Volume Rendering Via Data-Parallel Primitives

M. Larsen, S. Labasan, P. Navrátil, J.S. Meredith, and H. Childs

31 Oct,2023

1 Presented By

Dasari Charithambika
210302
Statistics and Data Science
Indian Institute of Technology Kanpur
cdasari21@iitk.ac.in

Divya Gupta
210353
Statistics and Data Science
Indian Institute of Technology Kanpur
divyagupta21@iitk.ac.in

2 Problem Statement

This paper presents a challenging problem due to the diverse array of hardware architectures available for supercomputing. The issue is particularly pronounced in the context of visualization software for two main reasons:

- **Large Code Bases**: Visualization software typically involves extensive code bases, often exceeding a million lines of code, and adheres to various community standards.
- **Numerous Algorithms**: Visualization software employs a wide range of algorithms. Optimizing the software's performance for a specific platform necessitates optimizing each of its algorithms, unlike simulation codes where optimization often focuses on a single "key loop."

3 Motivation

- **Shift in Supercomputing**: Supercomputing is shifting towards energy-efficient architectures with many cores. This necessitates algorithms that can perform well across a range of hardware platforms.
- **Challenges in Visualization Software**: Visualization software is complex, and optimizing it for diverse architectures is a significant challenge.
- **Domain-Specific Languages (DSLs)**: Domain-specific languages are gaining importance as a way to create adaptable algorithms that work well on different platforms.
- **Contributing a New Algorithm:** The paper introduces a novel volume rendering algorithm composed of data-parallel primitives, addressing the need for adaptable and efficient visualization algorithms.

4 Methodology

4.1 Data-parallel primitive

- In the data-parallel paradigm, essential primitives like map, reduce, gather, and scatter serve as building blocks for constructing algorithms.
- Dax, EAVL, and PISTON have adopted a programming model based on functors and operators. Functors are user-defined structures that specify a function and local data, which data-parallel primitive operators apply to input arrays.
- This approach allows these libraries to abstract away low-level concerns such as memory allocation and thread management, enabling users to focus on designing algorithms using the provided data-parallel primitives.
- Users are empowered to reimagine and implement algorithms by leveraging these primitives, simplifying the development process for high-performance computing applications.

4.1.1 Map

- Computational work-horse of data-parallel primitives.
- It is the body of a for loop, where each iteration of the loop can be executed independently.
- Every iteration is randomly carried out in parallel, with no dependence on earlier executions.

4.1.2 Gather and Scatter

- Copies items in parallel from the input arrays to the output arrays, where the input arrays are of length n and the output arrays are of length m.
- In Gather, the user specifies a set of indices x of length m, where out[i] =input[x_i] for each i in m.
- In Scatter, the user specifies a set of indices x of length n, where out[x_i] = input[i] for each i in n.

4.1.3 ReverseIndex

- A specialized Scatter operation that can generate the set of indices to be used in subsequent Gather operations when passed a Boolean array of flags specifying whether each input element is included in the set.

4.1.4 Reduce

- It is the operations that combine the input values of an array to a single output value.
- Ex-1:- Summing all the values in an array.
- Ex-2:- Finding the maximum value in an array.

4.1.5 Scan

- Like Reduce, it has loop-carried dependencies, but the output is an array instead of a single value, and the result at each point in the output array is the partial reduction up to that point.
- Ex-1:- Prefix sum[i] - the sum of elements in an array up to i_{th} position

4.2 Algorithm

- The algorithm is a sampling-based method of rendering images with the goal of packing W $\times$ H $\times$ S samples into a big buffer, where W and H stand for the width and height of the image and S for the sample depth.
- The algorithm splits the sampling process into passes, each of which focuses on a distinct depth-based portion of the buffer, in order to effectively manage memory.
- By recognizing opaque pixels after each run, this method enables early ray termination, which lowers memory needs and computational costs.
- Each pass consists of four phases:
  1. Pass Selection
  2. Screen Space Transformation
  3. Sampling
  4. Compositing
- The algorithm also depends on an Initialization step.

