LA-UR-12-10227

Title:	PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators
Author(s):	Li-ta Lo Christopher Sewell James Ahrens
Intended for:	Eurographics Symposium on Parallel Graphics and Visualization, May 13-14, 2012

Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs and T. Kuhlen (Editors)

PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators

Li-ta Lo[1], Christopher Sewell[1] and James Ahrens[1]

[1]CCS-7, CCS Division, Los Alamos National Laboratory, Los Alamos, NM 87544

Abstract

Due to the wide variety of current and next-generation supercomputing architectures, the development of high-performance parallel visualization and analysis operators frequently requires re-writing the underlying algorithms for many different platforms. In order to facilitate portability, we have devised a framework for creating such operators that employs the data-parallel programming model. By writing the operators using only data-parallel primitives (such as scans, transforms, stream compactions, etc.), the same code may be compiled to multiple targets using architecture-specific backend implementations of these primitives. Specifically, we make use of and extend NVIDIA's Thrust library, which provides CUDA and OpenMP backends. Using this framework, we have implemented isosurface, cut surface, and threshold operators, and have achieved good parallel performance on two different architectures (multi-core CPUs and NVIDIA GPUs) using the exact same operator code. We have applied these operators to several large, real scientific data sets, and have open-source released a beta version of our code base.

Categories and Subject Descriptors (according to ACM CCS): D.1.3 [Computer Graphics]: Concurrent Programming—Parallel programming

1. Introduction

While advances in supercomputer design are providing ever-increasing levels of performance and degrees of parallelism, there has not yet been a convergence towards a single type of architecture, nor does such a convergence appear to be on the horizon. For example, currently among national laboratories in the United States, Argonne has the Blue Gene based Intrepid Supercomputer, Los Alamos has the Cell-based Roadrunner Supercomputer, and Oak Ridge is converting the Jaguar Supercomputer to a GPU-based system. In its exascale supercomputing initiative, the United States Department of Energy has made it clear that systems using at least two different architectures will be built. Thus, a scientist who would like to develop visualization or analysis operators that take advantage of the available computational power and parallelism is often faced with the challenge of having to re-implement his/her operators whenever his/her code is to be run on another system.

In an attempt to address such issues, we have developed "PISTON", a cross-platform framework for visualization and analysis operators that enables a scientist to write an operator algorithm using a platform-independent data-parallel programming model that can then be automatically compiled on the backend to executables for different target architectures. In this paper, we first highlight some relevant related work and then describe our programming model and the implementation of three visualization operators using this framework. Finally, we present some performance results for our operators running on different architectures and close with our planned future work and conclusions.

2. Related Work

The primary contribution of PISTON is that it allows for the development of visualization and analysis operators that achieve both portability and performance. The exact same operator code can be compiled for multiple architectures, taking advantage of the parallelism of each.

Current production visualization software packages, such as VTK, ParaView, or Visit, typically provide a large set of visualization and analysis operators, but, in general, these operators do not take full advantage of acceleration hardware or multi-core architectures. There are a few instances of operators that exploit the parallelism of a particular ar-

chitecture, such as shaders using GLSL on GPUs in VTK, but these are fairly rare, and only target a single architecture. The visualization literature is of course full of research on using parallelism to accelerate operators (e.g., [SEL11, NBE*04, MSM10, JLZ*09]). However, again, each acceleration technique generally only targets one specific architecture rather than being applicable across platforms. Furthermore, each of these accelerated operators tend to be implemented only in a different individual's research code and are not easily available to users as part of a comprehensive visualization and analysis library.

There are also several on-going efforts in programming language design to incorporate portability and abstraction in high-performance computing (e.g., [MAG*11, MIA*07, DJP*11, DG08, CPA*10]). In theory, visualization and analysis operators written using such a language could achieve performance and portability. Nevertheless, these efforts are still relatively early in the research phase. Thus, we felt it was imperative that we proceed with the development of our framework to enable the development of cross-platform data-parallel visualization and analysis operators. There is an immediate and pressing need for such operators. Furthermore, this work allows us to begin to identify and solve challenges specific to the development of these operators so that they might be easily integrated into a portable language once one is fully developed and gains traction within the community.

OpenCL is a widely-used language that is supported by a number of hardware vendors and provides portability across multiple architectures [MUN10]. However, it has several limitations in its use for the development of cross-platform visualization and analysis operators. For instance, it is not supported by a number of architectures, such as Blue Gene. Furthermore, it requires programming at a very low level (using a subset of C99), and, while a given OpenCL program can be compiled and run on different platforms, running efficiently on different architectures usually requires a number of low-level, platform-specific optimizations. Thus, while the code itself may be portable, its performance generally is not. Some have made the case that OpenCL can achieve performance equal to CUDA, but still concede that architecture-specific optimizations are necessary to achieve good performance with either OpenCL or CUDA [FVS11]. In contrast, by restricting the programmer to using a fixed set of high-level data-parallel primitives, each of which can be efficiently implemented in the backend for different architectures, PISTON enables the developer to write high-level code that achieves fairly good performance across all platforms supported by the backend.

