LA-UR-14-20777

Approved for public release; distribution is unlimited.

Title:

VOLUME-OF-FLUID INTERFACE RECONSTRUCTION ALGORITHMS ON NEXTGENERATION COMPUTER ARCHITECTURES

Author(s):

Marianne Francois
Li-ta Lo
Christopher Sewell

Intended for:

Proceedings of the ASME 2014 4th Joint US-European Fluids Engineering Division Summer Meeting and 12th International Conference on Nanochannels, Microchannels, and Minichannels. August 2014. Chicago, IL.

Proceedings of the ASME 2014 4th Joint US-European Fluids Engineering Division Summer Meeting and 12th International Conference on Nanochannels, Microchannels, and Minichannels
FEDSM2014
August 3-7, 2014, Chicago, Illinois, USA

FINAL - FEDSM2014-21894

VOLUME-OF-FLUID INTERFACE RECONSTRUCTION ALGORITHMS ON NEXT-GENERATION COMPUTER ARCHITECTURES

Marianne M. Francois
T-3: Fluid Dynamics and Solid Mechanics
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Li-Ta Lo
CCS-7: Applied Computer Science
Data Science at Scale Team
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

Christopher Sewell
CCS-7: Applied Computer Science
Data Science at Scale Team
Los Alamos National Laboratory
Los Alamos, NM 87545, USA

ABSTRACT

With the increasing heterogeneity and on-node parallelism of high-performance computing hardware, a major challenge to computational physicists is to work in close collaboration with computer scientists to develop portable and efficient algorithms and software. The objective of our work is to implement a portable code to perform interface reconstruction using NVIDIA's Thrust library. Interface reconstruction is a technique commonly used in volume tracking methods for simulations of interfacial flows. For that, we have designed a two-dimensional mesh data structure that is easily mapped to the 1D vectors used by Thrust and at the same time is simple to work with using familiar data structures terminology (such as cell, vertices and edges). With this new data structure in place, we have implemented a recursive volume-of-fluid initialization algorithm and a standard piecewise interface reconstruction algorithm. Our interface reconstruction algorithm makes use of a table look-up to easily identify all intersection cases, as this design is efficient on parallel architectures such as GPUs. Finally, we report performance results which show that a single implementation of these algorithms can be compiled to multiple backends (specifically, multi-core CPUs, NVIDIA GPUs, and Intel Xeon Phi coprocessors), making efficient use of the available parallelism on each.

INTRODUCTION

The variety of hardware architectures used in high-performance computing is large and continually growing. These include vendor-specific variations on staples such as GPUs and multi-core CPUs, as well as specialized architectures such as IBM's Blue Gene and the Cell processor, and emerging technologies such as Intel's MIC architecture. Supercomputer architectures currently in use at national laboratories range from Blue Gene (e.g., Argonne's Intrepid) to Cell (e.g., Los Alamos's Roadrunner) to GPUs (e.g., Oak Ridge's Jaguar upgrade). The Department of Energy has stated that systems using at least two different architectures will be built as part of its exascale computing initiative.

Our objective is to develop portable and efficient codes using the NVIDIA Thrust Library to run on various architectures. The Thrust library **[9]** is a C++ template library for CUDA. It provides a powerful, flexible and easy way to develop parallel algorithms and data structures. It provides a high-level interface to program on GPUs as well as multi-core CPUs (since it supports OpenMP and TBB). However, the Thrust library has a simplistic data model and only employs one-dimensional vectors, making it challenging to perform multi-dimensional physics-based simulation. Further Thrust's OpenMP backend is not optimized for all the different architectures on which it may be run. We have previously presented our PISTON framework, in which we have extended the Thrust library and implemented several common visualization algorithms, such as contouring, using this data-parallel model **[4]**. In this paper, we apply this methodology to a volume tracking code using a further extension of Thrust and PISTON, which we call PINION, to support simulation codes.

In the present work, we develop and implement a single portable code using PINION to perform piecewise linear interface reconstruction in two-dimensions. Interface reconstruction is a technique used in the volume tracking method **[6]** to limit numerical diffusion of the interface in the calculation of interfacial flows. It allows one to compute the location of the boundary (i.e. interface) between multiple fluids/materials given the volume fraction information on a mesh. Computations of incompressible interfacial flows on GPUs has been presented in **[2]** and **[5]** using volume-of-fluid method and in **[10]** using a level-set method. In these previous works, the focus was on accelerating the Poisson solver of the incompressible flow solver using GPUs. Here, our focus is only on the interface reconstruction algorithm as a starting point.

The paper is organized as follows. First, we describe how we have devised our two-dimensional mesh data structure and mesh data operators. Then, we describe our algorithms to achieve interface reconstruction. Finally, we present performance results on various architectures.

MESH DATA STRUCTURE AND OPERATORS

In FORTRAN, a two-dimensional Cartesian structured mesh is easily represented using a multi-dimensional array of indices (i,j). In Thrust, however, only one-dimensional arrays are available. The data model we have implemented in PINION for this work, uses a higher level of abstraction than the one we had in PISTON. In PINION, we provide a data model that supports physics-model implementation, whereas in PISTON none were provided, since PISTON is intended for visualization applications.

Mesh Data Structure

The mesh data structure we employ here consists of three one-dimensional arrays: list of cell ids, list of vertex ids and list of edge ids. The nomenclature and numbering for our mesh is illustrated in Figure 1.

