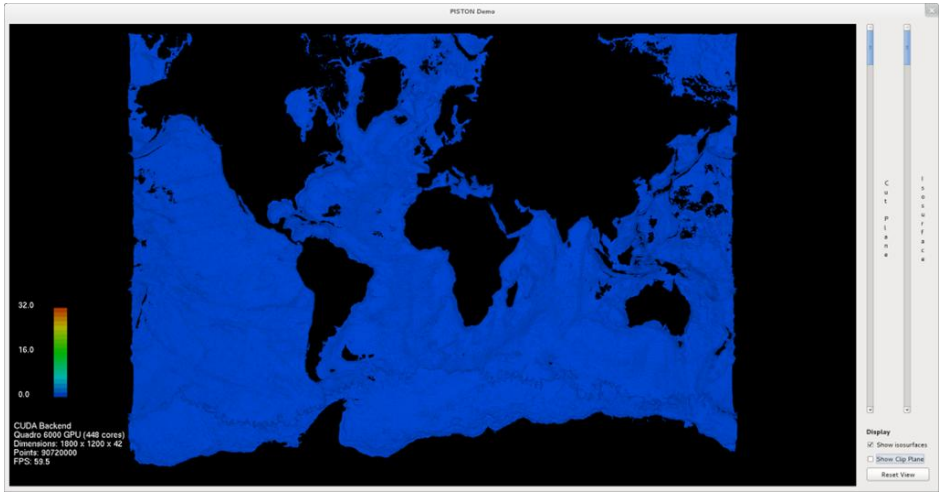
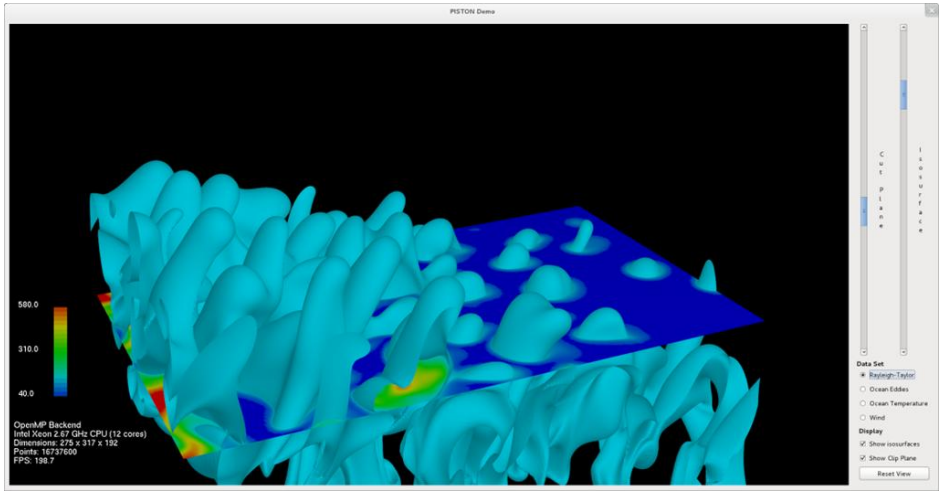
PISTON : A portable cross-platform framework for data-parallel visualization operators

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





Los Alamos NATIONAL LABORATORY EST.1943
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
- Future work
  - New targets – OpenCL/AMD, OpenMP/BlueGene