3. Design

3.1. Data-Parallel Programming with Thrust

Data parallelism is a programming model in which independent processors perform the same operation on different pieces of data. In contrast, using task parallelism, the processors may simultaneously perform different operations on different pieces of data. Due to the ever-increasing data sizes with which we are faced, we expect data parallelism to be an effective method for exploiting parallelism on current and next generation architectures. The theoretical basis for the data parallel programming model is laid out in Guy Blelloch's seminal work [BLE90].

Thrust [THR12] is a C++ template library released by NVIDIA. Its primary target is CUDA, but it can also target OpenMP, and its design supports the development of additional backends targeting other architectures. It works, in effect, as a source-to-source compiler: high-level C++ code that makes calls to Thrust data-parallel primitive operators is filled in with a CUDA kernel invocation or OpenMP code implementing the primitive with the given parameters. Thrust allows the user to program using an interface that is very similar to (although not identical to) the C++ Standard Template Library. It provides host and device vector types (analogous to std::vector in the STL) that reside in the memory of the host and of the computation device, respectively, which simplify memory management and the transfer of data between the host and the device.

```
input                     4  5  2  1  3
-------------------------------------
transform(+1)             5  6  3  2  4
inclusive_scan(+)         4  9 11 12 15
exclusive_scan(+)         0  4  9 11 12
exclusive_scan(max)       0  4  5  5  5
transform_inscan(*2,+) 8 18 22 24 30
for_each(-1)              3  4  1  0  2
sort                      1  2  3  4  5
copy_if(n % 2 == 1)       5  1  3
reduce(+)                             15

input1                    0  0  2  4  8
input2                    3  4  1  0  2
-------------------------------------
upper_bound               3  4  2  2  3
permutation_iterator      4  8  0  0  2
```

Listing 1: *Example data-parallel primitive operations*

Most of the algorithms that Thrust provides are data-parallel primitives and operate on the host and device vector types. The challenge for the developer is to design his/her operator algorithms to use only these primitives. The reward for the developer is that his/her operator code will then be very efficient and portable. These primitives include `sort`, `transform`, `reduce`, `inclusive_scan`, `exclusive_scan`, `transform_inclusive_scan`, `transform_exclusive_scan`, `copy_if` (stream compaction), `upper_bound` (a binary search which finds,

for each element of the second input array, the highest index in the first sorted input array at which the element could be placed without breaking the sorted ordering), `for_each`, and `permute` (`permutation_iterator`). Each of these primitives can work with user-defined data types and functors as well as with standard C++ data types. Examples of several of these data-parallel primitives, as implemented in Thrust, are given in Listing 1.

3.2. Isosurface Operator

The first visualization operator we implemented in PISTON was isosurfaces for structured grids. A naive data-parallel algorithm is presented in Figure 1. The illustration is for a 2D contour, but easily generalizes to 3D isosurfaces. The top row shows an example set of grid cells. Vertices with associated scalar values on one side of the specified isovalue are marked with black dots, while those with scalar values on the other side of the specified isovalue are unmarked. The `copy_if` primitive can be used to compact "valid" cells; i.e., those that will generate geometry because they contain at least one vertex below and at least one vertex above the isovalue. Using a standard marching-cubes look-up table keyed by the pattern of marked and unmarked vertices [LC87], contours are generated for each valid cell. Some valid cells generate "phantom" geometry, marked with a flag, such that each valid cell generates the same number of vertices. This follows the standard parallel computing technique of reducing branching within the kernel by doing extra computation and then filtering out unneeded results. In addition, it allows each thread to know in advance the offset into the output vertex array to output the geometry it generates. In a final pass, a `copy_if` is used to stream compact the real (non-phantom) vertices to produce the final output array. While this approach produces correct results, it is rather inefficient, first because of the large amount of global memory movement incurred by copying all the data (scalar values at each vertex) for each valid cell with the initial `copy_if` pass, and also because of the extra `copy_if` pass at the end to eliminate phantom geometry.

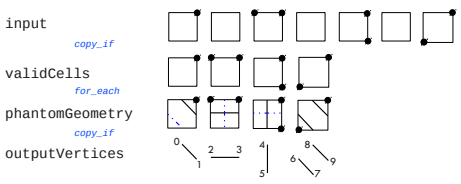

Figure 1: *Naive algorithm for isosurface generation*

Our optimization of this algorithm is loosely based on the idea behind the HistoPyramid algorithm [DZ07]. The general principle is that it generates a "reverse mapping" from output vertex index to input cell index (rather than from input cell index to output vertex index), allowing it to "lazily" apply operations only to cells that will generate the output vertices. Instead of explicitly constructing and traversing a HistoPyramid tree, stream compaction is accomplished with a scan and traversal with a binary search.

