LA-UR-11-11980

Approved for public release;
distribution is unlimited.

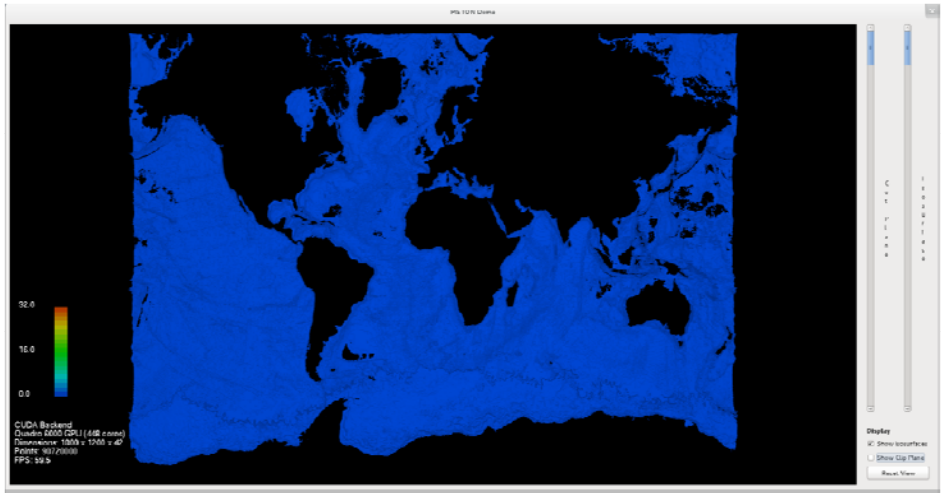
Title: Portability and Performance for Visualization and Analysis Operators Using the Data-Parallel PISTON Framework

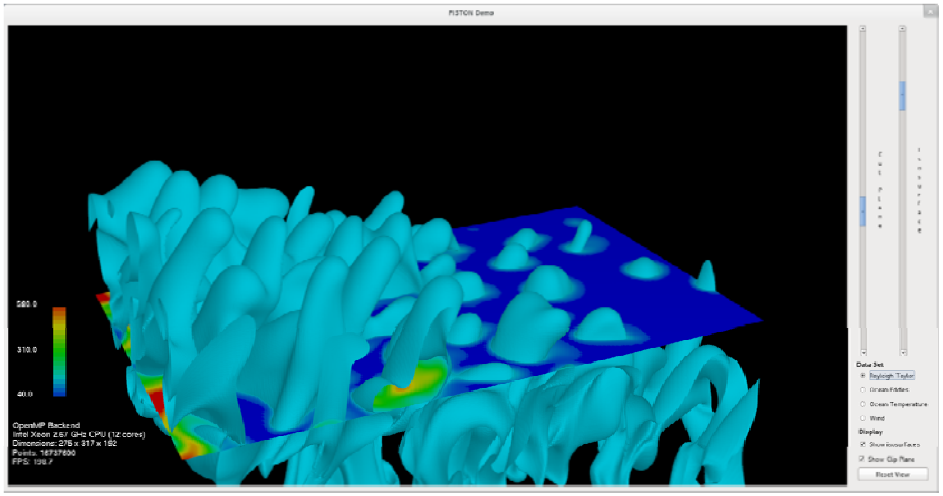
Author(s): Christopher Sewell
Li-ta Lo
James Ahrens

Intended for: Many-Core Library Tutorial and Code Sprint, September 19-20, 2012

Portability and Performance for Visualization and Analysis Operators Using the Data-Parallel PISTON Framework

Chris Sewell
Li-Ta Lo
James Ahrens
Los Alamos National Laboratory





Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA

LA-UR-11-11980

Outline

- Motivation
  - Portability and performance of visualization and analysis operations on current and next-generation supercomputers
- Introduction to data-parallel programming and the Thrust library
- Implementation of visualization operators
  - Isosurface, Cut Surfaces, Threshold
- Current target architectures and performance
  - CUDA/Nvidia GPU & OpenMP/Multi-core machines
- On-going work
  - OpenCL backend, unstructured grids, more operators, ParaView integration, multi-node parallelism, curvilinear coordiantes
- Tutorials

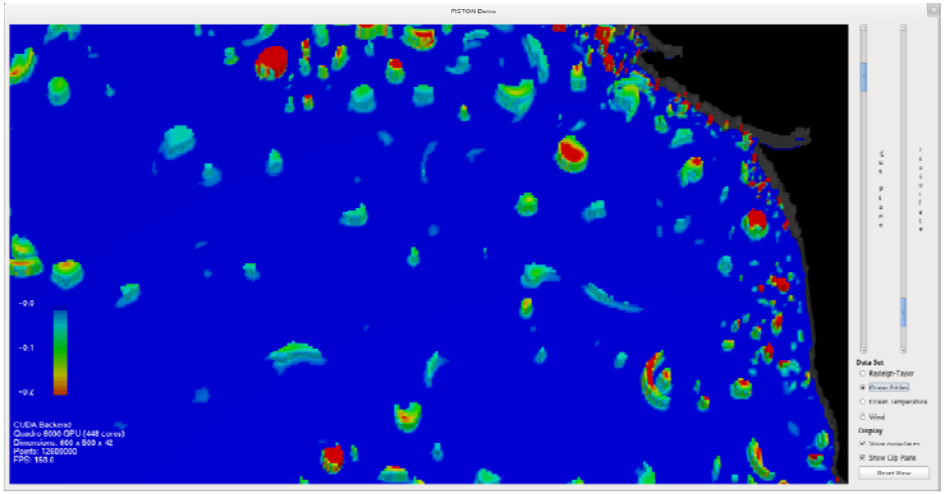
Motivation / Related Work

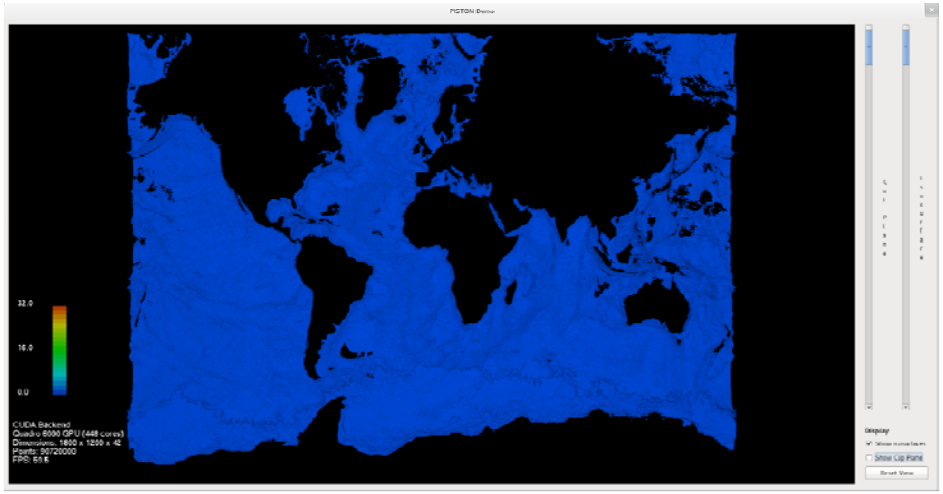
- Current production visualization software does not take full advantage of acceleration hardware and/or multi-core architecture
  - Vtk, ParaView, Visit
- Research on accelerating visualization operations are mostly hardware-specific; few were integrated in visualization software
  - CUDA SDK demo
  - Dyken, Ziegler, "High-speed Marching Cubes using Histogram Pyramids", Eurographics 2007.
- Most work in portability and abstraction layers/languages are not ready (yet)…
- Can we accelerate our visualization software with something that is based on "proven" technology and portable across different architectures?
  - Data parallel libraries
    - NVidia Thrust library

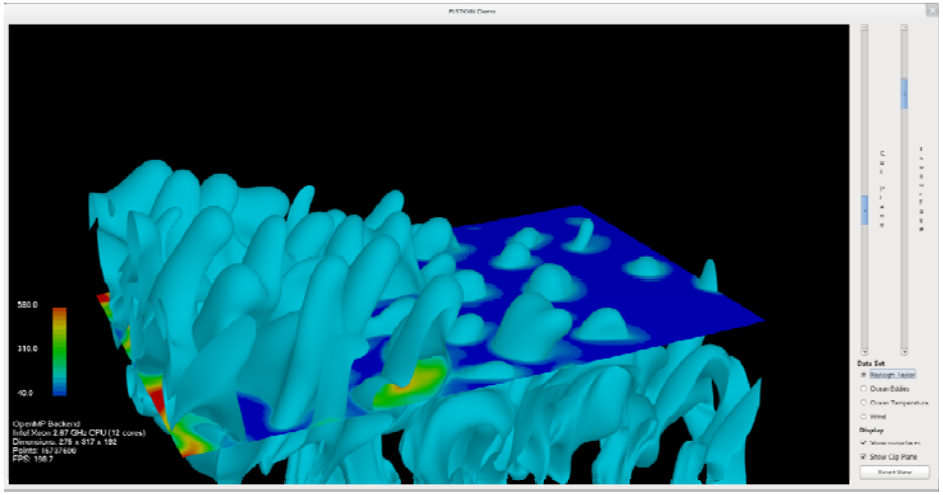
Brief Introduction to Data-Parallel Programming and Thrust

- What is data parallelism?
  - When independent processors performs the same task on different pieces of data
  - Due to the massive data sizes we expect to be simulating we expect data parallelism to be a good way to exploit parallelism on current and next generation architectures
  - "The data parallel bible" - Blelloch, "Vector Models for Data Parallel Computing"
- What is Thrust?
  - Thrust is a NVidia C++ template library for CUDA. It can also target OpenMP and we are creating new backends to target other architectures
  - Thrust allows you to program using an interface similar the C++ Standard Template Library (STL)
  - Most of the STL algorithms in Thrust are data parallel

Videos of PISTON in Action







Brief Introduction to Data-Parallel Programming and Thrust

- Why use Thrust instead of CUDA?
  - Thrust offers a data parallel abstraction. We believe code written in this abstraction will be portable to future systems.
  - Specifically, in this talk we will show the same algorithm written in Thrust running on NVidia GPUs and multi-core CPUs.
- What data structures does Thrust provide?
  - Currently Thrust provides thrust::host_vector and thrust::device_vector, which are analogous to std::vector in the STL and reside in the host/device memory.
  - These vector data structures simplify memory management and transferring data between the host and device.