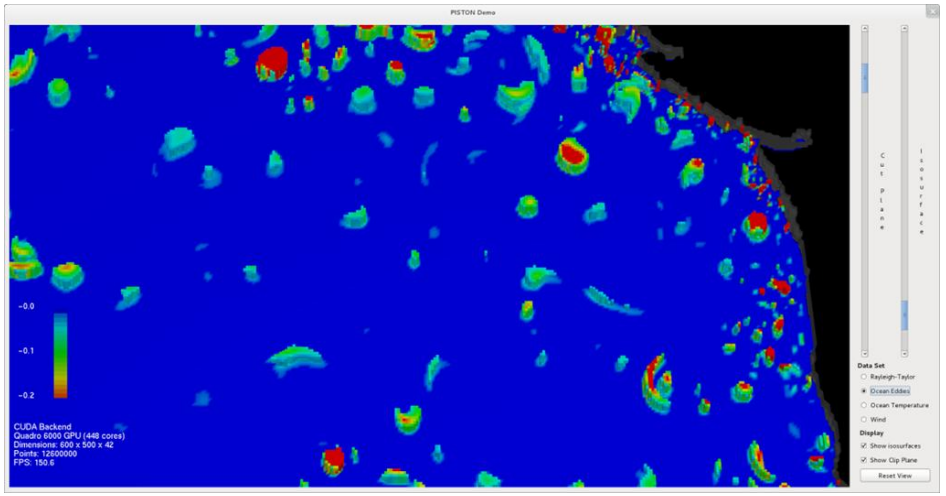
Motivation / Related Work

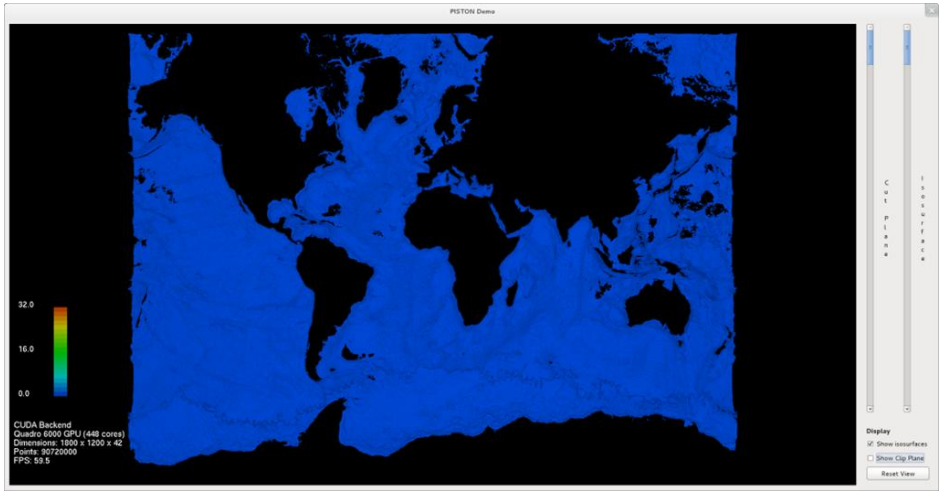
- Current production visualization software does not take full advantage of acceleration hardware and/or multi-core architecture
  - Vtk, ParaView, Visit
- Research on accelerating visualization operations are mostly hardware-specific; few were integrated in visualization software
  - CUDA SDK demo
  - Dyken, Ziegler, "High-speed Marching Cubes using Histogram Pyramids", Eurographics 2007.
- Most work in portability and abstraction layers/languages are not ready (yet)...
  - Scout, DAX, Liszt
- 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

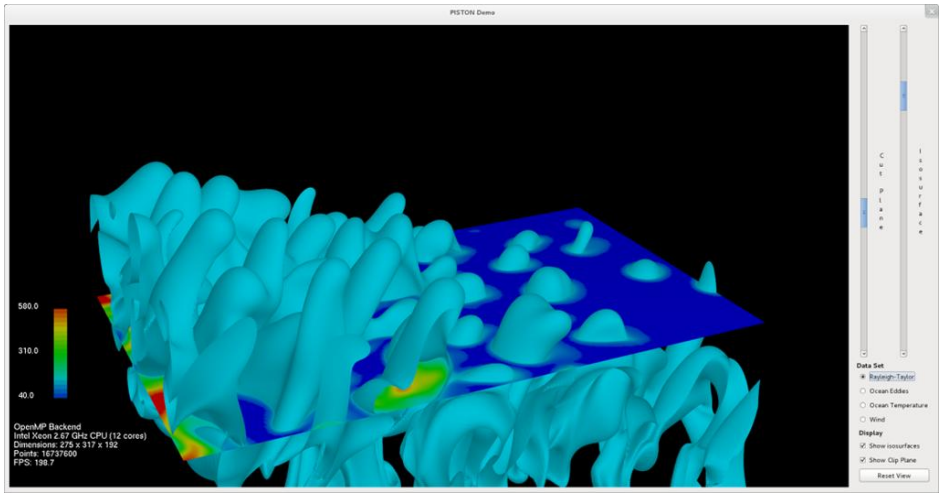
Brief Introduction to Data-Parallel Programming and Thrust

- 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







Los Alamos NATIONAL LABORATORY EST. 1943

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

LA-UR-11-11980

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?

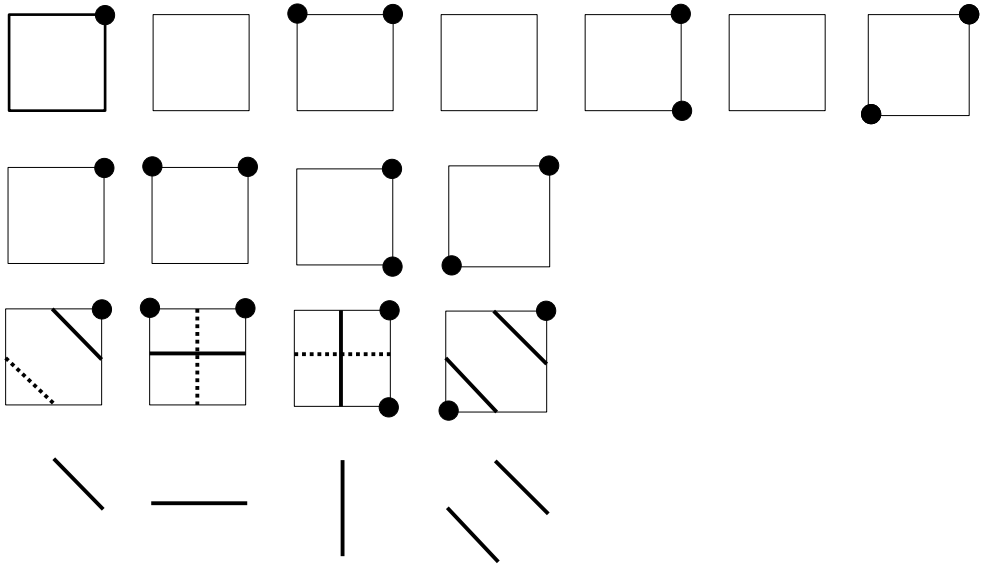
- sorting: thrust::sort and thrust::sort_by_key
  - 4 5 6 8 7 2 1 3 :sort: 1 2 3 4 5 6 7 8
- transformations: thrust::transform
  - Any unary and binary operation
  - 4 5 6 8 7 2 1 3 :transform plus 1: 5 6 7 9 8 3 2 4
- reductions: thrust::reduce and thrust::transform_reduce
  - 4 5 6 8 7 2 1 3 :sum reduce: 36
- scans: thrust::inclusive_scan, thrust::exclusive_scan, thrust::transform_inclusive_scan, etc.
  - 4 5 6 7 8 2 1 3 :sum scan: 4 9 15 22 30 32 33 36
- Binary search, stream compaction, scatter/gather, etc.
- Work with user defined data types and operators/functors too