Pseudocode for our algorithm is given in Listing 2, and the algorithm is illustrated using example input in Figure 2. The algorithm consists of the following steps:

1. As in Figure 1, the top row (`input`) presents an example set of input grid cells, along with their global indices, with vertices marked or unmarked depending on whether their associated scalar values fall above or below the specified isovalue.
2. Using the `transform` primitive (pseudocode line 1), with a user-defined functor that computes the Marching Cube case number index for a cell based on its pattern of marked and unmarked vertices, a vector of case number indices (`caseNums`) is generated.
3. Also using this `transform` primitive, a vector of the number of output vertices generated by each cell is produced (`numVertices`).
4. A `transform_inclusive_scan` (pseudocode line 4) with a functor that returns one for any value greater than zero is then performed on the number of vertices in `numVertices` to enumerate the valid cells (`validCellEnum`). The last element of this vector indicates the total number of valid cells (pseudocode line 6).
5. A binary search (`upper_bound`, pseudocode line 7) is then performed on a counting iterator (`Counting-Iterator`) that enumerates the valid cells (zero through the total number of valid cells minus one) searching in the `validCellEnum` vector to find the index of the first element greater than each counting iterator element.
6. The result of this search is shown in the sixth row (`validCellIndices`).
7. This compact vector of global indices of the valid cells is used to fetch the number of output vertices for each valid cell using a `permutation_iterator` (`numVerticesCompacted`), which exists only as an unnamed temporary variable generated by `make_permutation_iterator` in the pseudocode (line 10).
8. Finally, an `exclusive_scan` on `numVer-ticesCompacted` (pseudocode line 10) gives the starting offset into the global output vertex array for each valid cell (`numVerticesEnum`). The total number of vertices in the output (pseudocode line 14) is the sum of the last elements of the final two vectors (the starting offset of the final valid cell plus the number of vertices produced by the final valid cell).
9. The vertex list output by the algorithm using `for_each` (pseudocode line 15) is shown in the final row (`outputVertices`).

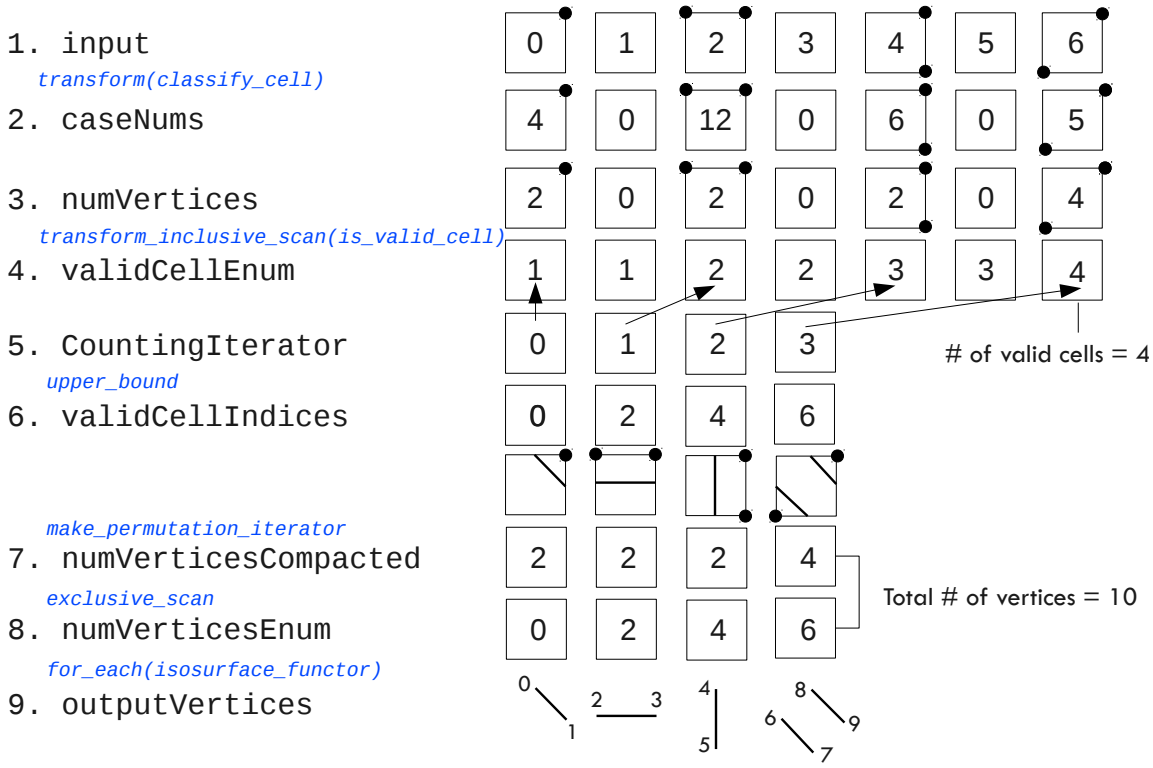

Figure 2: *Optimized algorithm for isosurface generation*

In summary, our algorithm offers a number of efficiency advantages. It avoids branching within the kernels, except within the binary search. However, the effect of the branching within the binary search is minimized since the input is in sorted order, allowing threads for nearby elements to follow the same branching pattern. There is no global memory movement in which all cell data is copied from widely scattered memory locations (as in the initial `copy_if` of the naive algorithm). Instead, only indices into the original input vector need to be written to the valid cell vectors. There is no needless waste of computing resources due to the computation of extraneous "phantom" geometry. In addition, the algorithm takes advantage of kernel fusion provided by Thrust's "combination" operators, such as `transform_inclusive_scan`, and by the use of Thrust's `permutation_iterator` with Thrust operators. We found our optimized algorithm to be approximately one order of magnitude faster than the naive algorithm. Furthermore, the design of this algorithm is general enough that it can be easily adapted to perform several other operations, such as cut surfaces and thresholds, as described in the following subsections.