Brief Introduction to Data-Parallel Programming and Thrust

What algorithms does Thrust provide?

- Sorts
- Transforms
- Reductions
- Scans
- Binary searches
- Stream compactions
- Scatters / gathers

Challenge: Write operators in terms of these primitives only

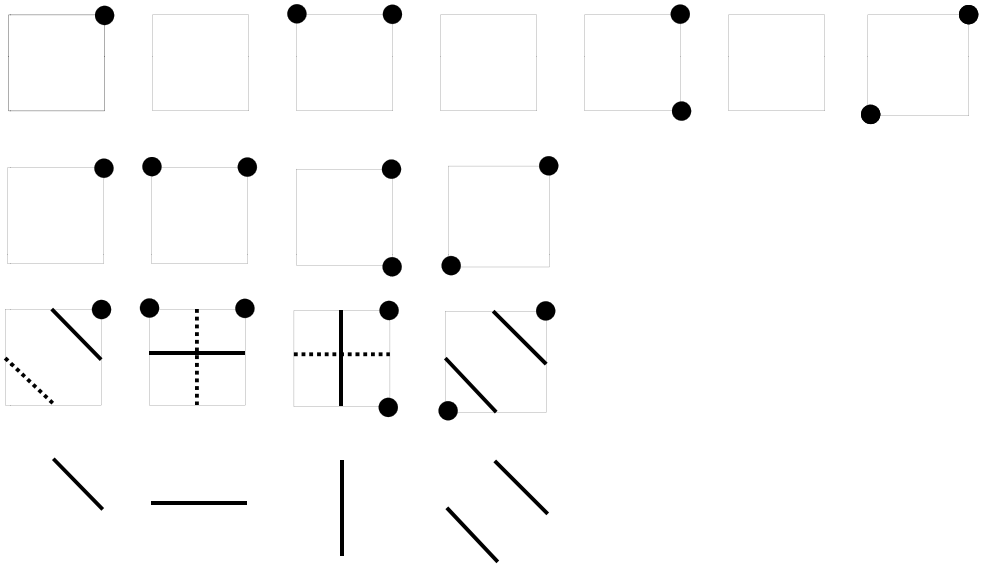
Reward: Efficient, portable code

```
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
```

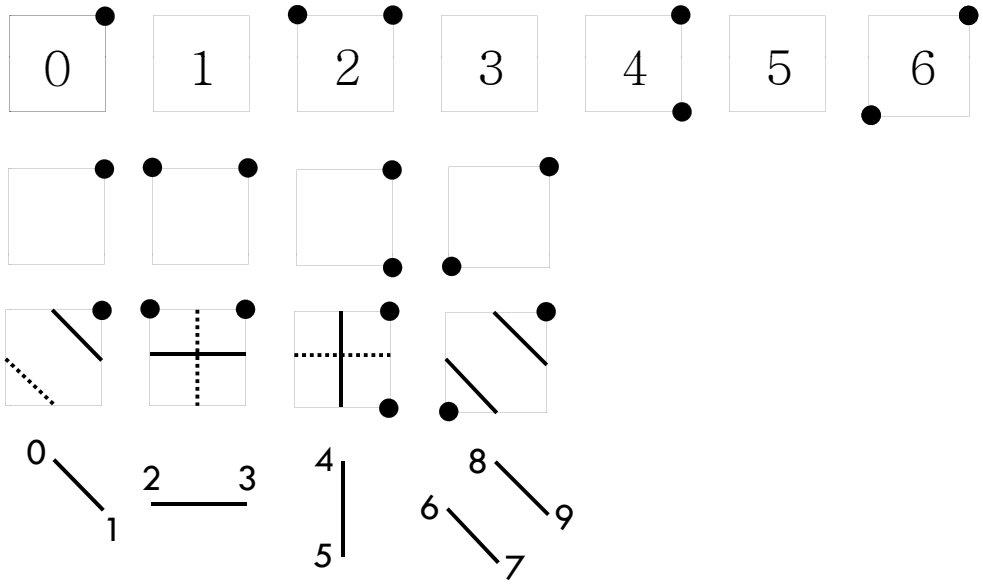
Isosurface with Marching Cube – the Naive Way

- Classify all cells by *transform*
- Use *copy_if* to compact valid cells.
- For each valid cell, generate same number of geometries with flags.
- Use *copy_if* to do stream compaction on vertices.
- This approach is too slow, more than 50% of time was spent moving huge amount of data in global memory.
- Can we avoid calling *copy_if* and eliminate global memory movement?



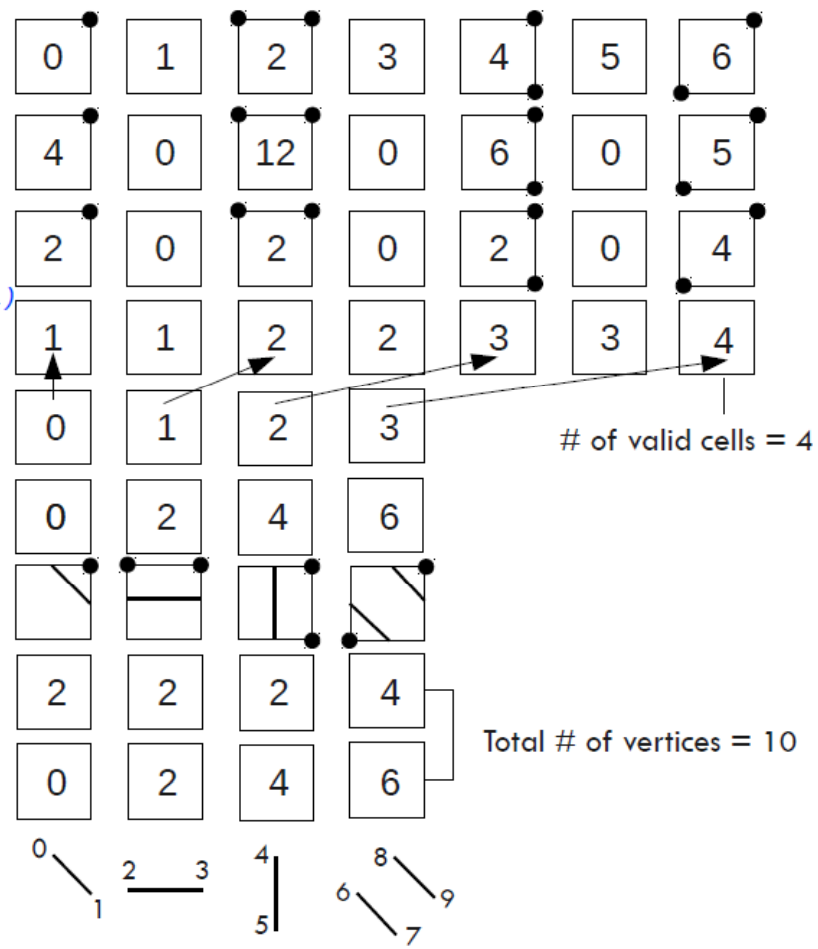
Isosurface with Marching Cube – Optimization

- Inspired by HistoPyramid
- The filter is essentially a mapping from input cell id to output vertex id
- Is there a "reverse" mapping?
- If there is a reverse mapping, the filter can be very "lazy"
- Given an output vertex id, we *only* apply operations on the cell that would generate the vertex
- Actually for a range of output vertex ids



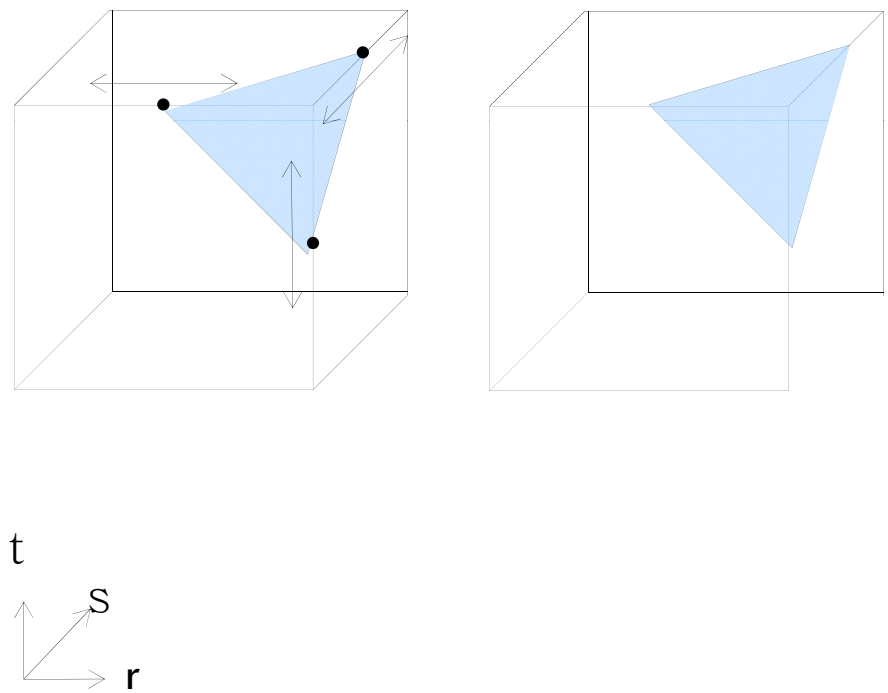
Isosurface with Marching Cubes Algorithm

