SAND2015-2273C

Visualization for Exascale: Portable Performance is Critical

Kenneth Moreland[1], *Matthew Larsen*[2], *and Hank Childs*[2]

Researchers face a daunting task to provide scientific visualization capabilities for exascale computing. Of the many fundamental changes we are seeing in HPC systems, one of the most profound is a reliance on new processor types optimized for execution bandwidth over latency hiding. Multiple vendors create such accelerator processors, each with significantly different features and performance characteristics. To address these visualization needs across multiple platforms, we are embracing the use of data parallel primitives that encapsulate highly efficient parallel algorithms that can be used as building blocks for conglomerate visualization algorithms. We can achieve performance portability by optimizing this small set of data parallel primitives whose tuning conveys to the conglomerates.

Keywords: scientific visualization, exascale.

Introduction

Although the basic architecture for high-performance computing platforms has remained homogeneous and consistent for over a decade, revolutionary changes are coming. Power constraints and physical limitations are impelling the use of new types of processors, heterogeneous architectures, and deeper memory and storage hierarchies. Such drastic changes propagate to the design of software that is run on these high-performance computers and how we use them.

The predictions for extreme-scale computing are dire. Recent trends, many of which are driven by power budgets, which max out at 20 MW [18], indicate that future high-performance computers will have different hardware structure and programming models to which software must adapt. The predicted changes from petascale to exascale are summarized in Table 1.

Table 1. Comparison of a petascale supercomputer to an expected exascale supercomputer [1].

		Exascale (Prediction)		
System Parameter	Petascale	Swim Lane 1	Swim Lane 2	Factor Change
System Peak	2 PF	1 EF		500
Power	6 MW	≤20 MW		3
System Memory	0.3 PB	32–64 PB		100–200
Total Concurrency	225 K	1 B×10	1 B×100	40,000–400,000
Node Performance	125 GF	1 TF	10 TF	8–80
Node Core Count	12	1,000	10,000	83–830
Network BW	1.5 GB/s	100 GB/s	1,000 GB/s	66–660
System Size (nodes)	18700	1,000,000	100,000	50–500
I/O Capacity	15 PB	300–1,000 PB		20–67
I/O BW	0.2 TB/s	20–60 TB/s		100–300

A particularly alarming feature of Table 1 is the increase in concurrency of the system: up to 5 orders of magnitude. We currently stand about halfway through the transition from petascale to exascale and we can observe this prediction coming to fruition through the use of accelerator

[1] Sandia National Laboratories
[2] University of Oregon

or many-core processors. In the November 2014 Top500 supercomputer list, 75 of the computers contain many-core components, including half of the top 10 computers.

A many-core processor achieves high instruction bandwidth by packing many cores onto a single process. To achieve the highest density of cores at the lowest possible power requirement, these cores are trimmed of latency-hiding features and require careful coordination to achieve peak performance. Although very scalable on distributed memory architectures, our current parallel scientific visualization tools, such as ParaView [2] and VisIt [6], are inadequate on these machines.

Overhauling our software tools is one of the principal visualization research challenges today [7]. A key strategy has been the use of data parallel primitives, since the approach enables simplified algorithm development and helps achieve portable performance.

1. Data Parallel Primitives

Data parallelism is a programming model in which processing elements perform the same task on different pieces of data. Data is arranged in long vectors, and the base tasks apply an operation across all the entities in one or more vectors. Using a sequence of data parallel primitives simplifies expressing parallelism in an algorithm and simplifies porting across different parallel devices. It takes only a few select data parallel primitives to efficiently enable a great number of algorithms [5].

Scientific visualization algorithms typically use data parallel primitives like map, scan, reduce, and sort, which are commonly available in parallel libraries [4, 16]. Several recent research projects for visualization software on next-generation architectures such as Dax [13], PISTON [9], and EAVL [11] use this data parallel approach to execute algorithms [17]. Based on this core similarity, a new project — VTK-m — is combining their respective strengths in execution and data models into a unified framework.

2. Patterns for Data Parallel Visualization

Using data parallel primitives greatly simplifies the process of implementing algorithms on highly-threaded machines and makes these algorithms performance portable. However, implementing many scientific algorithms in terms of data parallel primitives like scan and sort is not straightforward. Fortunately, many scientific visualization algorithms follow familiar algorithmic structures [14], and common patterns emerge.