3.3. Cut Surface Operator

A cut surface is described with an implicit function, and data attributes are mapped onto the cut surface [SML06]. The algorithm for our cut surface operator follows directly from the isosurface operator. Two scalar fields are associated with the structured grid. The first is derived from the equation of the cut surface; the isosurface of this field yields the cut surface geometry. The second scalar field is the data field (the same field as is used as the input to the plain isosurface operator as described in the previous section) and is used for coloring the cut surface according to the color map. Only one interpolation parameter needs to be computed per vertex, as it is used to determine both the vertex location and the scalar value at that position. Since all vertices generated by the Marching Cube algorithm are located on edges of the structured grid, the interpolation parameter can be computed using only a one-dimensional interpolation between the val-

```
thrust::transform(CountingIterator(0), CountingIterator(0)+Ncells,
        thrust::make_zip_iterator(thrust::make_tuple(caseNums.begin(),
        numVertices.begin()), classify_cell(input, ...));
thrust::transform_inclusive_scan(numVertices.begin(), numVertices.end(),
        validCellEnum.begin(), is_valid_cell(), thrust::plus<int>());
numValidCells = validCellEnum.back();
thrust::upper_bound(validCellEnum.begin(), validCellEnum.end(),
        CountingIterator(0), CountingIterator(0)+numValidCells,
        validCellIndices.begin());
thrust::exclusive_scan(thrust::make_permutation_iterator(numVertices.begin(),
        validCellIndices.begin()),
        thrust::make_permutation_iterator(numVertices.begin(),
        validCellIndices.begin() + numValidCells, numVerticesEnum.begin());
numTotalVertices = numVertices[validCellIndices.back()] + numVerticesEnum.back();
thrust::for_each(thrust::make_zip_iterator(thrust::make_tuple(
        validCellIndices.begin(), numVerticesEnum.begin(),
        thrust::make_permutation_iterator(caseNums.begin(),
        validCellIndices.begin()),
        thrust::make_permutation_iterator(numVertices.begin(),
        validCellIndices.begin()))), ..., isosurface_functor(outputVertices, ...));
```

Listing 2: *Pseudocode for isosurface algorithm*

ues at two grid points, as opposed to needing to compute a full trilinear interpolation.

3.4. Threshold Operator

Thresholding selects data that lies within a range of data [SML06]. Our threshold operator also makes use of the same basic algorithmic design as the isosurface operator. Using the transform primitive, a vector of "valid" flags is generated indicating which cells will generate geometry. In this case, only cells for which the scalar value at each vertex falls within the specified threshold range will be classified as "valid". A compact vector of the global indices of the valid cells is generated, as with the isosurface operator, by performing a binary search for the elements of a counting iterator within the result of an inclusive scan of the vector of valid flags. Several additional operations are performed in the threshold operator in order to identify "interior" valid cells that are fully contained inside a block of other valid cells. Such cells would have no surfaces visible on the exterior, and so no geometry needs to be generated for such cells. Interior cells are identified and enumerated amongst the valid cells in the same way as valid cells are identified and enumerated amongst all grid cells, except that in this case the functor used by the transform primitive sets the flag based on whether the given cell has any non-valid neighbor cells. In the final step, six quads are generated for each valid, exterior cell to cover each of its six faces.

3.5. Applicability to Additional Operators

The data-parallel programming model illustrated by the isosurface, cut surface, and threshold operators presented here is applicable to a wide variety of additional types of operators. In [BLE90], algorithms are presented for such diverse problems as generalized binary search, closest pair, quickhull, merge hull, maximum flow, minimum spanning tree, matrix-vector multiplication, and linear system solving, all using only transform, permute, and scan primitive operators on vectors, plus some simple scalar and scalar-vector operators. We have plans to implement a simple glyph operator that uses Thrust's `for_each` primitive with a functor that rotates, scales, translates, and colors the input icon at a given point, and a render operator that builds a k-d tree using Thrust's `sequence`, `fill`, `reduce`, `permutation_iterator`, `copy`, `count_if`, and `copy_if` primitives and traverses the tree `for_each` pixel in a ray-casting algorithm. Many basic statistical operators could be constructed using Thrust's `reduce` primitive paired with appropriate functors to compute maximums, minimums, means, standard deviations, etc. The flexibility of Thrust's primitives, especially `transform` and `for_each`, with user-defined functors make it fairly easy to construct at least a naive data-parallel algorithm for almost any inherently parallel algorithm. For example, a functor that traces a particle path in a vector field applied over a rake of initial particles with the `for_each` primitive would yield a simplistic streamline algorithm. Achieving optimal performance will still require clever operator-specific algorithmic design, such as the reverse mapping concept that improved on the naive

algorithm for our isosurface operator, but, in contrast to effort spent on making platform-specific algorithm optimizations using other approaches, the gains from such optimizations in a data-parallel algorithm will apply across all supported hardware platforms.