1. input
   transform(classify_cell)
2. caseNums
3. numVertices
   transform_inclusive_scan(is_valid_cell)
4. validCellEnum
5. CountingIterator
   upper_bound
6. validCellIndices
   make_permutation_iterator
7. numVerticesCompacted
   exclusive_scan
8. numVerticesEnum
   for_each(isosurface_functor)
9. outputVertices



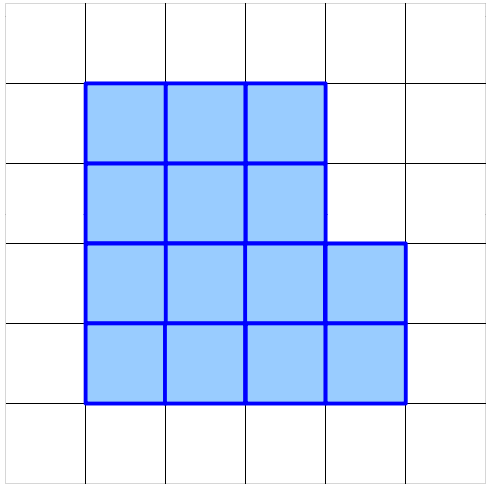
Cut Surfaces

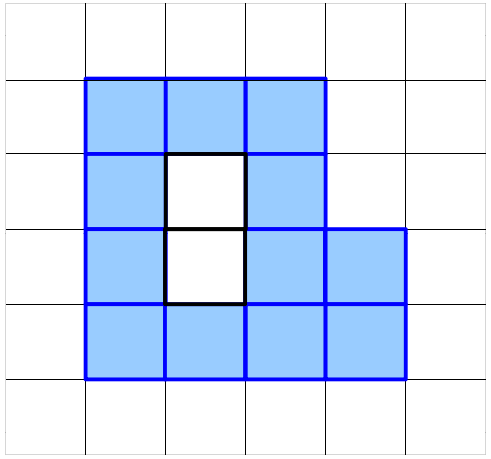
- All the vertices generated by marching cube are on the cell edges.
- They have only one degree of freedom, not three.
- 1D interpolation only, no need to do trilinear interpolation on scalar field.
- Two scalar fields, one for generating geometry (cut surface) the other for scalar interpolation.
- Less than 10 LOC change, negligible performance impact to isosurface.



Threshold

- Again, very similar to marching cube
  - Classify cells, stream compact valid cells and generate geometries for valid cells.
  - Optimization: what does the "inside" of a brick look like? Do we even care?
- Additional passes of cell classification and stream compaction to remove "interior cells"





Additional Operators

- Blelloch's "Vector Models for Data-Parallel Computing"

Data Structures
- Graphs: Neighbor reducing, distributing excess across edges
- Trees: Leaffix and rootfix operations, tree manipulations
- Multidimensional arrays

Computational Geometry
- Generalized binary search
- k-D tree
- Closest pair
- Quickhull
- Merge Hull

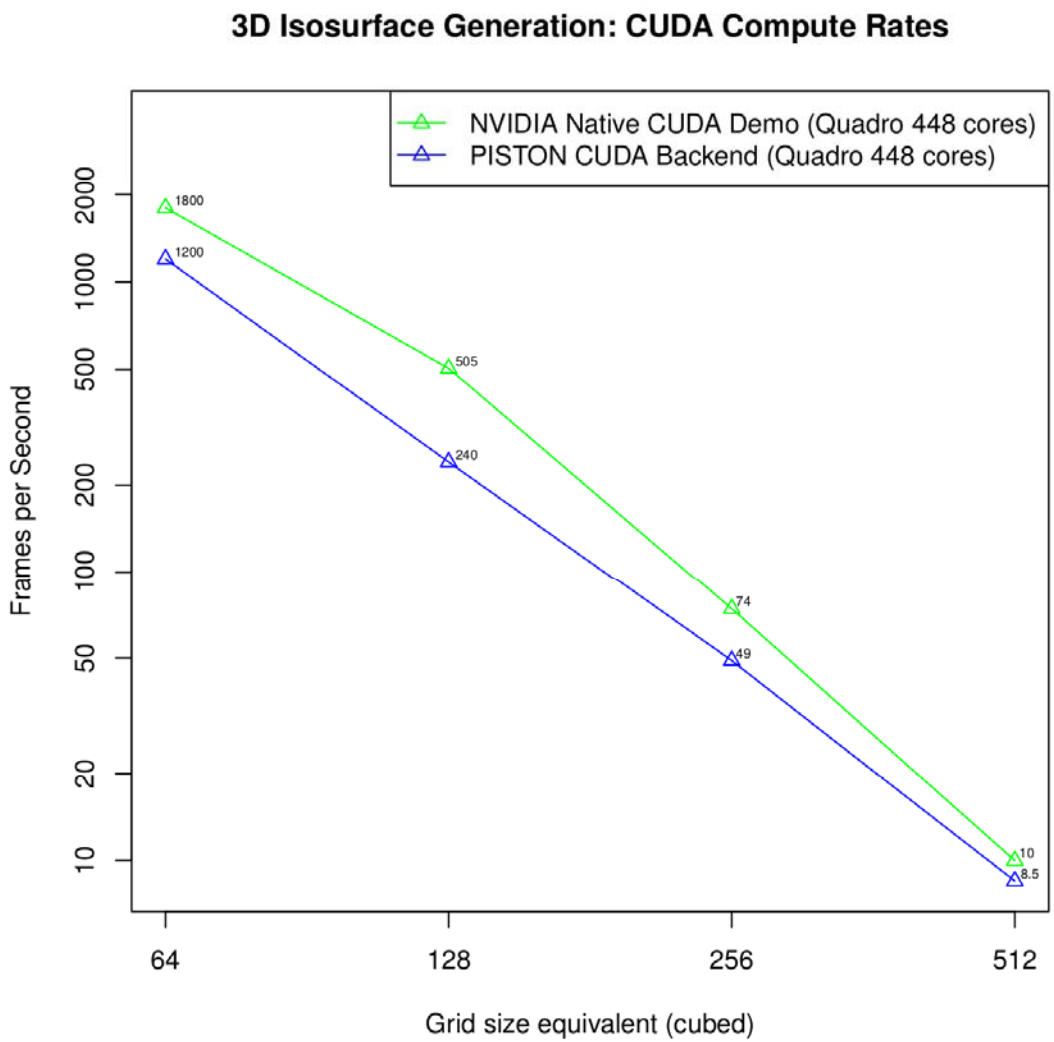
Graph Algorithms
- Minimum spanning tree
- Maximum flow
- Maximal independent set

Numerical Algorithms
- Matrix-vector multiplication
- Linear-systems solver
- Simplex
- Outer product
- Sparse-matrix multiplication

- Our on-going work: glyphs; rendering (rasterizing version and ray-casting version with k-D Tree); statistics with `reduce` primitive
- At least a naïve algorithm usually possible using the flexibility of `transform` and `for_each` primitives with user-defined functors; efficient global communication usually requires use of scans
- Efficient algorithms will still require clever design, but will be beneficial across platforms