Three very common patterns in scientific visualization are stream compaction, reverse index lookup, and topology consolidation. In this section we describe these patterns using a Marching-Square-like algorithm applied to the simple example mesh shown in Figure 1.

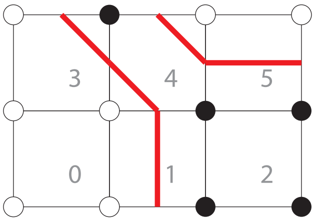

Figure 1. Mesh for contour algorithm examples.

2.1. Stream Compaction

One common feature of visualization algorithms is that the size of the output might depend on the data values in the input and cannot be determined without first analyzing the data. For example, in the mesh of Figure 1 we note that there is no contour in cells 0 and 2, a single contour line in cells 1, 3, and 5, and two contour lines in cell 4. When generating these contour segments in parallel, it is not known where to place the results. We could allocate space assuming the worst case scenario that every cell has the maximum number of contour segments, but that guess tends to be much larger than the actual required memory. Instead, we want to pack the result tightly in an array. This process is known as *stream compaction*. Stream compaction can be performed in two data parallel operations, which are demonstrated in Figure 2 (adapted from Lo, Sewell, and Ahrens [9]).

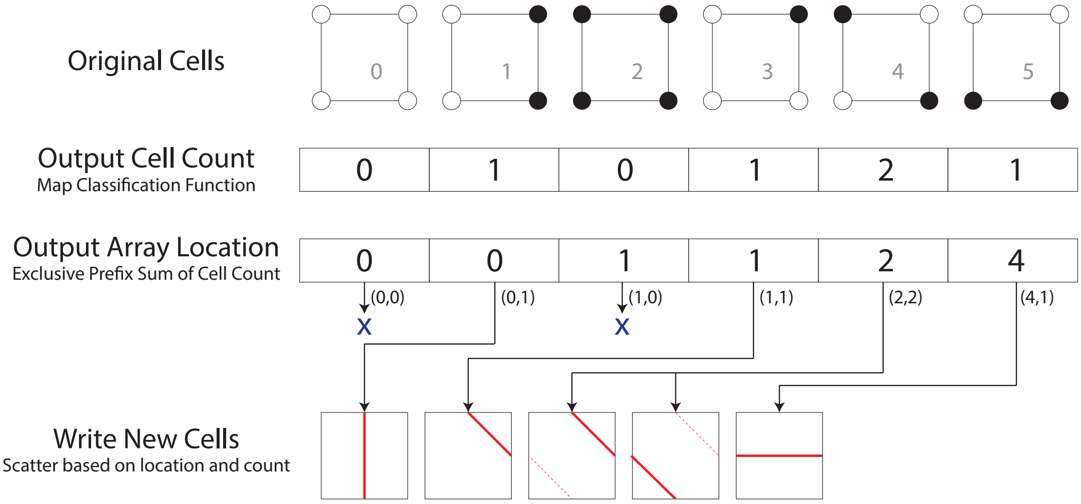

Figure 2. Steps to perform the stream compaction pattern using data parallel primitives.

First, a mapping operation is performed to count the size of the output per cell. Second, an exclusive prefix sum (scan) operation is performed. The result of the prefix some for each entry is the sum of all output up to that point. This sum can be directly used as an index into the compact output array. A final map of the per-element algorithm can now run, placing its results into the appropriate location of the output array.

2.2. Reverse Index Lookup

Directly using the indices from the stream compaction operation results in a *scatter* operation where each thread takes data from an input element and writes to one or more output elements using random access. Although the scatter done by the basic stream compaction is functionally correct, it is known that current many-core processors tend to perform better with *gather* operations where each thread is assigned a single output element but can access random input elements [19]. The steps to reverse the index lookup from a scatter to a gather are demonstrated in Figure 3.

We start with an array that maps each input to the location in its corresponding output location. However, we generate this output array location using an inclusive scan rather than an exclusive scan. This has the effect of shifting the array to the left by one to make the indexing of the next step work better. The next step is to search for the upper bound of the array location for each output element index. The upper bound will be the first entry greater than the value we search for. This search requires the target array location indices to be sorted, which it assuredly is because it is generated from a prefix sum. The search for every index can be done independently in parallel.