Challenge: Write operators in terms of these primitives only

Reward: Efficient, portable code

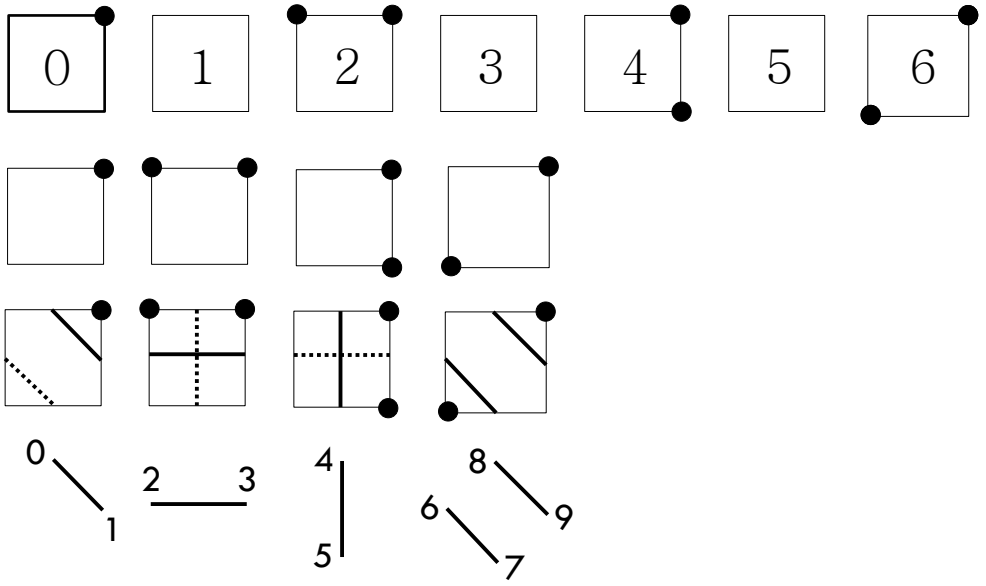
Isosurface with Marching Cube – the Naive Way

- 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?



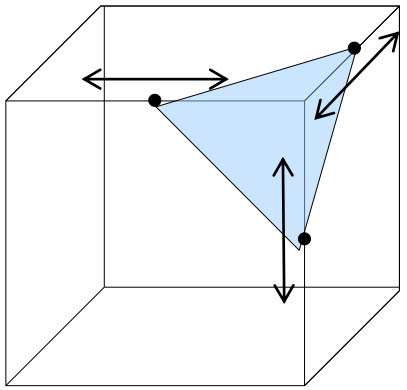
Isosurface with Marching Cube – Optimization

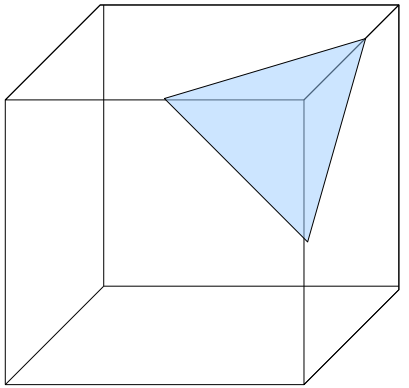
- Inspired by HistoPyramid
- The filter is essentially a mapping from input cell id to output vertex id
- Is there a "reverse" mapping?
- If there is a reverse mapping, the filter can be very "lazy"
- Given an output vertex id, we *only* apply operations on the cell that would generate the vertex
- Actually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

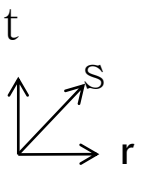


Cut Surfaces

- 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.