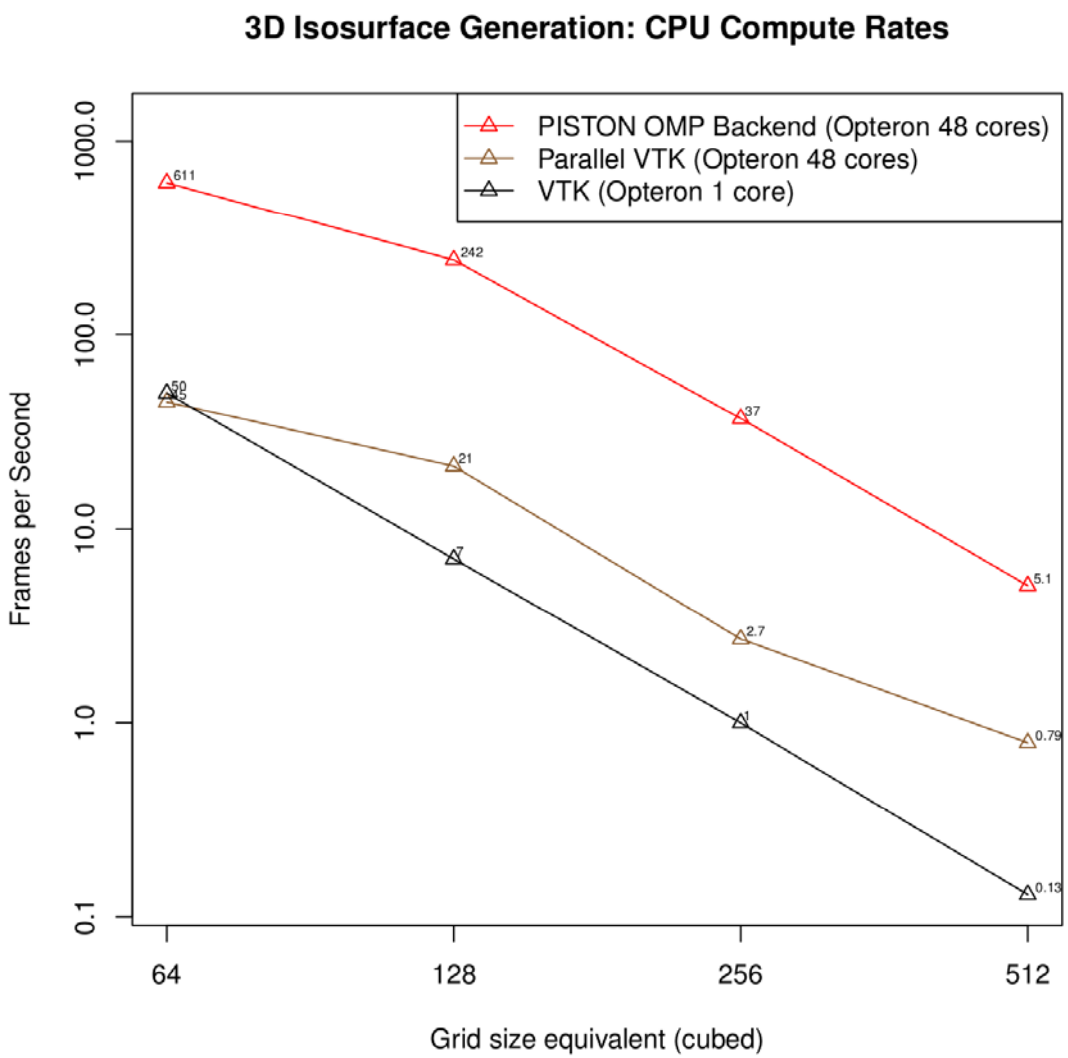
PISTON CUDA Backend Performance

- Limited performance degradation relative to native CUDA optimized code
- PISTON
  - Limited use of shared/texture memory due to portability
- NVIDIA CUDA Demo
  - Works only with data set with power of 2 per dimension, allowing use of shift instead of integer division
  - Memory inefficient; runs out of texture/global memory when data size is larger than 512^3



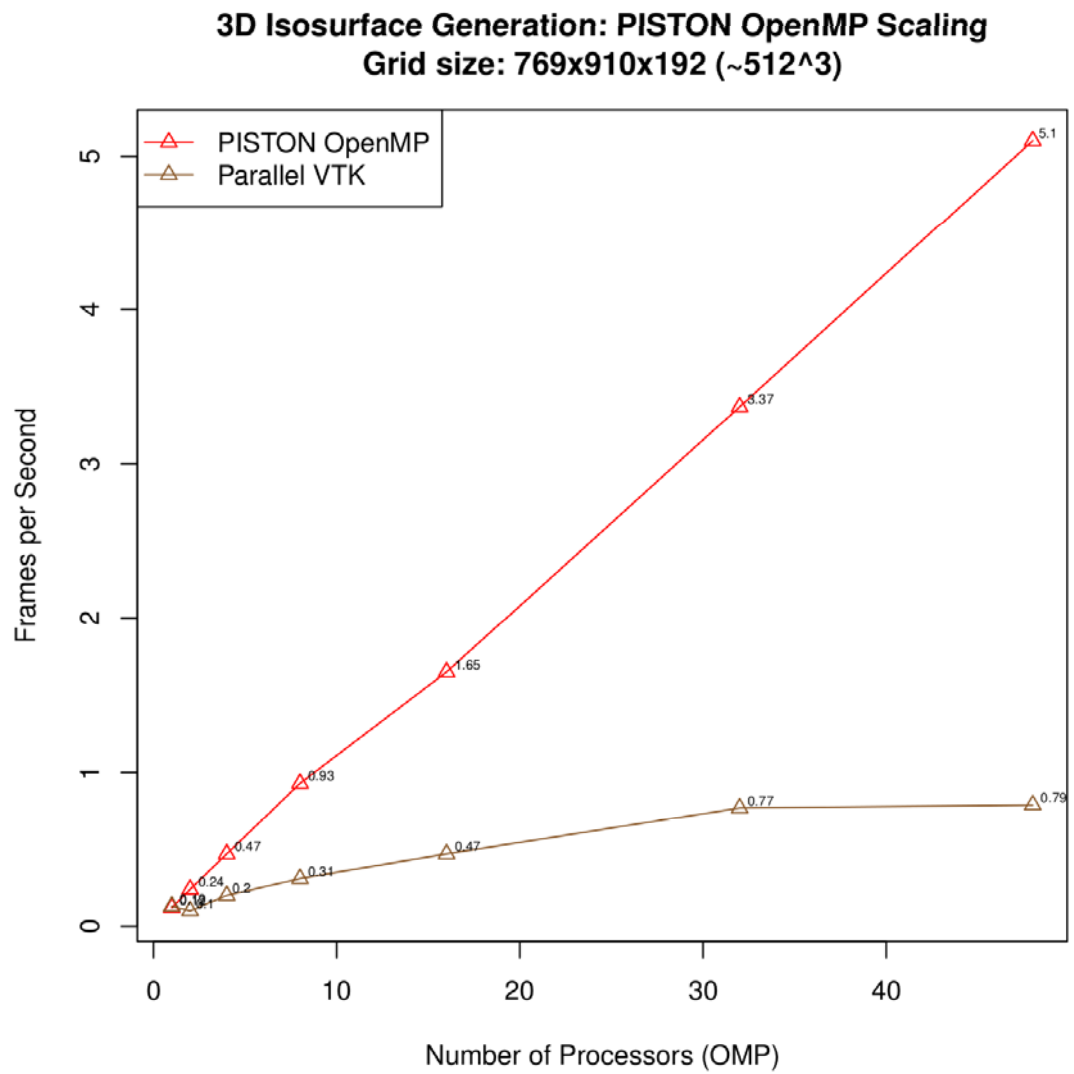
PISTON OpenMP Backend Performance

- Compile time #define/-D switches between backends
- Wrote our own parallel scan implementation for Thrust OpenMP backend
- Significantly better performance than both single process and parallel VTK



