**(array) returns the length of an array.

Finally, our algorithm utilizes two new helper functions:

- **expand**(map,count,output,N) (Algorithm 1) scatters values from 0 to N-1 into the output according to the map and then fills the values in between. Put another way, the function duplicates each value in the sequence 0 to N-1 by its respective count and returns this result as the output array. The count array stores these counts, but it is only used to determine if a certain value has a count of 0.
- **indexInSubGroup**(map,offset,output,N) function (Algorithm 2) takes the sequence from 0 to N-1 and subtracts from each value an offset given by the permutation of the offset array by the map array. This function is used to index the elements within each subgroup of a larger supergroup. The map array defines what subgroup each element of the supergroup belongs to and the offset array defines the offset of each subgroup within the supergroup.

Algorithm 1: the expand helper function

```
1 /* INPUT */
2 integer: N; integer arrays: map[N], count[N]
3 /* OUTPUT */
4 integer array: output[N]
5 /* LOCAL */
6 integer array: countingArray[N]

7 countingArray = counting(N) // 0, 1, ... , N
8 scatterIf<>((countingArray, map, count), output
    )
9 inclusiveScan<Maximum>((output), output)
```

Algorithm 2: The indexInSubGroup helper function

```
1 /* INPUT */
2 integer: N; integer arrays: map[N], offset[]
3 /* OUTPUT */
4 integer array: output[N]

5 transform<Minus>((counting(N), permute(offset,
    map)), output)
```

Appendix B: Open Source Code

The source code presented and evaluated in this paper is available at https://www.kennethmoreland.com/rasterization-dpp/.

References

[A*25] AHRENS J., ET AL.: The ECP ALPINE project: In situ and post hoc visualization infrastructure and analysis capabilities for exascale. *The International Journal of High Performance Computing Applications 39*, 1 (Jan. 2025), 32–51. 3

[Abr09] ABRASH M.: Rasterization on Larrabee. *Dr. Dobb's Journal* (May 2009). 2

[AKC*22] ATZORI M., KÖPP W., CHIEN S. W. D., MASSARO D., MALLOR F., PEPLINSKI A., REZAEI M., JANSSON N., MARKIDIS S., VINUESA R., LAURE E., SCHLATTER P., WEINKAUF T.: In situ visualization of large-scale turbulence simulations in nek5000 with paraview catalyst. *J. Supercomput. 78*, 3 (Feb. 2022), 3605–3620. 2

[BH12] BELL N., HOBEROCK J.: Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371. 3

[Ble90] BLELLOCH G. E.: *Vector Models for Data-Parallel Computing*. MIT Press, 1990. ISBN 0-262-02313-X. 3

[CBW*11] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., BONNELL K., MILLER M., WEBER G. H., HARRISON C., PUGMIRE D., FOGAL T., GARTH C., SANDERSON A., BETHEL E. W., DURANT M., CAMP D., FAVRE J. M., RÜBEL O., NAVRÁTIL P., WHEELER M., SELBY P., VIVODTZEV F.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *Proceedings of SciDAC 2011* (Denver, CO, July 2011). 2

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, Sept. 1993), SIGGRAPH '93, Association for Computing Machinery, pp. 231–238. 9

[LBCH22] LARSEN M., BRUGGER E., CHILDS H., HARRISON C.: Ascent: A Flyweight In Situ Library for Exascale Simulations. In *In Situ Visualization For Computational Science*. Mathematics and Visualization book series from Springer Publishing, Cham, Switzerland, May 2022, pp. 255 – 279. 3

[LBMC16] LESSLEY B., BINYAHIB R., MAYNARD R., CHILDS H.: External Facelist Calculation with Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (Groningen, The Netherlands, June 2016), pp. 10–20. 8, 9

[LHK*16] LARSEN M., HARRISON C., KRESS J., PUGMIRE D., MEREDITH J. S., CHILDS H.: Performance Modeling of In Situ Rendering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)* (Salt Lake City, UT, Nov. 2016), pp. 24:1–24:12. 3

[LK11] LAINE S., KARRAS T.: High-performance software rasterization on GPUs. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics* (New York, NY, USA, 2011), HPG '11, Association for Computing Machinery, pp. 79–88. event-place: Vancouver, British Columbia, Canada. 2

[LLC20] LESSLEY B., LI S., CHILDS H.: HashFight: A Platform-Portable Hash Table for Multi-Core and Many-Core Architectures. In *IS&T International Symposium on Electronic Imaging: Visualization and Data Analysis (VDA)* (Burlingame, CA, Jan. 2020), pp. 376–1–376–12. 9