The results from the upper bound give is the reverse map from output index to input index. However, a problem that arises is that multiple output elements may come from the same input elements but are expected to produce unique results. In this example input cell 4 produces two

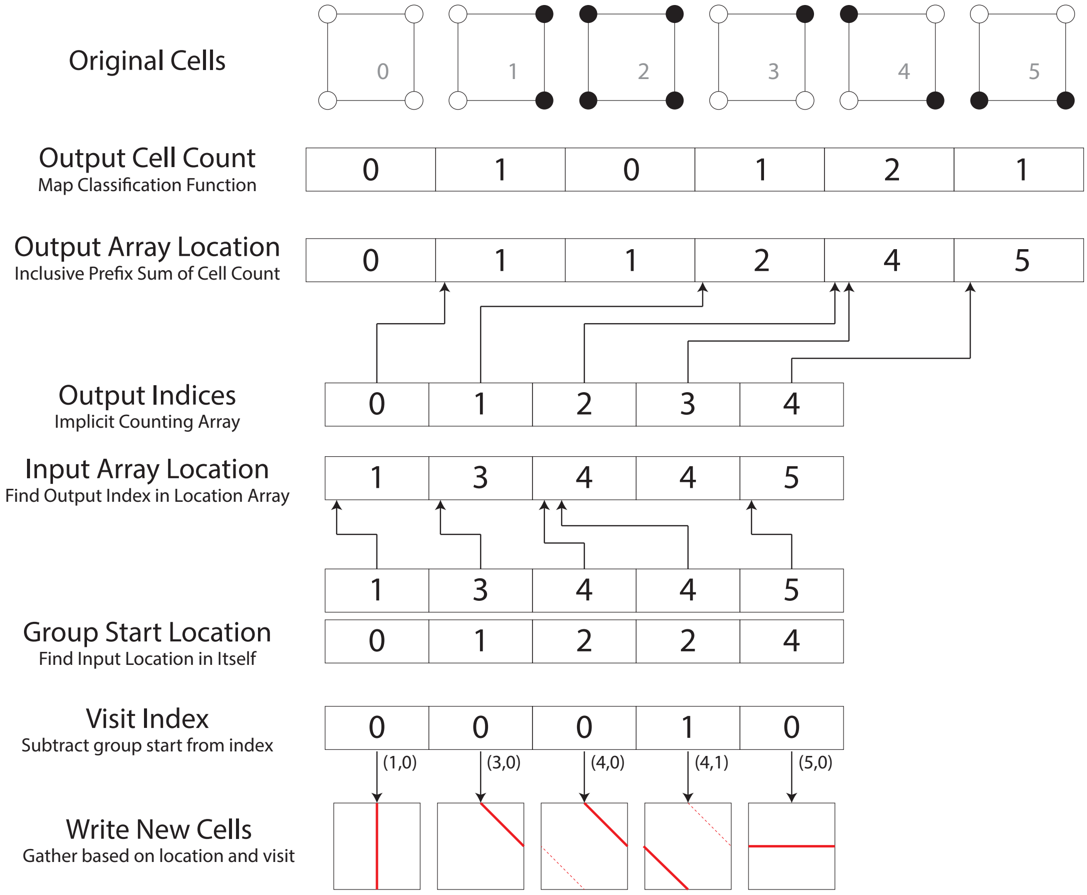

Figure 3. Steps to perform a reverse lookup after stream compaction using data parallel primitives.

contour elements, so two entries in the input array location map point to that cell. How are the threads running on these two cells to know which element to produce? We solve this problem by generating what we call a *visit index*.

The visit indices are generated in two steps. First, we perform a lower bound search of each value in the input array location map into the same map. The lower bound search finds the last entry less than the value we 1parallel. The result is the index to the first entry in the input array location map for the group associated with the same input element. We then take this array of indices and subtract them from the output index to get a unique index into that group. We call this the visit index. Using the pair from input array location map and the visit index, each thread running for a single output element can uniquely generate the data it is to produce.

2.3. Topology Consolidation

Another common occurrence in visualization algorithms is for independent threads to redundantly create coincident data. For example, output elements 0 and 3 from Figures 2 and 3 come from cells 1 and 4, respectively, in Figure 1 and share a vertex. This shared vertex is independently interpolated in separate threads and the connection of these output elements is lost. It is sometimes required to consolidate the topology by finding these coincident elements and merging them.