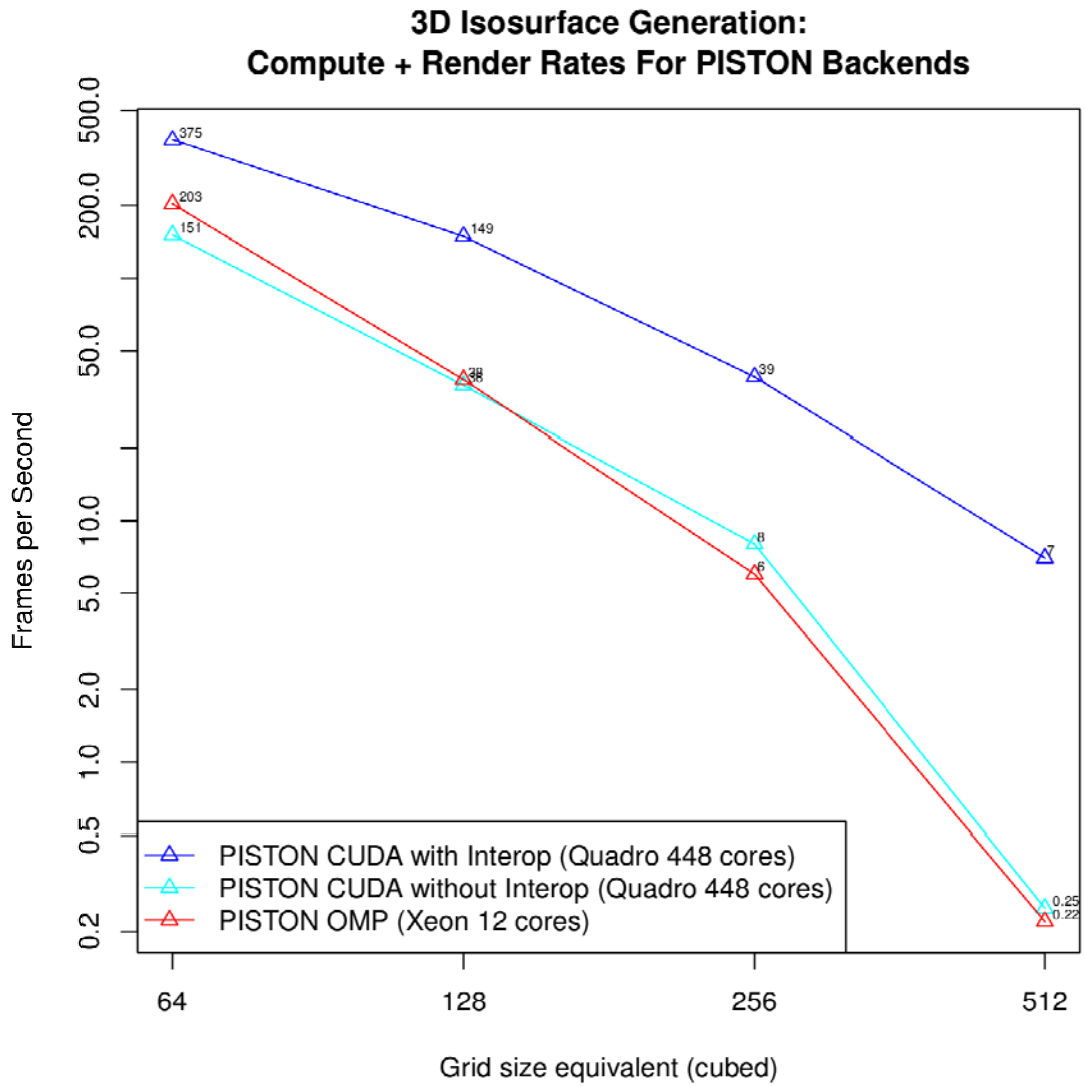
PISTON OpenMP Scaling Performance

- Significantly better scalability in term of # of cores than parallel VTK



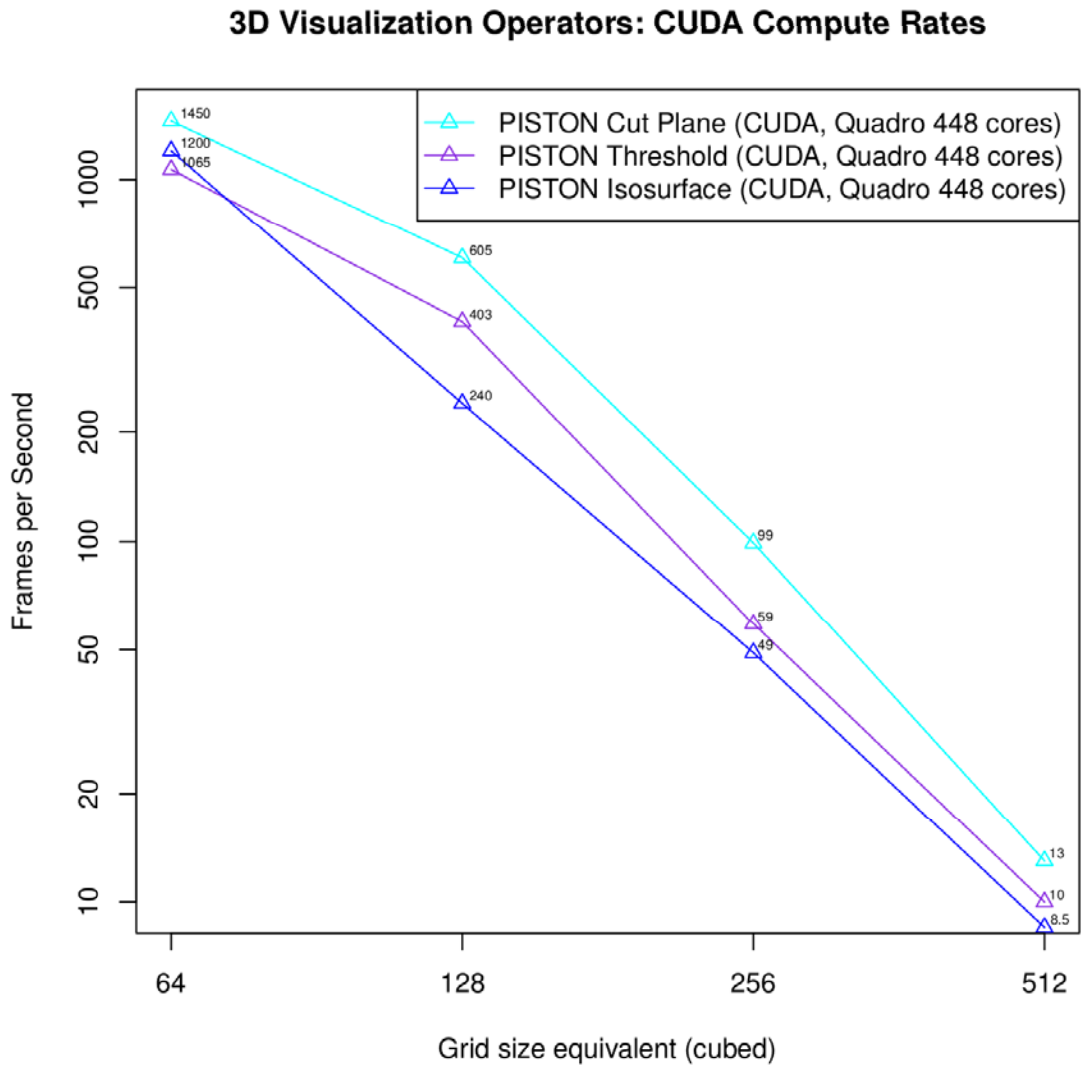
PISTON Compute and Render Results

- Compute and render results
  - CUDA and OpenMP backends
- CUDA/OpenGL interop
  - Platform specific, non-portable
  - Output geometries directly into OpenGL VBO
  - Avoid round trip between device and host memory movement
  - Vastly improves rendering performance and reduces memory footprint



PISTON Visualization Operators

- Three fundamental visualization operations
- All based on the same basic data-parallelism
- Very similar performance characteristics
  - Cut plane is the fastest since it generates 2D planes
  - Threshold comes next because there is no interpolation for scalar nor position
  - Isosurface is actually the most complicated operator

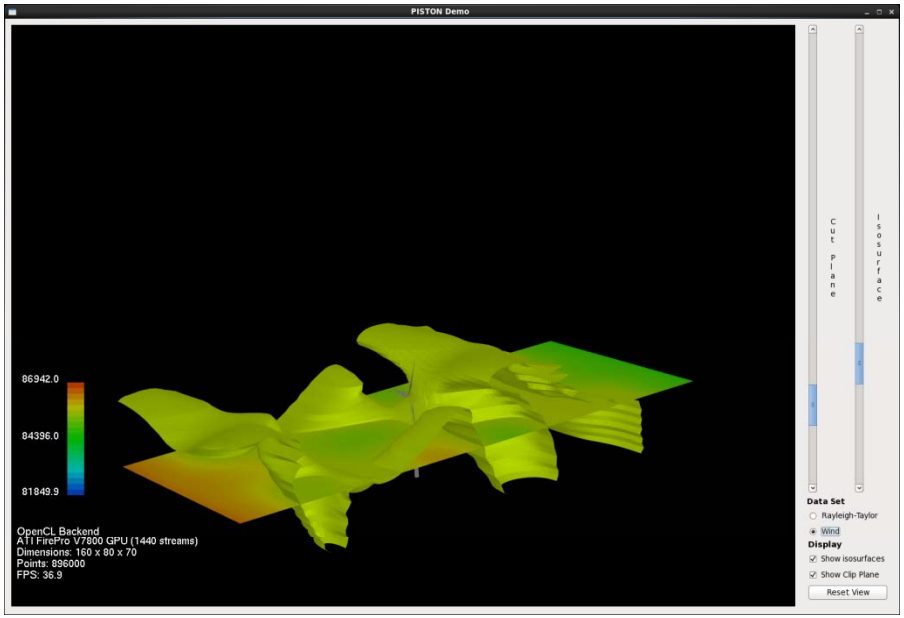


OpenCL Backend

- Motivation: Support for compiling visualization operators for a wide variety of additional GPU and CPU architectures
- Challenges
  - OpenCL is not built into Thrust
  - OpenCL is based on C99, making support for C++ features difficult
  - OpenCL compiles kernels from strings at run-time rather than from source files
- Current Approach
  - 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
  - Our Thrust-like backend uses run-time type information to handle simple templating and functor calls, substituting for key words in string
  - Kernel source only needs to be compiled once for each time it appears in code

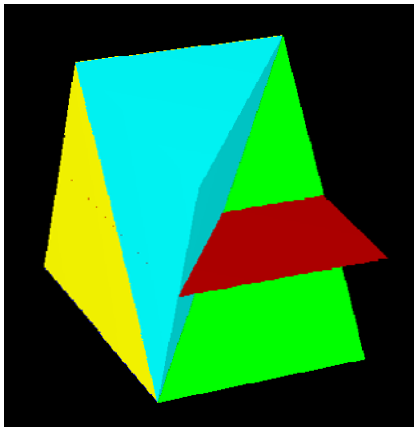
OpenCL Backend Results

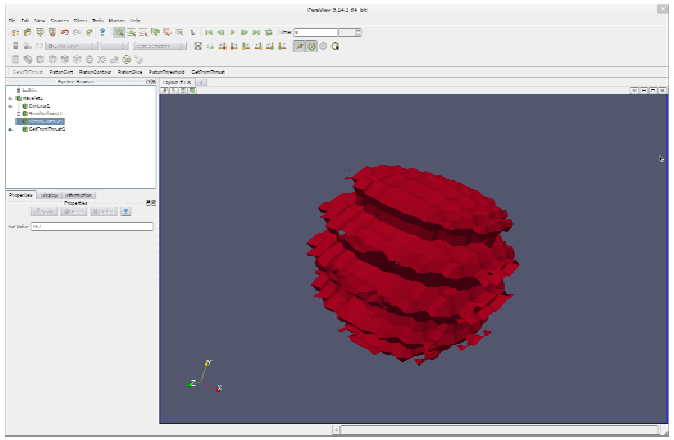
- Preliminary Results
  - Successfully implemented isosurface and cut plane operators in OpenCL with code *almost* identical to that used for the Thrust-based CUDA and OpenMP backends
  - With interop on AMD FirePro V7800 (1440 streams), we can run at about 6 fps for 256^3 data set (2 fps without interop)

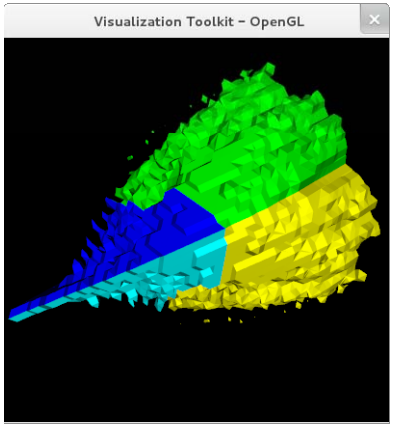


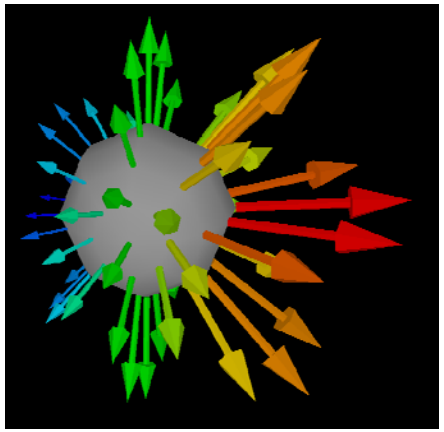
On-going and Future Work

- Marching Tetrahedra: a first step towards support for unstructured grids
- Integration with ParaView
- Multi-node parallelism with VTK/ParaView's MPI constructs
- More operators, more backends