4.2.1 Initialization Step

- Calculate the minimum and maximum depth of each tetrahedron using camera transform and storing the smallest and largest depths in respective arrays.
- This is accomplished with a Map primitive.

4.2.2 Pass Selection

- Identify which cells can possibly contribute to a sample.
- Constructing a Boolean array of size N for N cells, **True** if the corresponding cell can possibly contribute, and **False** otherwise.
- The second step of this phase is to create an array of tetrahedrons that can contribute samples during this pass.
- This is accomplished by successive use of four data-parallel primitives: **Reduce, Exclusive Scan, Reverse Index,** and **Gather**.
- First, a **Reduce** primitive counts the number of active tetrahedrons, i.e., tetrahedrons that have "true" in the Boolean array.
- Second, an **Exclusive Scan** primitive calculates the index that each of the active tetrahedrons will be copied into.
- Third, a **ReverseIndex** primitive uses the result of the Exclusive Scan to do the final primitive, a Gather.
- This **Gather** collects the indices of the active tetrahedrons into the output array from this phase.

4.2.3 Screen Space Transformation

- Uses a Map primitive to transform the active tetrahedrons into screen space using the camera transform.
- The result of this phase is an array of m tetrahedrons.

4.2.4 Sampling

- Using the axis-aligned bounding box (AABB), the sampler considers every possible pixel and depth that the cell could contribute to and extracts barycentric coordinates.
- An array containing pixel opacities serves as input. The sampling functor can use this data to make decisions, allowing it to potentially terminate sampling early, much like early ray termination.
- With this information, the sampling functor can decide to abort sampling, in a vein similar to early ray termination.

4.2.5 Compositing

- This last stage iterates across groupings of samples using a Map primitive again.
- The functor uses the samples and pixel information to composite the color for that pixel, at least with the samples seen so far.

```
/*Input*/
array: float tetCoords[$N$*12]
/*Output*/
array: byte pixels[$w * h$]
/*Local Arrays*/
array: byte passRanges[$N$*2] //min pass, max pass
array: bool passFlags[$N$]
array: int currentTets[$M$]
array: int indxScan[$M$]
array: int gatherIndxs[$M$]
array: float screenSpaceTets[$M$*12]
array: float samples[($w * h$) / $numPasses$]
//Initialization
passRanges← map<FindPasses>(tetCoords)
for $pass = 0 < numPasses$ do
    //Pass Selection
    flags← map<Thresh>(passRanges, $pass$)
    $m \leftarrow$ reduce<Add>(flags)
    indxScan←scan<Exclusive>(flags)
    gatherIndxs←reverseIndex<>(indxScan,flags)
    currentTets←gather<>(tetIndxs,gatherIndxs)
    //Screen Space Transformation
    screenSpaceTets←map<ScreenSpace>(currentTets,tets)
    //Sampling
    samples←map<Sampler>(screenSpaceTets,pixels)
    //Compositing
    pixels←map<Composite>(samples,pixels)
end
```

Figure 1: Data-Parallel Primitives Pseudo-code

- N = total number of tetrahedrons.
- M = maximum number of tetrahedrons in a single pass.
- m = actual number of tetrahedrons in the current pass.
- w and h are the width and height of the image.