4. Results

4.1. Performance Evaluation

We evaluated the performance of our three operators compiled to a CUDA backend and running on a 448-core NVIDIA Quadro 6000 with 6 GB of memory, and compiled to an OpenMP backend and running on a 48-core 1.9 GHz AMD Opteron 6168.

The expected cost of portability is performance. Therefore, we compared the computation rates of our isosurface operator compiled to CUDA with the native CUDA Marching Cubes demo that is distributed as part of NVIDIA's CUDA SDK [CUDA10]. As shown in Figure 3, the native CUDA demo was somewhat faster than the PISTON implementation. For example, when operating on a data set in a 256^3 grid, the demo was able to compute 74 isosurfaces per second, while PISTON computed 49. The loss of performance is not too surprising, given that the PISTON implementation makes multiple kernel calls, and is not able to make efficient use of shared and texture memory on the GPU, optimize the number of threads per block or blocks per grid, or fine-tune memory access patterns, since these are architecture-specific optimizations. Nevertheless, PISTON's performance is still fairly close to that of the hand-optimized CUDA demo code. Furthermore, a direct comparison of these computation rates is not entirely fair to the PISTON code, because the CUDA demo places several restrictions on the input data that allow it to make additional optimizations. The CUDA demo only operates on data sets that have a grid size that is an exact power of two in each dimension, allowing it to use shifts instead of integer division, and it cannot operate on data sets larger than 512^3 because it runs out of texture and global memory. In contrast, the PISTON isosurface operator allows inputs whose dimesions are not powers of two, and can handle data sets somewhat larger than 512^3 (approximately 700^3 on this GPU with this data set).

Using the exact same isosurface operator code that was compiled to CUDA, we can compile the operator to use OpenMP on a multi-core CPU by just changing a compiler switch that causes Thrust to use the OpenMP backend instead of the CUDA backend. For comparison, we used implementations of the isosurface operator using VTK and using Parallel VTK. Isosurface computation rates using PISTON, VTK, and Parallel VTK are shown in Figure 4. The scan primitive in Thrust as included in NVIDIA's CUDA distribution included only a serial implementation in the OpenMP backend, so we wrote our own parallel version using the standard data-parallel scan algorithm [BLE90]. As

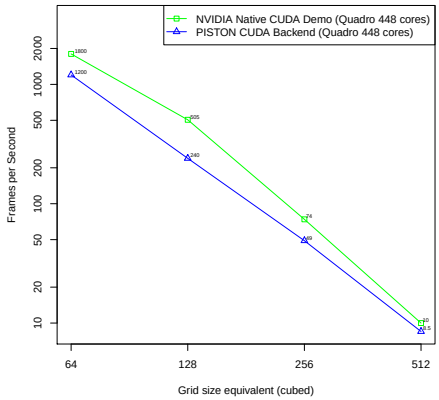

Figure 3: *3D Isosurface Generation: CUDA Compute Rates*

would be expected, the parallel OpenMP PISTON code running on 48 cores was significantly faster than the serial VTK code running on a single core. For example, with a 256^3 data set, PISTON computed 37 isosurfaces per second while VTK computed only one. The parallel VTK implementation was faster than single-core VTK but much slower than the PISTON code, computing 2.7 isosurfaces per second on the 256^3 data set. One likely reason for its inferior performance is that it simply divides the domain equally among the available processors, which results in an unbalanced workload, since the distribution of geometry-producing cells is not uniform. In contrast, by generating the "reverse mapping", PISTON operates only on valid (geometry-producing) cells, resulting in a very balanced workload. As shown in Figure 5, the PISTON OpenMP isosurface code also scales much better with the number of processors than the Parallel VTK implementation, since each of the data-parallel primitives used in the PISTON algorithm is very scalable.

All of the preceeding results reflect only the rates at which isosurfaces are computed and do not include the time taken to render them. Figure 6 presents the rates at which isosurfaces are computed and rendered using PISTON with its CUDA backend and with its OpenMP backend. The OpenMP executable was run on a 12-core 2.67 GHz Intel Xeon X5650 with an NVIDIA Quadro 6000 graphics card (as opposed to the 48-core AMD system used for the preceeding compute-only results). For the CUDA backend, the compute plus render rate depends heavily upon whether the computed results have to be transferred to the CPU and then back to the GPU for rendering. With these transfers, the CUDA backend has very similar performance to the OpenMP backend (8 and 6 isosurfaces computed and rendered per second, respectively, for a 256^3 data set). However, if CUDA's "interop" feature is utilized, vertex buffer objects can be pre-allocated on the GPU, and pointers passed