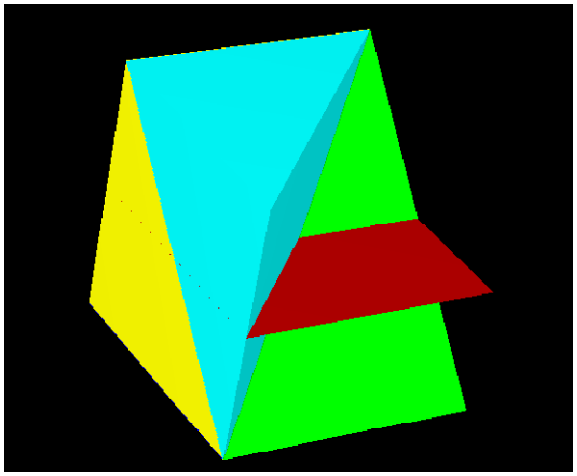


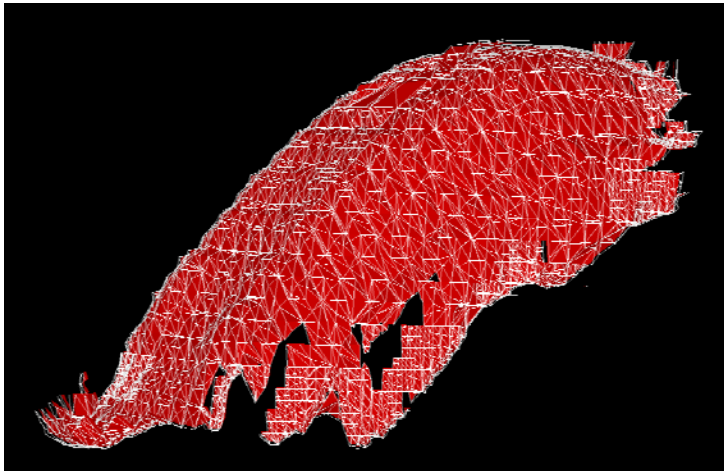




Marching Tetrahedra

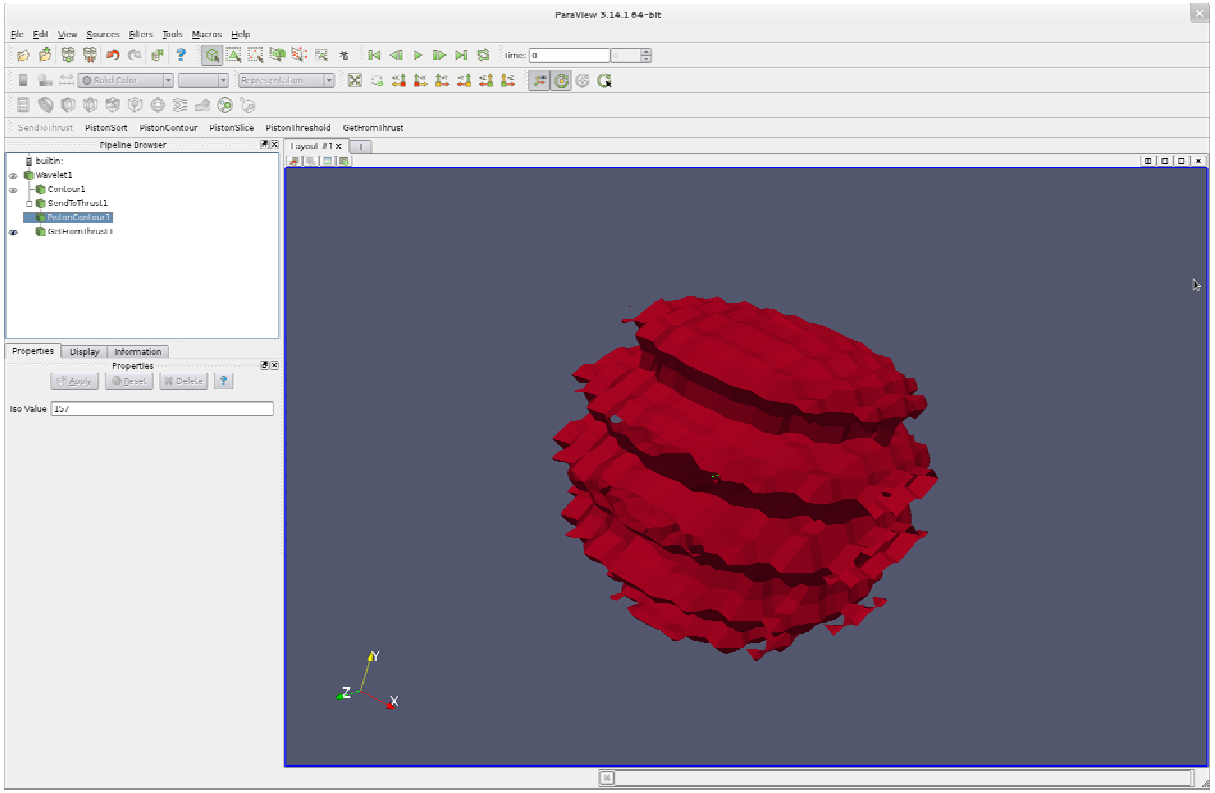
- Current procedure
  - Tetrahedralize uniform grid or unstructured grid (e.g., AMR mesh)
  - Generate isosurface geometry based on look-up table for tetrahedral cells
- Next step: Develop PISTON operator to tetrahedralize grids, and/or to compute isosurface directly on AMR grid
- Polytypic algorithm design





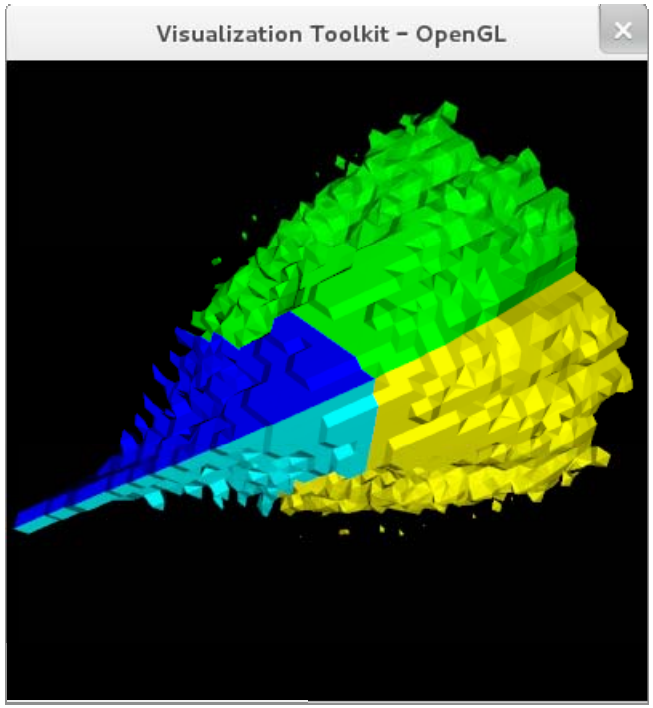
Integration with ParaView

- Filters that use PISTON data types and algorithms integrated into ParaView prototype
- Utility filters interconvert between standard VTK data format and PISTON data format (thrust device vectors)
- Can chain PISTON filters; soon will support interop for on-card rendering