5 Our Testing Options

5.1 Hardware Architectures

- **CPU1**: NERSC's Edison machine, where each node contains two Intel "Ivy Bridge" processors, and each processor contains 12 cores, running at 2.4 GHz. Each node contains 64 GB of memory.
- **CPU2**: the same configuration as CPU1, but using only one of the 24 cores.
- **CPU3**: An Intel i7 4770K with 4 hyper-threaded cores (8 virtual cores total) running at 3.5GHz, and with 32 GB of memory.
- **GPU1**: An NVIDIA GTX Titan Black (Kepler architecture) with 2,880 CUDA cores running at 889 MHz, and with 6 GB of memory

5.2 Data Sets

- **Enzo-1M**: a cosmology data set. This data set was natively on a rectilinear grid, which was then decomposed into tetrahedrons. The total number of tetrahedrons - 1.31 million.
- **Enzo-10M**: a 10.5 million tetrahedron version of Enzo1M.
- **Nek5000**: a 50 million tetrahedron unstructured mesh from a thermal hydraulics simulation.
- **Enzo-80M**: an 83.9 million tetrahedron version of Enzo1M.

5.3 Camera positions

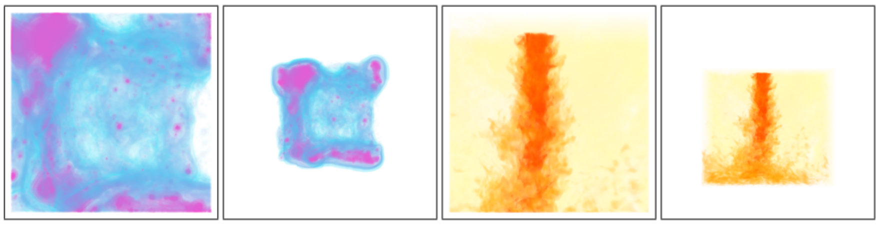


Figure 1: *Volume renderings produced in this study. The two images on the left are of density from a cosmology simulation. The two images on the right are of temperature from a thermal hydraulics simulation. For each pair of images, the larger one is zoomed in (meaning the data set fills the screen) and the smaller one is zoomed out (meaning the data set is surrounded by white space, which is the default view for many visualization tools).*

- Volume renderings for these data sets, including the **zoomed-out** and **zoom-in** camera positions.

6 Results

We must try a few configurations before we can see the results.

6.1 Configuration

Our study consisted of two rounds and 56 total tests.

6.1.1 Round 1: Evaluation of Data-parallel primtives

This round was designed to better understand the basic performance of our volume renderer. It varied three factors:

- Hardware architecture (CPU and GPU): 2 options.
- Data set: 4 options.
- Camera position (zoomed in and zoomed out): 2 options.

We tested on the cross-product of these options: $2 \times 4 \times 2 = 16$
We conducted tests with multiple transfer functions, but their variations did not notably affect the results, so we present results from a single transfer function chosen from our pool.

6.1.2 CPU Performance

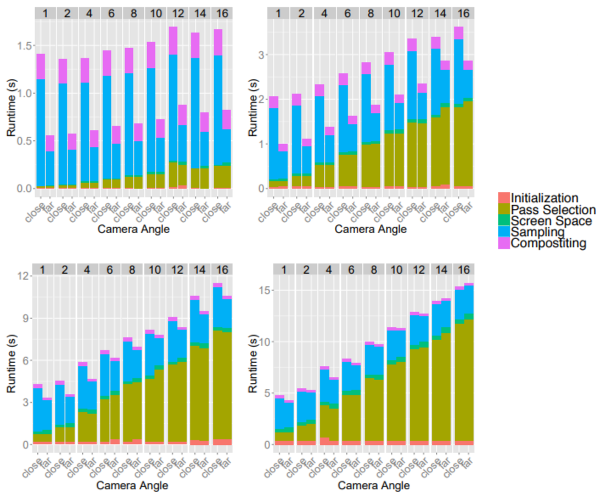


Figure 2: *Running times for our algorithm. **Enzo-1M** is top left, **Enzo-10M** is top right, **Nek5000** is bottom left, and **Enzo-80M** is bottom right. These tests were run on **CPU1** and renderings from both camera angles were made. Within a figure, the number of passes increases from left to right.*

- As the data size grows, the overall time also goes up.
- The amount of work is proportional to the number of samples, as well as the number of cells.