The general approach to topology consolidation is to define a simple hash for each element that uniquely identifies the element for all instances. That is, two hash values are equal if and only if the associated elements are coincident. Once hashes are generated, a sort keyed on the hashes moves all coincident elements to be adjacent in the storage arrays. At this point it is straightforward to in parallel designate groups of coincident elements and reduce the groups to a single element.

For the specific case of merging vertices, Bell [3] proposes using the point coordinate triple as the hash. However, that approach is intolerant to any numerical inaccuracy. A better approach is to use integer-based hashes, which can usually be derived from the input topology. For example, contour algorithms like Marching Cubes always define contour vertices on the edges of the input mesh. These edges (and therefore the vertices) can be uniquely defined either by an enumerated index or by the pair of indices for the edge's vertex endpoints. Miller, Moreland, and Ma [12] show this approach is faster than using point coordinates and can also be applied to topological elements other than vertices.

3. Results

One of the promises of using data parallel primitives to build scientific visualization algorithms is performance portability. That is, a single implementation using data parallel primitives should work well across computing devices with vastly different performance characteristics from traditional latency-optimized multi-core CPUs to bandwidth-optimized many-core GPUs. Furthermore, portable data parallel primitive implementations should have close to the performance of a non-portable algorithm designed and optimized specifically for a particular device. Recent research indicates that data parallel primitive algorithms are in fact quite performance portable.

Maynard et al. [10] compare a threshold algorithm written with data parallel primitives across many devices. The algorithm shows good performance on both multi-core CPU and many-core GPU devices. Interestingly, the data parallel primitive algorithm running serially on a single CPU core still beats the equivalent VTK implementation.

Lo, Sewell, and Ahrens [9] demonstrate the performance of a Marching Cubes algorithm implemented with data parallel primitives. Their algorithm is compared with the equivalent CUDA reference implementation optimized for that architecture. The two implementations get comparable performance. The data parallel primitive implementation is also shown to get good performance and scalability on multi-core CPUs.

But perhaps the most encouraging evidence comes from a recent performance study conducted by Larsen et al. [8] for ray tracing in the context of data parallel primitives. Ray tracing is a challenging use case since it is computationally intense and contains both regular and irregular memory access patterns. Moreover, this is an algorithm with "guaranteed not to exceed" standards, in the form of Intel's Embree [20] and NVIDIA's OptiX [15]. These products each are supported by teams of developers and have been under development for multiple years. Further, they make full use of architectural knowledge, including constructs like instrinsics, and tune for Intel and NVIDIA products, respectively.

Larsen implements his ray tracer within EAVL and provides a performance study against OptiX on multiple NVIDIA GPUs and against Embree on Intel Xeon and Xeon Phi architectures. His study includes both scientific data sets and standard ray tracing data sets (e.g., Stanford dragon). Figure 4 shows one of the scientific data sets.

Encouragingly, the performance comparison finds that the EAVL ray tracer is competitive with the industry standards. It is within a factor of two on most configurations and does particularly well on the scientific data sets. In fact, it even outperforms the industry standards on some older architectures (since the industry standards tend to focus on the latest architectures).

Overall, this result is encouraging regarding the prospects for portable performance with data parallel primitives, in that a single, architecture-agnostic implementation was comparable to two highly-tuned, architecture-specific standards. Although the architecture-specific standards are clearly faster, the gap is likely acceptable for our use case. Further, the data parallel primitive approach is completed by a graduate student in a period of months whereas the industry standards take experts years (or more); the encumbrence from data parallel primitives could actually be even smaller given additional effort and expertise.

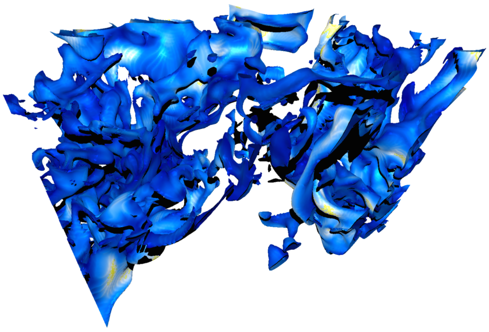

Figure 4. Rendering from ray tracing study on an isosurface of 650,000 triangles.