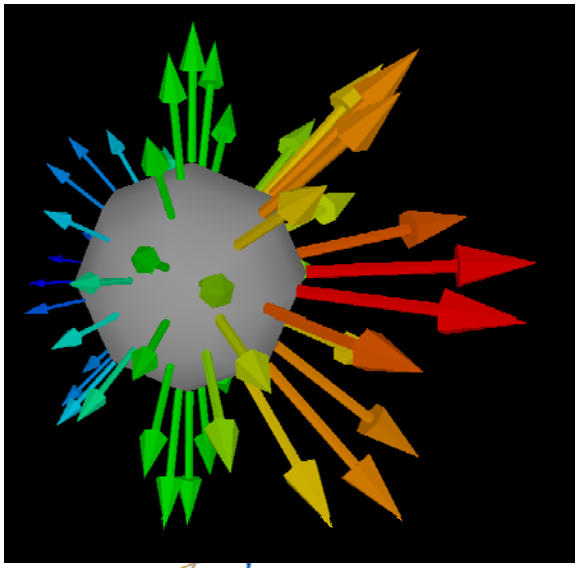
Inter-node Parallelism

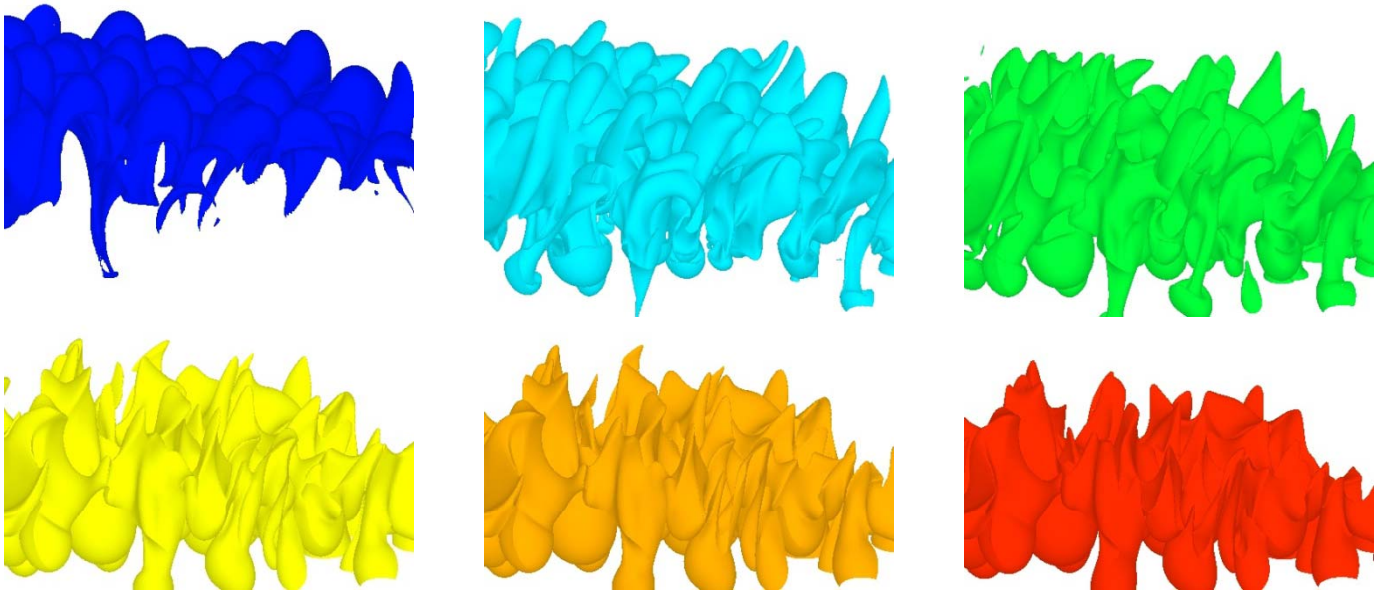
- Domain partitioned by VTK's MPI libraries
- Each node then uses PISTON filters to compute results for its portion of the domain
- Results combined by VTK's compositors



Additional Operators

- Current prototypes
  - Glyphs
  - Renderer – rasterizing and ray-casting versions (using K-D Tree), allowing the generation of images on systems without OpenGL
  - Halo finder for cosmology simulations
- Ultimately want to support a large subset of ParaView filters, plus analysis operators





Recent Development – Curvilinear Coordinates

- Generic algorithm design
  - High-level algorithms are independent of coordinate systems
  - Implemented by multiple layers of coordinate transformations
  - Due to kernel fusion, very little performance impact

Open-Source Release

- Open-source release
  - Tarball: http://viz.lanl.gov/projects/PISTON.html
  - Repository: https://github.com/losalamos/PISTON

Acknowledgments and Resources

- 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

- For more information, see
  http://viz.lanl.gov/projects/PISTON.html

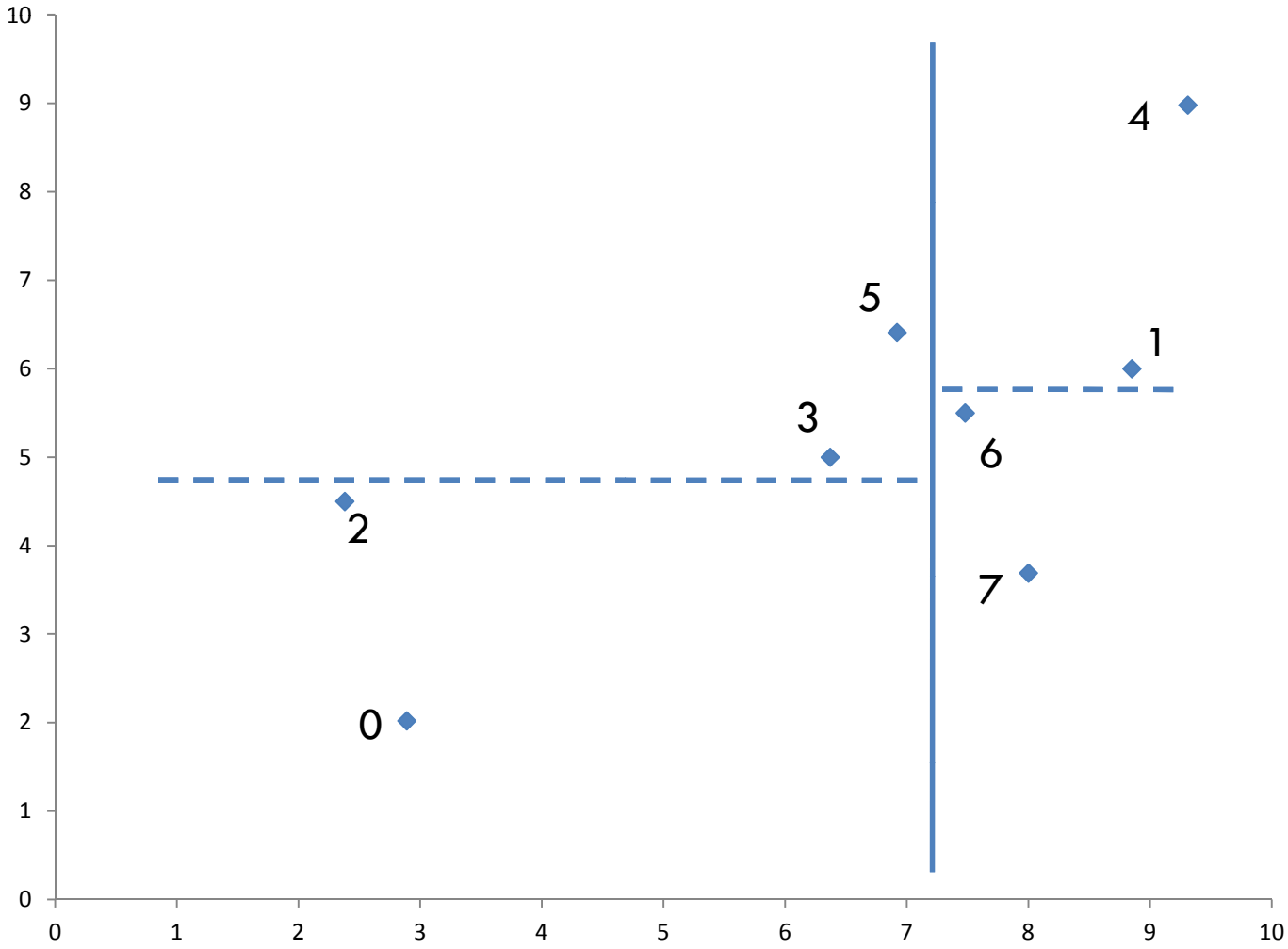
Tutorials

- Be a Thrust user
  - Thrust QuickStartGuide examples
- Be a PISTON user
  - tutorial1{OMP/GPU}: create a tangle field and apply the PISTON isosurface operator
  - demo{OMP/GPU}: load a VTK structured grid from a file and apply the PISTON isosurface, cut plane, or threshold operator
- Be a PISTON developer
  - tutorial2{OMP/GPU}: write a simple simulation (boid flocking) and a simple visualization operator (glyphs) using Thrust primitives, and chain them together (optionally using interop with CUDA)
- Be a PISTON algorithm designer
  - tutorial3{OMP/GPU}: use data parallel primitives to design a KD-tree algorithm

PISTON vs. Thrust

- Thrust provides:
  - An STL-like interface for memory management (host/device vectors) and data-parallel algorithms
  - Backend implementations of the data-parallel algorithms for CUDA, as well as lower-quality implementations for OpenMP and TBB
- PISTON provides:
  - A library of visualization and analysis operators implemented using Thrust
  - A data model for scalar fields (e.g., VTK structured grids; unstructured grids in-progress)
- PISTON enhances:
  - Non-CUDA backends
  - Interface to support distributed memory operations

KD-Tree



KD Tree: Overview

		Point Ids	X Ranks	Y Ranks
	computeGlobalRanks	0 1 2 3 4 5 6 7	1 6 0 2 7 3 4 5	0 5 2 3 7 6 4 1
Level 1	computeFlags	F T F F T F T T	**F T F F T F T T**	F T F F T F T T
Level 1	segmentedSplit	0 2 3 5 | 1 4 6 7	1 0 2 3 | 6 7 4 5	0 2 3 6 | 5 7 4 1
Level 1	0 0 0 0 1 1 1 1	F F F F | T T T T	F F F F | T T T T	F F F F | T T T T
Level 1	renumberRanks		1 0 2 3 | 2 3 0 1	0 1 2 3 | 2 3 1 0
Level 2	computeFlags	F F T T | T T F F	F F T T | T T F F	**F F T T | T T F F**
Level 2	segmentedSplit	0 2 | 3 5 | 6 7 | 1 4	1 0 | 2 3 | 0 1 | 2 3	0 1 | 2 3 | 1 0 | 2 3
Level 2	0 0 1 1 2 2 3 3	F F | T T | F F | T T	F F | T T | F F | T T	F F | T T | F F | T T
Level 2	renumberRanks	| | | |	1 0 | 0 1 | 0 1 | 0 1	0 1 | 0 1 | 1 0 | 0 1

KD Tree: computeGlobalRanks

A	Input coordinates	2.9	8.9	2.4	6.4	9.3	6.9	7.5	7.6
B	CountingIterator(0)	0	1	2	3	4	5	6	7
C	sort by key(A,B)	2.4	2.9	6.4	6.9	7.5	7.6	8.9	9.3
		2	0	3	5	6	7	1	4
D	scatter(B,C)	1	6	0	2	7	3	4	5

KD Tree: computeFlags

A	Input ranks	1 6 0 2 7 3 4 5
B	Input segmentIds	0 0 0 0 0 0 0 0
C	CountingIterator(1)	1 2 3 4 5 6 7 8
D	Reverse inclusive scan by key(B,C,max) // # elements in segment	8 8 8 8 8 8 8 8
E	transform(E[i]=D[i]/2) // # median index	4 4 4 4 4 4 4 4
F	transform(F[i]=A[i]>=E[i])	F T F F T F T T

