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

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

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
Outline

- Overview of Questions for the Panel
- 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, unstructured grids, more operators, ParaView integration, multi-node parallelism
Questions for the Panel

- What fundamental problem are you trying to solve?
  - Portable performance for visualization and analysis operators

- What are your plans to deal with exascale-specific issues (massive concurrency, distributed memory, memory overhead, fault tolerance)?
  - Data-parallel programming model; requires architecture-specific optimizations for a limited set of “embarrassingly parallel” primitive operators

- What is your philosophy for dealing with ambiguity of the exascale architecture (multiple swim lanes, heterogeneous architectures)?
  - Data-parallel programming model; requires architecture-specific optimizations for a limited set of “embarrassingly parallel” primitive operators

- How is your technology implemented?
  - Extension of NVIDIA’s Thrust library

- What is the long-term result for this effort? (Production software? Research prototype?)
  - Open-source repository (already available); integrated into VTK/ParaView (already implemented in prototypes)
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
- Most work in portability and abstraction layers/languages are not ready (yet)...
  - Scout, DAX, Liszt
- OpenCL: code is portable but performance is not
- Can we accelerate our visualization software with something that is based on “proven” technology and portable across different architectures?
  - NVidia Thrust library
Brief Introduction to Data-Parallel Programming and Thrust

- What is data parallelism?
  - When independent processors perform 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

- 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
  - Provided device_vector and host_vector data structures simplify memory management and host/device memory transfers
Videos of PISTON in Action
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

<table>
<thead>
<tr>
<th>input</th>
<th>4 5 2 1 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>5 6 3 2 4</td>
</tr>
<tr>
<td>inclusive_scan(</td>
<td>+</td>
</tr>
<tr>
<td>exclusive_scan(</td>
<td>+)</td>
</tr>
<tr>
<td>exclusive_scan(max)</td>
<td>0 4 5 5 5</td>
</tr>
<tr>
<td>transform_inscan(*2,+)</td>
<td>8 18 22 24 30</td>
</tr>
<tr>
<td>for_each(-1)</td>
<td>3 4 1 0 2</td>
</tr>
<tr>
<td>sort</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>copy_if(n % 2 == 1)</td>
<td>5 1 3</td>
</tr>
<tr>
<td>reduce(</td>
<td>+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>input1</th>
<th>0 0 2 4 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>input2</td>
<td>3 4 1 0 2</td>
</tr>
</tbody>
</table>

| 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

```
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

# of valid cells = 4
Total # of vertices = 10
```
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”

<table>
<thead>
<tr>
<th>Data Structures</th>
<th>Graph Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graphs: Neighbor reducing, distributing excess across edges</td>
<td>Minimum spanning tree</td>
</tr>
<tr>
<td>Trees: Leaffix and rootfix operations, tree manipulations</td>
<td>Maximum flow</td>
</tr>
<tr>
<td>Multidimensional arrays</td>
<td>Maximal independent set</td>
</tr>
<tr>
<td>Computational Geometry</td>
<td>Numerical Algorithms</td>
</tr>
<tr>
<td>Generalized binary search</td>
<td>Matrix-vector multiplication</td>
</tr>
<tr>
<td>k-D tree</td>
<td>Linear-systems solver</td>
</tr>
<tr>
<td>Closest pair</td>
<td>Simplex</td>
</tr>
<tr>
<td>Quickhull</td>
<td>Outer product</td>
</tr>
<tr>
<td>Merge Hull</td>
<td>Sparse-matrix multiplication</td>
</tr>
</tbody>
</table>

- 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 $512^3$

3D Isosurface Generation: CUDA Compute Rates

- NVIDIA Native CUDA Demo (Quadro 448 cores)
- PISTON CUDA Backend (Quadro 448 cores)
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

3D Visualization Operators: CUDA Compute Rates

![Graph showing performance rates for different visualization operations.](Graph.png)
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)
Marching Tetrahedra

- Current procedure
  - Tetrahedralize uniform grid
  - Generate isosurface geometry based on look-up table for tetrahedral cells
- Next step: tetrahedralize unstructured grids
- 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
- Ultimately want to support a large subset of ParaView filters, plus analysis operators
Open-Source Release

- Open-source release
  - Repository: [https://github.com/losalamos/PISTON](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