- In small data sets like Enzo-1M, the extraction of the samples dominates the overhead for handling each cell.
- For Enzo-80M, the sampling time is nearly the same for both camera positions
- This is because the number of samples extracted has nearly doubled, so the majority of the time is being spent iterating over tetrahedrons.

6.1.3 GPU Performance

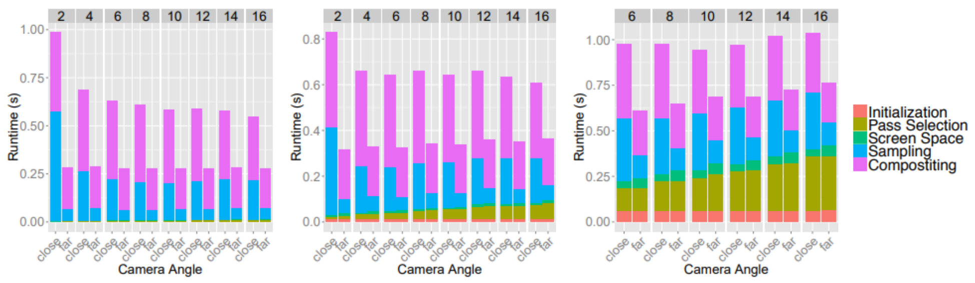


Figure 3: *Running times for our algorithm. From left to right, **Enzo-1M**, **Enzo-10M**, and **Nek5000**. These tests were run on **GPU1** and renderings from both camera angles were made. Within a figure, the number of passes increases from left to right. The **Enzo-80M** run failed, since it was too large for the GPU's 6 GB memory. Further, the **Nek5000** test only has results for 6 passes and above; fewer numbers of passes again ran into the GPU's memory limit.*

- While the dominant factor for CPU performance is sampling time, the dominant factor for GPU performance is compositing time.
- CPUs are typically better suited for tasks that involve branching logic and complex calculations, making them efficient at sampling.
- GPUs are highly parallel processors optimized for performing repetitive, data-intensive tasks in parallel, making them well-suited for compositing, where many pixel values need to be combined simultaneously.

Kernel	Time	Registers	Occupancy
Screen Space	0.008s	70	38%
Sampling	0.202s	57	47%
Compositing	0.416s	37	68%

Table 1: *Elapsed time, registers per thread, and achieved occupancy for a close up view of the **Enzo-10M** data set with four passes on **GPU1**. The statistics for pass selection were omitted since they were difficult to extract; this phase makes use of multiple data-parallel primitives, which in turn each use multiple CUDA kernels.*

- Despite the compositing kernel's lower register usage per thread and higher achieved occupancy, the performance is limited due to the data access pattern and the relatively small number of operations required for the compositing process.

6.1.4 Assessing Performance Portability

- The primary advantage of the data-parallel primitive approach is achieving **portable performance**. We can assess our progress in this regard by analyzing the performance on both CPU and GPU platforms.

Phases	GPU		CPU	
	Time	IPC	Time	IPC
Pass Selection	0.018	1.628	0.514	0.268
Screen Space	0.008	1.704	0.047	0.682
Sampling	0.202	2.477	1.495	1.125
Compositing	0.416	0.131	0.249	1.071

Table 2: *Measurements of CPU and GPU performance, by phase for a close up view of the **Enzo-10M** data set with four passes. The measurements are of time (in seconds) and of instructions executed per cycle (denoted IPC) per core.*

- The instructions per cycle (**IPC**) indicates how data-intensive the computation is.
- Pass Selection should have a low IPC value since it involves iterating through an array of data and performing a few computations on them.
- In this phase, the GPU displays a high IPC value, while the CPU displays a low one. The reason behind this disparity is that GPU-optimized structures meant for quickly combining memory accesses in big arrays do not work well with data-parallel processes.
- The screen space and sampling phases both have high IPC values.
- This aligns with the expectation that these phases are primarily compute-bound, with data movement playing a secondary role in performance. Additionally, the elapsed time is consistent, with the GPU outperforming the CPU due to its significantly higher FLOP capacity.
- The CPU performs as anticipated, while the GPU excels in compute-intensive tasks. However, the efficiency of data-intensive activities like pass selection and compositing depends on the specific use case.
- Portable performance achieved through data-parallel primitives is generally effective, but there are still challenges and limitations to be aware of, indicating that optimization opportunities and pitfalls exist in this context.