KD Tree: segmentedSplit

A	Input pointIds	0 2 3 5 1 4 6 7
B	Input flags	F F T T T T F F
C	Input segmentIds	0 0 0 0 1 1 1 1
D	exclusive_scan_by_key(C,B) // total number of true flags preceding in segment	0 0 0 1 0 1 2 2
E	CountingIterator(0)	0 1 2 3 4 5 6 7
F	inclusive_scan_by_key(C,E,min) // total number of elements in previous segments	0 0 0 0 4 4 4 4
G	CountingIterator(1)	1 2 3 4 5 6 7 8
H	Reverse inclusive_scan_by_key(C,G,max) // index of last element in its segment (+1)	4 4 4 4 8 8 8 8
I	inclusive_scan_by_key(C,inverse(B)) // total number of false flags so far in segment	1 2 2 2 0 0 1 2
J	transform(J[i]=(if(B[i]) F[i]+I[H[i]-1]+D[i] else F[i]+I[i]-1))	0 1 2 3 6 7 4 5
K	scatter(A,J)	0 2 3 5 6 7 1 4

KD Tree: renumberRanks

A	Input ranks	0	2	3	6	5	7	4	1
B	Input flags	F	F	F	F	T	T	T	T
C	Input segmentIds	0	0	0	0	1	1	1	1
D	ConstantIterator(1)	1	1	1	1	1	1	1	1
E	exclusive_scan_by_key(C,D)	0	1	2	3	0	1	2	3
F	scatter(E,A)	0	3	1	2	2	0	3	1
G	scatter(B,A)	F	T	F	F	T	T	F	T
H	segmentedSplit(F,G)	0	1	2	3	3	2	0	1
I	CountingIterator(0)	0	1	2	3	4	5	6	7
J	inclusive scan by key(C,I,min) // total number of elements in previous segments	0	0	0	0	4	4	4	4
K	transform(H+J)	0	1	2	3	7	6	4	5
L	scatter(E,K)	0	1	2	3	2	3	1	0

KD Tree: renumberRanks (further segmented)

A	Input ranks	0 1 2 3 1 0 2 3
B	Input flags	F F T T F F T T
C	Input segmentIds	0 0 1 1 2 2 3 3
D	Input pre-split segmentIds	0 0 0 0 1 1 1 1
E	CountingIterator(0)	0 1 2 3 4 5 6 7
F	inclusive_scan_by_key(D,E,min)	0 0 0 0 4 4 4 4
G	transform(A+F)	0 1 2 3 5 4 6 7
H	ConstantIterator(1)	1 1 1 1 1 1 1 1
I	exclusive_scan_by_key(C,H)	0 1 0 1 0 1 0 1
J	scatter(I,G)	0 1 0 1 1 0 0 1
K	scatter(B,G)	F F T T F F T T
L	segmentedSplit(J,K,C)	0 1 0 1 1 0 0 1
M	inclusive_scan_by_key(C,E,min)	0 0 2 2 4 4 6 6
N	transform(L+M)	0 1 2 3 5 4 6 7
O	scatter(I,N)	0 1 0 1 1 0 0 1

OpenCL Backend

- Motivation: Support for compiling visualization operators for a wide variety of additional GPU and CPU architectures
- Challenges
  - OpenCL is not built into Thrust, requiring us to create a new backend from scratch
  - OpenCL is based on C99, making it difficult to support C++ features (templates, functors, iterators, etc.) integral to Thrust
  - OpenCL compiles kernels from strings in the host language at run-time rather than directly compiling C code embedded in the host language at compile-time

OpenCL Backend: Prototype Design

- PISTON provides a Thrust-like library of include files ("lathrust") that implement host/device vectors that can read and write data to the device using OpenCL, and OpenCL-native code for basic data-parallel primitives (scan, transform, etc.) in .cl files, with keywords as placeholders for calls to user-defined functions
- User writes an operator in C++, making calls to lathrust wrappers for the data-parallel primitives, optionally passing user-defined functors as arguments
- PISTON pre-processor extracts operators (which must be C99-compliant) from user-defined functors and outputs them to .cl files as functions named according to the class name of their functor
- At run-time, PISTON backend wrapper functions create a string by concatenating the contents of the data-parallel primitive .cl file and the pre-processor-generated .cl file, replace key words for user-defined function calls with the appropriate function name (based on the run-time type information of the functor argument) and key words for data types with actual data types (based on the templated instantiation data types), and make calls to OpenCL to build and execute the kernel

OpenCL Backend: Simple Example

util_math.cl

```
...
__inline__ float lerp(float a, float b, float t)
{
  return a + t*(b-a);
}
```

transform.cl

```
__kernel void transform(__global T_TYPE* input, __global T_TYPE* output)
{
    unsigned int i = get_global_id(0);
    output[i] = USER_OPERATOR(input[i]);
}
```

myOperator.inl

```
template <typename InputIterator>
class myOperator
{
public:
    typedef typename std::iterator_traits<InputIterator>::value_type value_type;
    InputIterator input, temp, output;  int n;
    myOperator(InputIterator input, int n) : input(input), n(n) { }

    void operator()()
    {
        lathrust::transform(input.begin(), temp.begin(), n, new doubleIt());

        lathrust::transform(temp.begin(), output.begin(), n, new tripleIt());
    }

    struct doubleIt : public lathrust::unary_function
    {
        doubleIt() { }

        value_type operator()(value_type value)
        {
            return 2*value;
        }
    };

    struct tripleIt : public lathrust::unary_function
    {
        tripleIt() { }

        value_type operator()(value_type value)
        {
            return 3*value;
        }
    };
};
```

Pre-processor

user.cl

```
value_type doubleIt()(value_type value)
{
    return 2*value;
}

value_type tripleIt()(value_type value)
{
    return 3*value;
}
```

lathrust backend

kernel_source

```
“
  ...
__inline__ float lerp(float a, float b, float t)
{
  return a + t*(b-a);
}

int doubleIt()(int value)
{
    return 2*value;
}

__kernel void transform(__global int* input, __global int* output)
{
    unsigned int i = get_global_id(0);
    output[i] = doubleIt(input[i]);
}
”
```

Compiled Kernel

clCreateProgramWithSource; clBuildProgram

lathrust backend

Compiled Kernel

kernel_source

```
“
  ...
__inline__ float lerp(float a, float b, float t)
{
  return a + t*(b-a);
}

int tripleIt()(int value)
{
    return 3*value;
}

__kernel void transform(__global int* input, __global int* output)
{
    unsigned int i = get_global_id(0);
    output[i] = tripleIt(input[i]);
}
”
```

OpenCL Backend: Advanced Topics

- The PISTON backend can provide the OpenCL function generated from the user-defined functor with access to the functor data by packaging the functor's data fields in a struct and passing it to the OpenCL function
- Large functor data fields are passed separately, and the backend replaces keywords in the OpenCL data-parallel primitive implementations to extend the set of parameters passed to the kernel and on to the user-defined function
- Permutation iterators are similarly implemented by passing an additional field to the kernel and replacing keywords in the OpenCL code with indexing into the permutation field
- The kernel source code is the same between executions of the same line of host code (even though the data it is sent may differ), so kernel compilation can be performance once at the beginning for each call of an lathrust wrapper, and the compiled kernel reused whenever that call is executed

OpenCL Backend: Functor Example

util_math.cl

```
...
__inline__ float lerp(float a, float b, float t)
{
  return a + t*(b-a);
}
```

transform.cl

```
__kernel void transform(__global T_TYPE* input, __global T_TYPE* output,
                        __global void* vstate FIELD_PARAMETERS)
{
    unsigned int i = get_global_id(0);
    output[i] = USER_OPERATOR(i, input[i], vstate PASS_FIELDS);
}
```

myOperator.inl

```
template <typename InputIterator>
class myOperator
{
public:
    typedef typename std::iterator_traits<InputIterator>::value_type value_type;
    InputIterator input, InputIterator offsets, output;  int n;  value_type scaleFactor;
    myOperator(InputIterator input, InputIterator offsets, value_type scaleFactor, int n) :
                    input(input), offsets(offsets), scaleFactor(scaleFactor), n(n) { }

    void operator()()
    {
      lathrust::transform(input.begin(), output.begin(), n, new offsetAndScale(scaleFactor, offsets));
    }

    struct offsetAndScale : public lathrust::unary_function
    {
      typedef struct offsetAndScaleData : public lathrust::functorData
      {
          value_type scaleFactor;
      } OffsetAndScaleData;
      virtual int getStateSize() { return (sizeof(OffsetAndScaleData)); }

      offsetAndScale(value_type scaleFactor, InputIterator offsets)
      {
         OffsetAndScaleData* dstate = new OffsetAndScaleData;
         dstate->scaleFactor = scaleFactor;
         state = dstate;
         addField(*offsets);
      }

      value_type operator()(int index, value_type value, OffsetAndScaleData* state,
                            value_type* offsets)
      {
         return ((state->scaleFactor)*(value + offsets[index]));
      }
    };
};
```

lathrust backend

kernel_source

```
“
  ...
__inline__ float lerp(float a, float b, float t)
{
  return a + t*(b-a);
}

int offsetAndScale()(int index, int value, OffsetAndScaleData* state, int* offsets)
{
    return ((state->scaleFactor)*(value + offsets[index]));
}

__kernel void transform(__global int* input, __global int* output, __global void* vstate,
                        __global void* field1)
{
    unsigned int i = get_global_id(0);
    output[i] = offsetAndScale(i, input[i], vstate, field1);
}
”
```

clCreateProgramWithSource; clBuildProgram

Compiled Kernel

Pre-processor

user.cl

```
value_type offsetAndScale()(int index, value_type value, OffsetAndScaleData* state,
                            value_type* offsets)
{
    return ((state->scaleFactor)*(value + offsets[index]));
}
```