[LLN*15] LARSEN M., LABASAN S., NAVRÁTIL P., MEREDITH J., CHILDS H.: Volume Rendering Via Data-Parallel Primitives. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)* (Cagliari, Italy, May 2015), pp. 53–62. 7, 8

[LMNC15] LARSEN M., MEREDITH J., NAVRÁTIL P., CHILDS H.: Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium* (Hangzhou, China, Apr. 2015), pp. 279–286. 3, 7

[LSA12] LO L., SEWELL C., AHRENS J.: PISTON: A portable cross-platform framework for data-parallel visualization operators. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012). 3

[Lue08] LUEBKE D.: Cuda: Scalable parallel programming for high-performance scientific computing. In *2008 5th IEEE international symposium on biomedical imaging: from nano to macro* (2008), IEEE, pp. 836–838. 3

[MAB*24] MORELAND K., ATHAWALE T. M., BOLEA V., BOLSTAD M., BRUGGER E., CHILDS H., HUEBL A., LO L.-T., GEVECI B., MARSAGLIA N., PHILIP S., PUGMIRE D., RIZZI S., WANG Z., YENPURE A.: Visualization at the exascale: Making it all work with vtk-m. *International Journal of High Performance Computing Applications (IJHPCA) 38*, 5 (Oct. 2024), 508—526. 3

[MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.-L.: Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)* (2011). 3

[MAPS12] MEREDITH J. S., AHERN S., PUGMIRE D., SISNEROS R.: EAVL: The Extreme-scale Analysis and Visualization Library. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), Childs H., Kuhlen T., Marton F., (Eds.), The Eurographics Association. 3

[MMP*21] MORELAND K., MAYNARD R., PUGMIRE D., YENPURE A., VACANTI A., LARSEN M., CHILDS H.: Minimizing Development Costs for Efficient Many-Core Visualization Using MCD^3. *Parallel Computing 108* (Dec. 2021), 102834. 3

[MSU*16] MORELAND K., SEWELL C., USHER W., LO L.-T., MEREDITH J., PUGMIRE D., KRESS J., SCHROOTS H., MA K.-L., CHILDS H., LARSEN M., CHEN C.-M., MAYNARD R., GEVECI B.: VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications 36*, 03 (May 2016), 48–58. Place: Los Alamitos, CA, USA Publisher: IEEE Computer Society. 2, 3

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., ET AL.: Optix: a general purpose ray tracing engine. *Acm transactions on graphics (tog) 29*, 4 (2010), 1–13. 3

[Pin88] PINEDA J.: A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph. 22*, 4 (June 1988), 17–20. 2

[SA22] SEGAL M., AKELEY K.: OpenGL 4.6 (Core Profile) - May 5, 2022. 2

[SCS*08] SEILER L., CARMEAN D., SPRANGLE E., FORSYTH T., ABRASH M., DUBEY P., JUNKINS S., LAKE A., SUGERMAN J., CAVIN R., ESPASA R., GROCHOWSKI E., JUAN T., HANRAHAN P.: Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 1–15. 2

[SFB*09] SUGERMAN J., FATAHALIAN K., BOULOS S., AKELEY K., HANRAHAN P.: Gramps: A programming model for graphics pipelines. *ACM Trans. Graph. 28*, 1 (Feb. 2009). 2

[SGA*22] STONE J. E., GRIFFIN K. S., AMSTUTZ J., DEMARLE D. E., SHERMAN W. R., GÜNTHER J.: Anari: A 3-d rendering api standard. *Computing in Science & Engineering 24*, 2 (2022), 7–18. 3

[TLGA*21] TROTT C. R., LEBRUN-GRANDIÉ D., ARNDT D., CIESKO J., DANG V., ELLINGWOOD N., GAYATRI R., HARVEY E., HOLLMAN D. S., IBANEZ D., ET AL.: Kokkos 3: Programming model extensions for the exascale era. *IEEE Transactions on Parallel and Distributed Systems 33*, 4 (2021), 805–817. 3

[WJA*16] WALD I., JOHNSON G. P., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRÁTIL P.: Ospray-a cpu ray tracing framework for scientific visualization. *IEEE transactions on visualization and computer graphics 23*, 1 (2016), 931–940. 3

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 1–8. 3

[ZWL17] ZELLMANN S., WICKEROTH D., LANG U.: Visionaray: A cross-platform ray tracing template library. In *2017 IEEE 10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)* (2017), pp. 1–8. 3