6.1.5 Scalability

Threads	1	2	4	8	16	24
Raw Time	43.9	24.2	12.9	7.1	3.8	2.5
Total Time	43.9	48.4	51.7	57.0	60.2	60.7

Table 3: *Times, in seconds, of a strong scaling study. The experiments were run on **CPU1**, using the **Enzo-10M** data set with the close up view and one pass. The times are reported as "total time", meaning the raw time to render the image multiplied by the number of threads. With this measurement, perfect scaling gives a fixed total time over all threads, while poor scaling leads to increases.*

- While adding threads does lead to a dropoff of 50% up to 24 threads (the number of CPU cores on the node), the algorithm appears to scale generally well overall.

6.2 Round 2: Comparison With Community Software

6.2.1 the HAVS volume renderer on the GPU

- A projected tetrahedron algorithm.
- HAVS involves sorting geometry and then rasterizing it, with the sorting step typically performed on the CPU and the sorted geometry transferred to the GPU for rendering.
- Instead, the study evaluated a parallel radix sort on GPU1 for different data sizes and presented the sorting time.

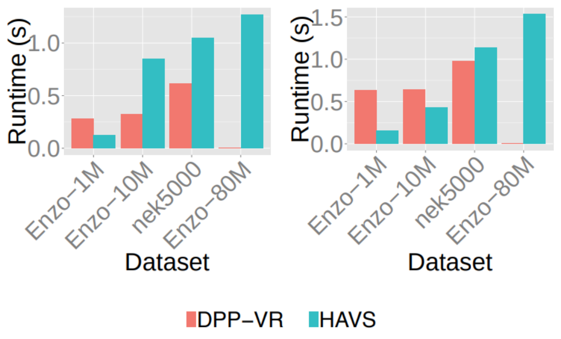
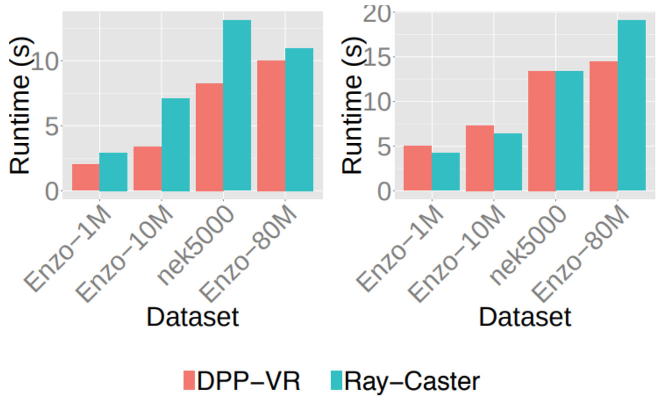


Figure 4: *Comparing the running times for our algorithm and **HAVS** on **GPU1** for multiple data sets. The left figure is for a zoomed out view, and the right figure is for a close up view.*

Our algorithm is slower than HAVS when zoomed in (due to the evaluation of a large number of samples) but faster than HAVS when zoomed out (with fewer samples to evaluate). Our algorithm didn't slow down as quickly as HAVS when data size increased.

6.2.2 the integration-based ray-caste on the CPU

- The study compared the our algorithm running on a CPU to the unstructured ray-caster implemented in VTK
- To address the scalability issues, the study switched to the CPU3 architecture, where the VTK implementation performed better, ensuring a more fair and meaningful comparison.'