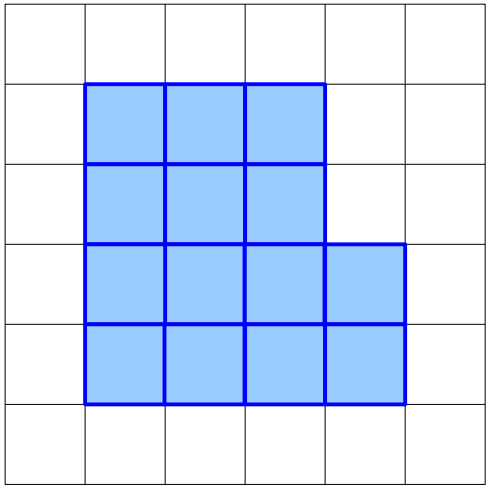


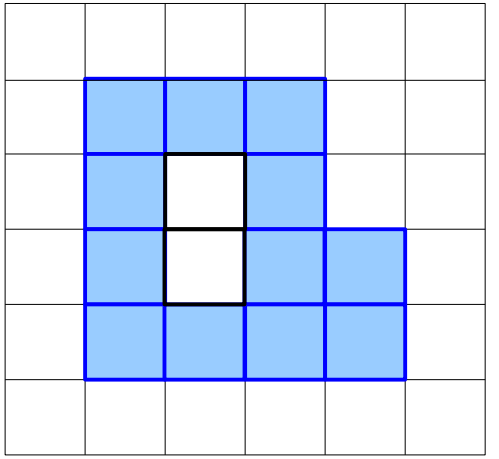




Threshold

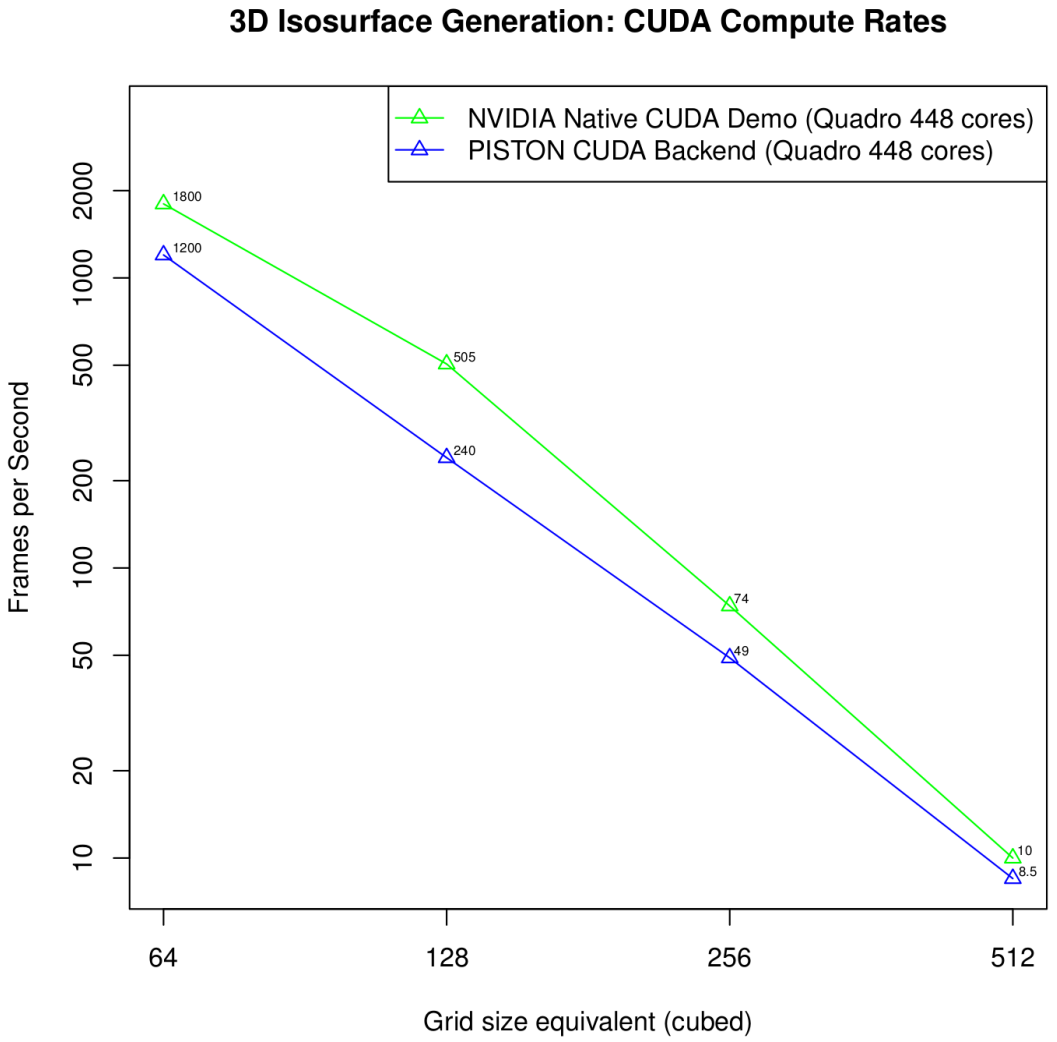
- Again, very similar to marching cube
  - Classify cells, stream compact valid cells and generate geometries for valid cells.
  - Optimization: what does the "inside" of a brick look like? Do we even care?
- Additional passes of cell classification and stream compaction to remove "interior cells"