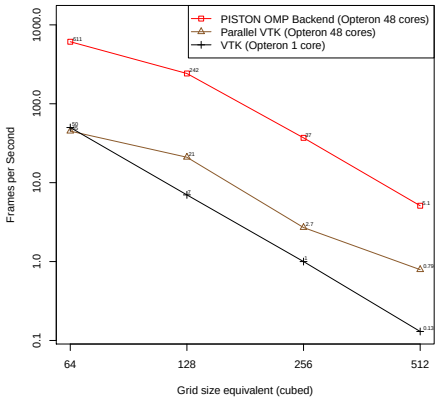

Figure 4: *3D Isosurface Generation: CPU Compute Rates*

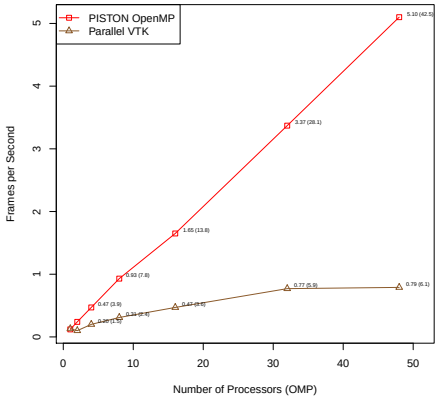

Figure 5: *3D Isosurface Generation: PISTON OpenMP Scaling (Grid size: 512^3); speed-ups versus one processor shown in parentheses*

to the CUDA computation so that the resulting isosurface vertices are output directly to these graphics objects, avoiding the need for a transfer to the CPU and back and reducing the total memory usage on the GPU. With this optimization, PISTON can compute and render 39 isosurfaces per second for the 256^3 data set. Of course, these rates are dependent on the specific hardware and number of cores used, so the point is not to directly compare the backends (for example, with more CPU cores and a less powerful GPU, the OpenMP performance would compare more favorably). Rather, the key point is that we can achieve good performance on different architectures using the exact same code, and that using a simple though architecture-specific optimization (interop) can significantly improve rendering performance on the GPU.

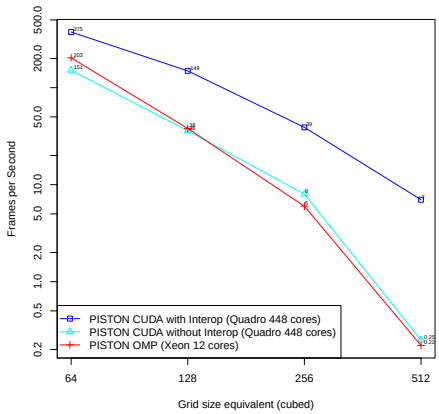

Figure 6: *3D Isosurface Generation: Compute plus Render Rates for PISTON Backends*

The results presented in the preceeding figures all used the isosurface operator. Figure 7 shows that all three operators described in this paper exhibit similar behavior, as all are based on the same basic data-parallel model. In these tests, the cut surface generated was a simple 2D plane, so this operator ran the fastest. The threshold operator was faster than the isosurface operator, despite having additional steps to identify interior cells, presumably because it does not have to compute any interpolations (which involve expensive divisions) for the scalar values nor for vertex positions.

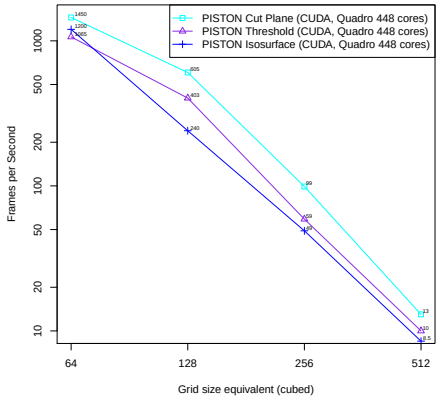

Figure 7: *3D Visualization Operators: CUDA Compute Rates*

4.2. OpenCL Prototype

We have also implemented a prototype OpenCL backend that allows us to run on platforms such as AMD GPUs.

There are a number of challenges involved in creating an OpenCL backend, including the fact that OpenCL is based on C99, making support for C++ features such as templates and classes difficult, and the fact that OpenCL compiles kernels from strings at run-time rather than from source files at compile-time. In our prototype, a pre-processor extracts operators from user-written functors and outputs them to .cl files. At run-time, our Thrust-like backend combines these user-derived .cl files with its own native OpenCL implementations of data-parallel primitives into kernel strings. It uses run-time type information to handle simple templating and functor calls, substituting for key words in the kernel source string. Since the kernel source depends only on the types of the arguments to the data-parallel primitive, the kernel source only needs to be compiled once for each time it appears in the code, not re-compiled each time it is called at run-time. Using this OpenCL backend, we have successfully implemented the isosurface and cut surface operators using code that is almost, but not exactly, identical to that used for the Thrust-based CUDA and OpenMP backends. As with CUDA, we have made an optimization to take advantage of OpenCL's interop feature. On an AMD FirePro V7800 with 1440 streams, we can compute and render about six isosurfaces per second using a 256^3 data set (or about two per second without using interop). Further optimizations may be expected to significantly improve this performance.

4.3. Example Applications with Scientific Data Sets

Example screen shots of these operators being applied to several real scientific data sets are presented in Figure 8. Shown are the isosurface and cut surface (cut plane) operators applied to the density field of a 275x317x192 Rayleigh-Taylor instability data set [LRG*09] running on a multi-core CPU with OpenMP, the threshold and cut surface (cut plane) operators applied to a 600x500x42 ocean eddy data set [WHP*11] running on an NVIDIA GPU with CUDA, the isosurface operator applied to an 1800x1200x42 ocean temperature data set [MPV10] running on an NVIDIA GPU with CUDA, and the isosurface and cut surface (cut plane) operators applied to the 160x80x70 pressure field surrounding a wind turbine [RL11] running on an AMD GPU with OpenCL.