Our algorithm performed faster on larger data sets, while the results were mixed on smaller data sets.

6.2.3 Sampling-based ray-caster in VisIt on the CPU.

- Also sampling-based, although it extracts samples by "rasterizing" geometry,
- Designed for distributed-memory parallelism, which involves redistributing samples for compositing after sampling.
- In this comparison, VisIt was executed in serial mode, and our algorithm was also run using CPU2 to maintain hardware parity.

Data & View	SW	SS	S	C	TOT
E-1M/Far	VisIt	0.47	12.9	2.34	15.7
E-1M/Far	DPP-VR	0.17	8.4	2.60	11.5
E-1M/Close	VisIt	0.47	23.5	5.35	29.4
E-1M/Close	DPP-VR	0.13	24.9	5.88	31.1
E-10M/Far	VisIt	3.94	48.3	0.81	53.0
E-10M/Far	DPP-VR	1.41	13.4	2.60	19.2
E-10M/Close	VisIt	4.03	51.8	1.74	57.5
E-10M/Close	DPP-VR	1.06	35.3	5.88	43.9
N-50M/Far	VisIt	24.7	355.5	0.58	391
N-50M/Far	DPP-VR	6.93	24.3	2.92	42.8
N-50M/Close	VisIt	24.8	395	1.02	421
N-50M/Close	DPP-VR	4.93	49.7	5.88	68.7
E-80M/Far	VisIt	33.6	351	0.31	385
E-80M/Far	DPP-VR	11.4	27.6	2.60	55.7
E-80M/Close	VisIt	33.6	361	0.62	396
E-80M/Close	DPP-VR	8.62	55.9	5.88	84.1

Table 4: *Time to volume render a single frame, in seconds.* ***SW*** *indicates the software used, either VisIt, or our data-parallel primitives algorithm (DPP-VR).* ***SS*** *denotes the screen space transformation time (expensive since it is done repeatedly in a multi-pass setting),* ***S*** *denotes the sampling time,* ***C*** *denotes the compositing time, and* ***TOT*** *denotes the total time to make the image*

VisIt's approach is beneficial with large cells (i.e., Enzo-1M), since it is amortizing its calculations. But our approach is faster with small cells(i.e., Enzo-80M), since the overhead VisIt pays per cell is no longer amortized away.

Overall, our algorithm is giving better performance compared to this algorithm which is optimized for a specific platform

7 Conclusion

- We presented a new algorithm for unstructured volume rendering that can composed entirely of data-parallel primitives.
- Moreover, because the algorithm used data-parallel primitives, the real advantages are benefits in portable performance, longevity, and programmability.
- Each new algorithm re-thought in terms of data-parallel primitives, including this one, enables the advancement of data-parallel primitive-based infrastructures that can run on multiple architectures.
- In terms of future work, this algorithm is planned to be extended to distributed memory parallelism.
- This work melds well with the distributed-memory algorithm, the natural extension is to replace its sampling and compositing phases with our data-parallel primitive approach.

References

- [CDM06] CHILDS H., DUCHAINEAU M., MA K.-L.: A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets. In Proceedings of Eurographics Symposium on Parallel Graphics and Visualization (EGPGV) (Braga, Portugal, May 2006)
- [CICS05]CALLAHAN S. P., IKITS M., COMBA J. L. D., SILVA C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. Visualization and Computer Graphics.
- [MAGM11] MORELAND K., AYACHIT U., GEVECI B., MA K.- L.: Dax Toolkit: A Proposed Framework for Data Analysis and Visualization at Extreme Scale. In Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization.
- [MAPS12] MEREDITH J. S., AHERN S., PUGMIRE D., SISNEROS R.: Eavl: the extreme-scale analysis and visualization library.
- [WMFC02] WYLIE B., MORELAND K., FISK L. A., CROSSNO P.: Tetrahedral projection using vertex shaders. In Proceedings of the 2002 IEEE symposium on Volume visualization and graphics