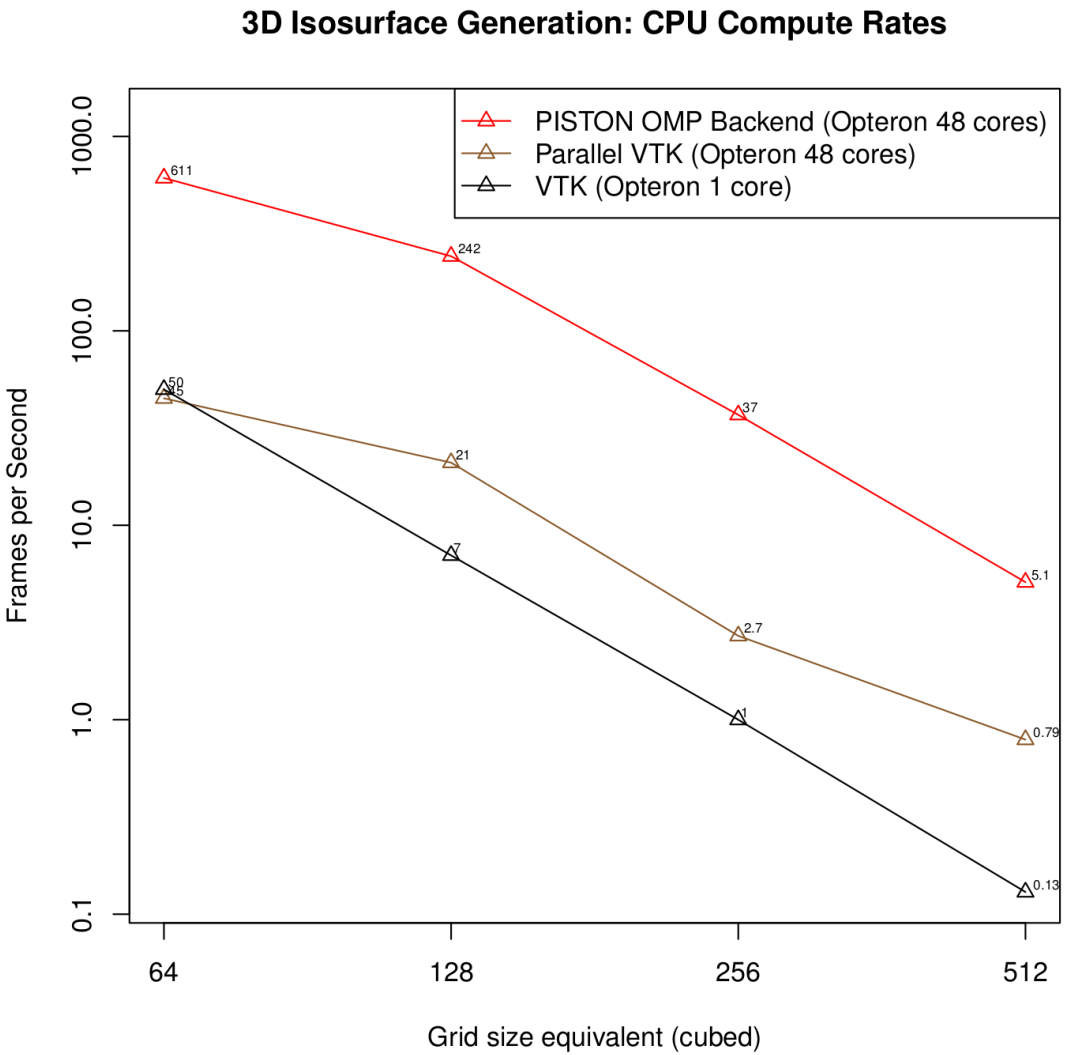
PISTON CUDA Backend Performance

- 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



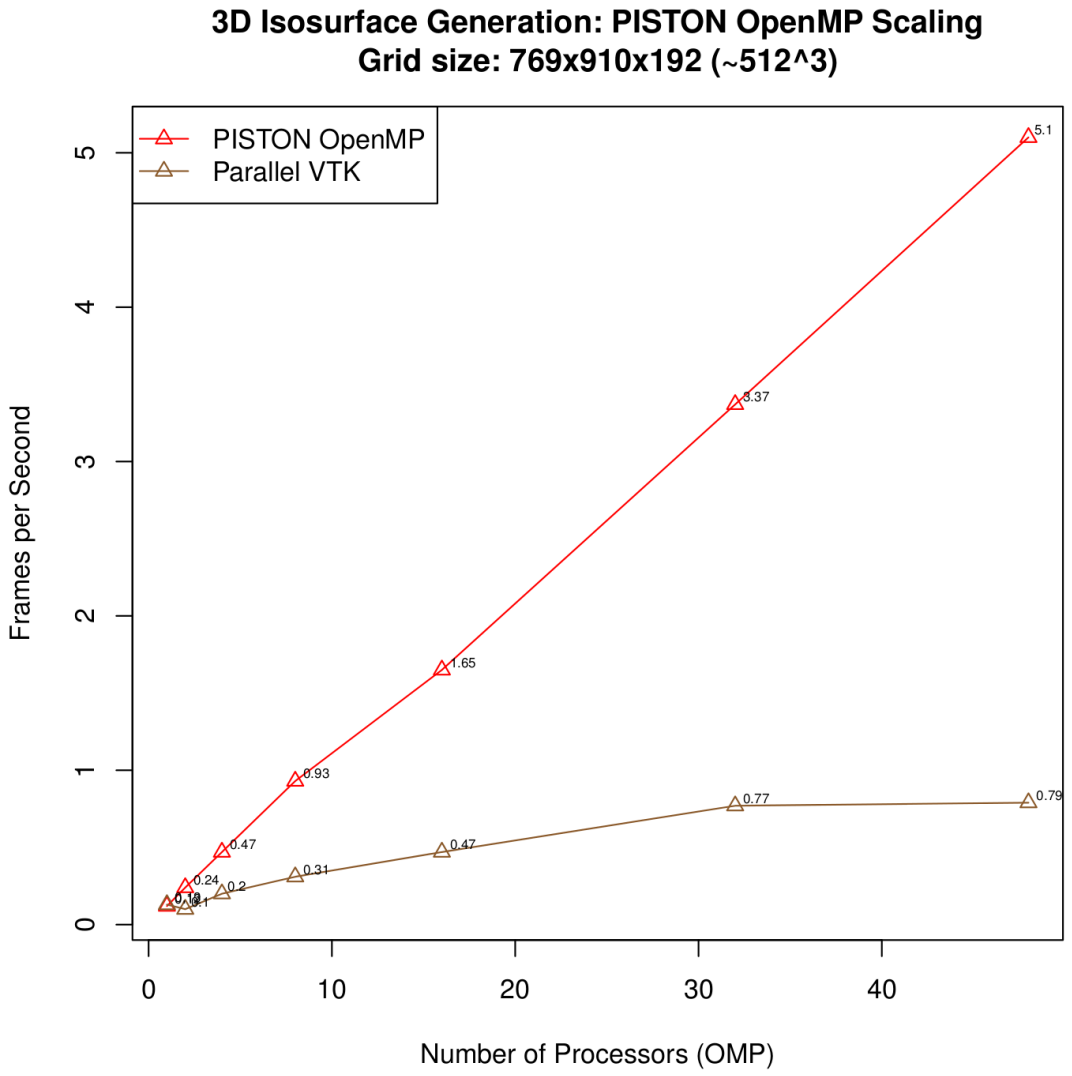
PISTON OpenMP Backend Performance

- 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



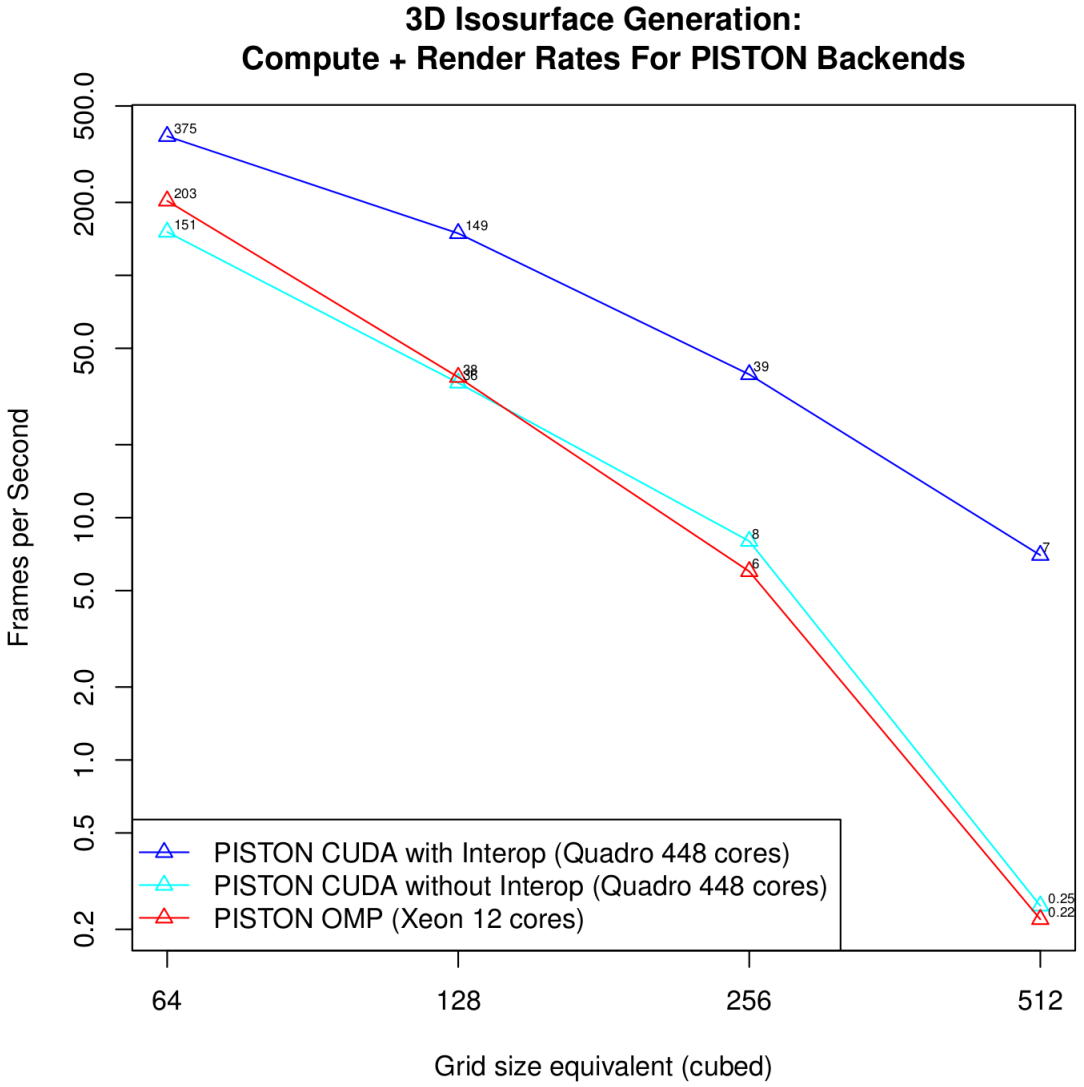
PISTON OpenMP Scaling Performance