5. Conclusions

We have devised a cross-platform framework for the development of visualization and analysis operators, making use of a data-parallel programming model and of the Thrust library. Using this framework, we have implemented three visualization operators: isosurface, cut surface, and threshold. Our performance results show that we can achieve good parallel performance on two different architectures (multi-core CPUs and NVIDIA GPUs) using the exact same operator code.

We expect our on-going work with PISTON to encompass several general areas of development. First, we intend to implement additional operators for visualization (such as glyphs or streamlines) and for analysis (such as halo finders or FFTs). Supporting unstructured grids for our existing operators is another important goal. Furthermore, we are collaborating with Kitware to integrate PISTON into ParaView, which would allow a user to interact with and chain together PISTON-based filters using a professional GUI while gaining the performance advantages of PISTON's parallel operator implementations on multiple architectures. In addition, PISTON could achieve multi-node parallelism by making use of ParaView's distributed-memory constructs while realizing on-node parallelism using its own backends. Finally, we intend to expand the set of supported architectures on which PISTON operators will run both by testing the existing backends on other platforms (such as OpenMP on Blue Gene) and modifying them as necessary, and potentially by developing new backends for Thrust.

The beta version of our source code has been made open source and is available on our project web page: http://viz.lanl.gov/projects/PISTON.html.

6. Acknowledgements

The work on PISTON was funded by the NNSA ASC CCSE Program, Thuc Hoang, national program manager, Bob Webster and David Daniel, Los Alamos program managers. We would also like to thank all of those who have provided us data sets, including Philip Jones, Mathew Maltrud, Sean Williams, Rodman Linn, Eunmo Koo, Tie Wei, and Daniel Livescu, as well as Jonathan Woodring, John Patchett, and the anonymous EGPGV reviewers for their constructive feedback.

References

[SEL11] Erik Smistad, Anne C. Elster, and Frank Lindseth. "Fast Surface Extraction and Visualization of Medical Images using OpenCL and GPUs". *The Joint Workshop on High Performance and Distributed Computing for Medical Imaging (HP-MICCAI)*, 2011.

[NBE*04] Timothy S. Newman, J. Brad Byrd, Pavan Emani, Amit Narayanan, and Abouzar Dastmalchi. "High Performance SMID Marching Cubes Isosurface Extraction on Commodity Computers". *Computers & Graphics*, Vol. 28 Issue 2, 2004, pages 213-233.

[MSM10] Steven Martin, Han-Wei Shen, and Patrick McCormick. "Multi-GPU Isosurface Extraction". *EGPGV '10: Proceedings of Eurographics Symposium on Parallel Graphics and Visualization 2010*. Pages 91-100, May 2010.

[JLZ*09] Han Jin, Bo Li, Ran Zheng, and Qin Zhang. "Fast Isosurface Extraction for Medical Volume Dataset on Cell BE". *Proceedings of the 2009 International Conference on Parallel Processing*. 2009.

[MAG*11] Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. "Dax Toolkit: A Proposed Framework for

Data Analysis and Visualization at Extreme Scale". *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, October 2011, pp. 97-104. DOI 10.1109/LDAV.2011.6092323.

[MIA*07] Patrick McCormick, Jeff Inman, James Ahrens, Jamaludin Mohd-Yusof, Greg Roth, and Sharen Cummins. "Scout: a data-parallel programming language for graphics processors". *Parallel Computing*, Volume 33 Issue 10-11. November 2007.

[DJP*11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. "Liszt: A Domain Specific Language for Building Portable Mesh-based PDE Solvers". *Supercomputing 2011*, Seattle, Washington.

[DG08] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Communications of the ACM*, 51(1):107-113, January 2008.

[CPA*10] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. H. Weber, and E. W. Bethel. "Extreme Scaling of Production Visualization Software on Diverse Architectures." *IEEE Computer Graphics and Applications*, 30(3):22âĂŞ31, May/June 2010.

[MUN10] A. Munshi. "The OpenCL Specification." Khronos OpenCL WorkingGroup, September 2010. Version 1.1, Revision 36. Project webpage: http://www.khronos.org/opencl/.

[FVS11] Jianbin Fang, A.L. Varbanescu, and H. Sips. "A Comprehensive Performance Comparison of CUDA and OpenCL". International Conference on Parallel Processing (ICPP) 2011, pp.216-225, Sept. 2011. doi: 10.1109/ICPP.2011.45

[BLE90] Guy Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press. ISBN 0-262-02313-X. 1990.

[THR12] Thrust library. Project webpage: http://code.google.com/p/thrust/. Accessed 2012.

[LC87] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High-Resolution 3D Surface Construction Algorithm". *Computer Graphics*, Vol. 21, Num. 4, July 1987.