4. Conclusion

Visualization software will need significant changes to excel in the exascale era, both to deal with diverse architectures and to deal with massive concurrency within a node. Recent results show that data parallel primitives are a promising technology to deal with both challenges. First, exploration into multiple algorithms have shown recurring trends, and will hopefully serve as a precursor to porting many of our community's algorithms reusing these same trends. Second, studies comparing performance with architecture-specific implementations have shown that the performance is very good. Researchers in this area — including the authors of this paper — are so encouraged that they have banded together to form a new effort, VTK-m, in an endeavor to provide production visualization software to the HPC community.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Numbers 10-014707, 12-015215, and 14-017566.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

1. Sean Ahern, Arie Shoshani, Kwan-Liu Ma, et al. Scientific discovery at the exascale. Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, February 2011.

2. Utkarsh Ayachit. *The ParaView Guide: A Parallel Visualization Application*. Kitware Inc., 4.3 edition, January 2015. ISBN 978-1-930934-30-6.

3. Nathan Bell. High-productivity CUDA development with the thrust template library, 2010.

4. Nathan Bell and Jared Hoberock. *GPU Computing Gems, Jade Edition*, chapter Thrust: A Productivity-Oriented Library for CUDA, pages 359–371. Morgan Kaufmann, October 2011.

5. Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990. ISBN 0-262-02313-X.

6. Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, pages 357–372. October 2012.

7. Hank Childs, Berk Geveci, Will Schroeder, Jeremy Meredith, Kenneth Moreland, Christopher Sewell, Torsten Kuhlen, and E. Wes Bethel. Research challenges for visualization software. *IEEE Computer*, 46(5):34–42, May 2013. DOI 10.1109/MC.2013.179.

8. Matt Larsen, Jeremy Meredith, Paul Navrátil, and Hank Childs. Ray-Tracing Within a Data Parallel Framework. In *Proceedings of the IEEE Pacific Visualization Symposium*, Hangzhou, China, April 2015. (to appear).

9. Li-ta Lo, Christopher Sewell, and James Ahrens. PISTON: A portable cross-platform framework for data-parallel visualization operators. pages 11–20. Eurographics Symposium on Parallel Graphics and Visualization, 2012.

10. Robert Maynard, Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. Optimizing threshold for extreme scale analysis. In *Visualization and Data Analysis 2013, Proceedings of SPIE-IS&T Electronic Imaging*, February 2013. DOI 10.1117/12.2007320.

11. J. S. Meredith, S. Ahern, D. Pugmire, and R. Sisneros. EAVL: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 21–30. The Eurographics Association, 2012.

12. Robert Miller, Kenneth Moreland, and Kwan-Liu Ma. Finely-threaded history-based topology computation. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2014. DOI 10.2312/pgv.20141083.

13. Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. 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*, pages 97–104, October 2011.

14. Kenneth Moreland, Berk Geveci, Kwan-Liu Ma, and Robert Maynard. A classification of scientific visualization algorithms for massive threading. In *Proceedings of Ultrascale Visualization Workshop*, November 2013. DOI 10.1145/2535571.2535591.

15. Steven G Parker et al. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)*, 29(4):66, 2010.

16. James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly, July 2007. ISBN 978-0-596-51480-8.

17. Christopher Sewell, Jeremy Meredith, Kenneth Moreland, Tom Peterka, Dave DeMarle, Li-Ta Lo, James Ahrens, Robert Maynard, and Berk Geveci. The SDAV software frameworks for visualization and analysis on next-generation multi-core and many-core architectures. In *2012 SC Companion (Proceedings of the Ultrascale Visualization Workshop)*, pages 206–214, November 2012. DOI 10.1109/SC.Companion.2012.36.

18. Rick Stevens, Andrew White, et al. Architectures and technology for extreme scale computing. Technical report, ASCR Scientific Grand Challenges Workshop Series, December 2009.

19. John A. Stratton, Christopher Rodrigues, I-Jui Sung, Li-Wen Chang, Nasser Anssari, Geng Liu, Wen mei W. Hwu, and Nady Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *IEEE Computer*, 48(8):26–32, August 2012. DOI 10.1109/MC.2012.194.

20. Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (proceedings of SIGGRAPH)*, 33(4):143, 2014.