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



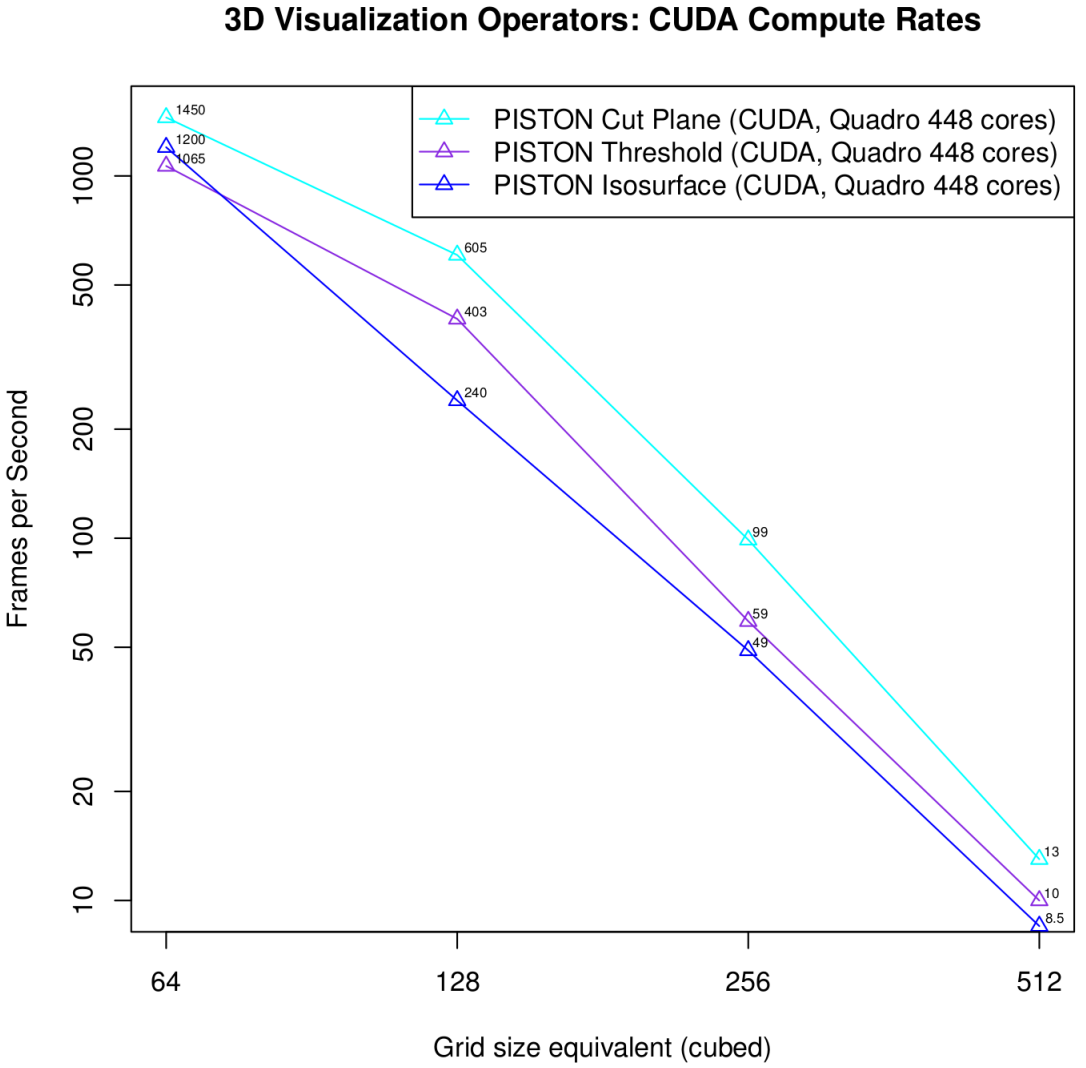
PISTON Compute and Render Results

- 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



Work in Progress: 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

Work in Progress: OpenCL Backend

- 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)
- Next Steps
  - Improve efficiency of implementation
  - Integrate our OpenCL backend more fully into Thrust
  - Implement threshold operator
  - Run OpenCL on other platforms (CPUs, etc.)

Future Work

- Open source release
  - Coming soon, last stage of approval process
- New targets
  - OpenCL/AMD, OpenMP/BlueGene
- More operators
- Integration with ParaView
  - Kitware is working on an experimental version