[DZ07] Christopher Dyken and Gernot Ziegler. "High-speed Marching Cubes using Histogram Pyramids". *Eurographics*, Vol. 26 Num. 3. 2007.

[SML06] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Chapters 6 and 9 and Glossary. Kitware. Fourth Edition, 2006.

[CUDA10] NVIDIA CUDA Programming Guide, Version 3.0, February 2010. Product webpage: http://developer.nvidia.com/gpu-computing-sdk.

[LRG*09] D. Livescu, J.R. Ristorcelli, R.A. Gore, S.H. Dean, W.H. Cabot, and A.W. Cook. "High-Reynolds Number Rayleigh-Taylor Turbulence". *Journal of Turbulence*, Vol. 10, 2009.

[WHP*11] Sean J. Williams, Matthew W. Hecht, Mark R. Petersen, Richard Strelitz, Mathew E. Maltrud, James P. Ahrens, Mario Hlawitschka, and Bernd Hamann. "Visualization and Analysis of Eddies in a Global Ocean Simulation". *EuroVis 2011*, June 2011, Bergen, Norway.

[RL11] E. K. Rodman and R. Linn. "Determining effects of turbine blades on fluid motion". U.S. Patent Application 13/118,307, May 2011.

[MPV10] M. Maltrud, S. Peacock, and M. Visbeck. "On the Possible Long-term Fate of Oil Released in the Deepwater Horizon Incident, Estimated by Ensembles of Dye Release Simulations". *Environmental Research Letters*, 5, doi: 10.1088/1748-9326/5/3/035301. 2010.

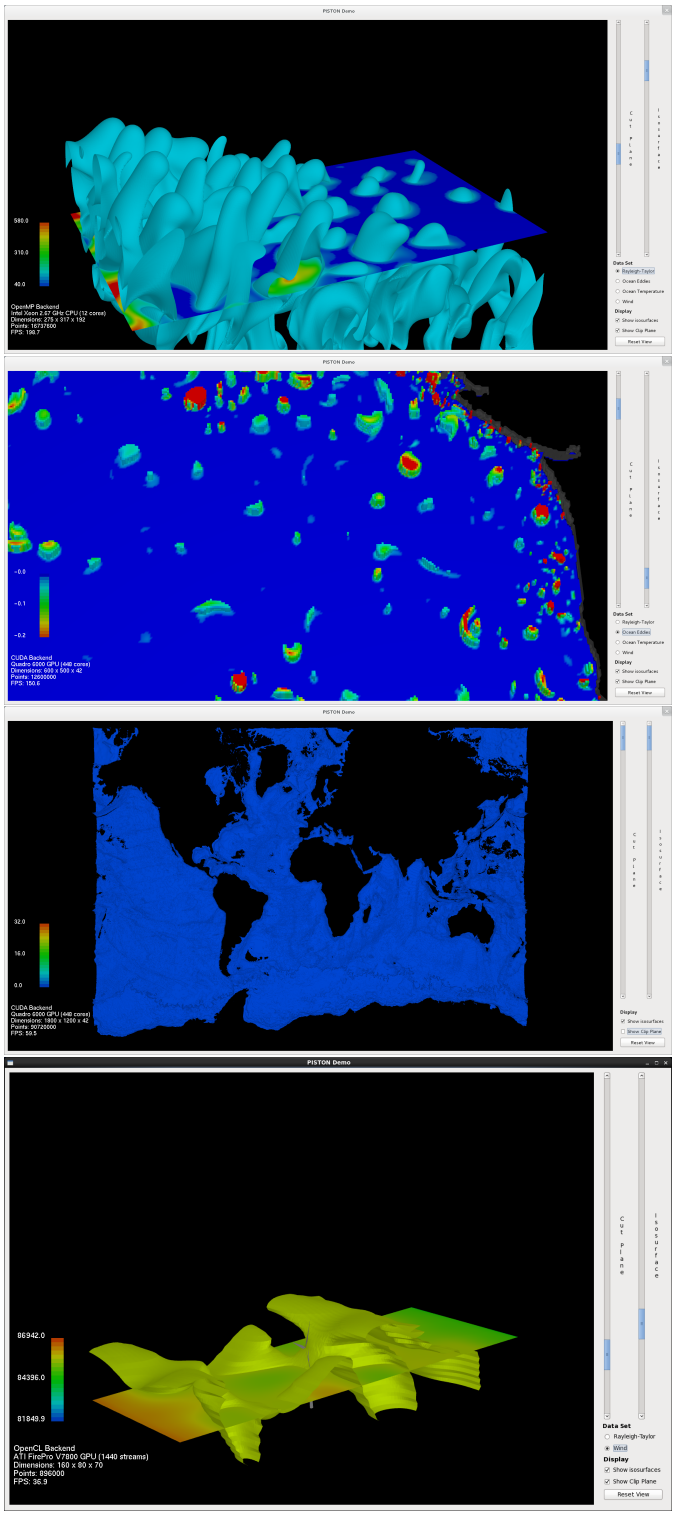

Figure 8: *Screenshots of PISTON operators applied to several scientific data sets: (top to bottom) Rayleigh-Taylor instability density field, ocean eddies, ocean temperature field, and wind turbine pressure field*