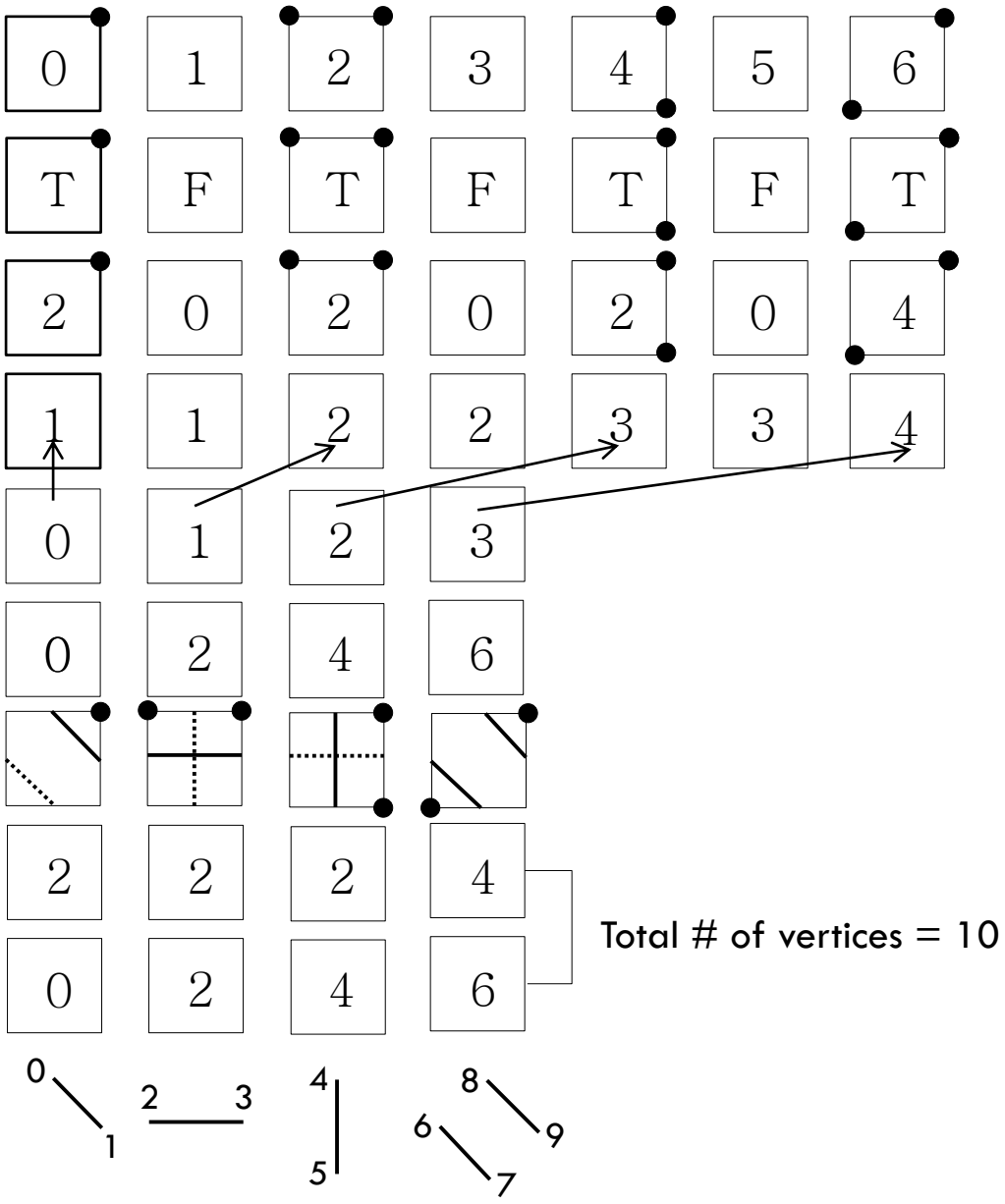
Acknowledgments and Resources

- 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
- See our demo at the Los Alamos booth!
- Ollie Lo will present a talk on PISTON at the NVIDIA booth at 4:30 on Tuesday

Isosurface with Marching Cube – Optimization

- Classify each cell, generate valid flags and # of output vertices
- Enumerate the number of valid cells by in-scan over valid flags
- Counting [0..# of valid cells)
- Binary search of the counts in the in-scan. This gives the global id of valid cells
- Use the global id of valid cells to fetch the # of output vertices
- Ex-scan on # of output vertices gives range of vertices generated by each valid cell
- The total # of vertices is the sum of the last elements.



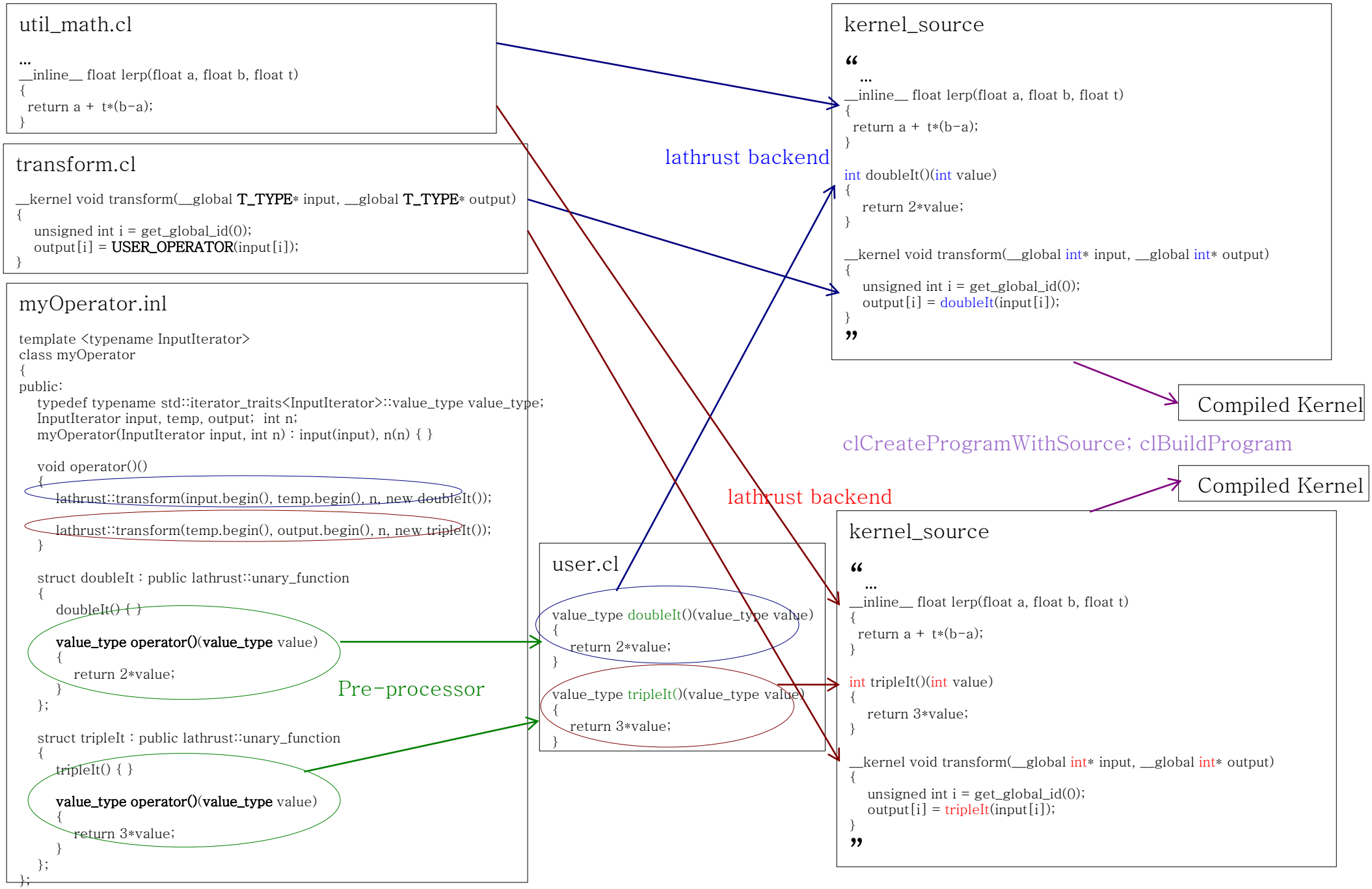
OpenCL Backend

- 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



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);
}
```

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

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]));
      }
    };
};
```

Pre-processor

user.cl

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