Mesh Data Operators

To get our mesh connectivity between cell, vertex and edge ids, and to do operations with our mesh data, we have implemented several mesh operators to find adjacency, boundaries, and neighbors. Our main mesh data operators are listed below with a description of what they do:

vertex_to_edges_op Adjacency operator for vertices, given one vertex id, return ids of 4 edges sharing the vertex as {Left, Right, Bottom, Top}, -1 means non-existence/boundary edges.

vertex_to_cells_op Adjacency operator for vertices, given one vertex id, return ids of 4 cells sharing the vertex as {Lower Left, Lower Right, Upper Left,Upper Right}, -1 means non-existence/boundary cells.

edge_to_vertices_op Boundary operator for edges, given one edge id, return ids of the two end vertices as {Left, Right} or {Bottom, Top}.

edge_to_cells_op Coboundary/adjacency operator for edges, given one edge id, return ids of 2 cell ids sharing the edge.

cell_to_edges_op Boundary operator for cells, given one cell id, return ids of 4 edges as {Bottom, Right, Top, Left}.

cell_to_vertices_op Second order boundary operator for cells, given one cell id, return ids of the 4 vertices as {Lower Left, Lower Right, Upper Left, Upper Right}.

cell_von_neumman_neighbor_op Given a cell return the 4 orthogonal neighboring cells in the following order {West, East, South, North}.

cell_moore_neighbor_op Given a cell return the 8 neighboring cells in the following order {W, E, S, N, SW, SE, NW, NE}.

vertex_position_op Given a vertex id, return the coordinates of the position of that vertex.

cell_center_position_op Given a cell id, return the coordinates of the cell center position of that cell.

edge_center_position_op Given an edge id, return the coordinates of the edge center position of an edge.

edge_normal_op Given an edge id, return the orthogonal vector (i.e. the normal) to that edge. The direction of the normal vector always points to the "right" side of the edge. The magnitude of the vector is the length of the edge.

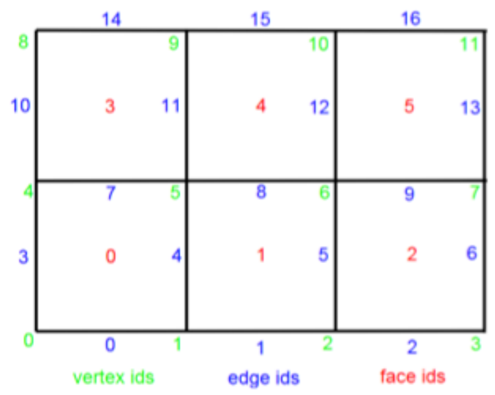

Figure 1. Mesh Data Structure Nomenclature and Numbering. The cell vertex ids are shown in green, the edges ids in blue and cell (face) ids in red.

ALGORITHMS AND IMPLEMENTATION WITH THRUST

The present piecewise linear interface reconstruction is divided into 3 steps:

1. Volume fraction initialization,
2. Calculation of the volume fraction gradient,
3. Interface reconstruction using an iterative volume matching procedure.

Volume Fraction Initialization

The volume fraction represents the volume (amount) of one material in a given computational cell with respect to the total volume of the computational cell. To initialize the volume fraction in every computational cell, we use a divide and conquer recursive algorithm. Given the computational grid and a mathematical expression for a shape (e.g. a circle) we compute the volume fractions in every cell by testing whether the cell vertices are inside or outside the 'shape'. If all the vertices are inside then the volume fraction of the given cell is set to one and if all the vertices are outside the volume fraction is set to zero. If the vertices of the cell are both inside and outside, we proceed by dividing the cell into four. The refinement is continued up to a specified level. In this work the maximum level is set to 5.

Code sample:
```
thrust::transform(grid.cell_id_begin(),
grid.cell_id_end(), d_vof.begin(),
make_vof_init(grid, circle()));
```

Interface Normal Calculation

Given the volume fractions, α, the interface unit normal, $\hat{n}$, is computed at the cell-center as the gradient of the volume fraction using Green-Gauss:

$$\hat{n} = \frac{\nabla\alpha}{|\nabla\alpha|} \qquad (1)$$

$$\nabla\alpha \approx \sum_{e=1}^{4} \alpha_e \hat{n}_e A_e \qquad (2)$$

where the index e, denotes the cell edge, A_e denotes the edge area and $\hat{n}_e$ denotes the unit vector normal. The edge volume fractions are obtained by first averaging the cell-centered volume fractions at cell vertices before averaging them on edges. The unit interface normal calculation is equivalent of using a 9-point stencil.

Code sample:
```
thrust::transform(grid.cell_id_begin(),
grid.cell_id_end(),d_vof_grad.begin(),
make_grad_vof(grid,vof_edge.begin(),
              d_area_edge.begin()));
```

Iterative Volume Matching Procedure

This step consists of finding the line equation that intersects the computational cell and for which the resulting bounded volume matches the initial material volume (given by the cell volume fraction). It consists of finding ρ in the following equation:

$$\hat{n} \cdot \vec{x} + \rho = 0 \qquad (3)$$

for which $f(\rho) = V(\rho) - V$ tends to zero, where V denotes volume.

In this work, we use a similar iterative procedure as the one employed in **[6]**. At each iteration, the intersection points of the line with the computational cell are found and the polygon area is computed until the polygon area matches the volume fraction. The main difference in our algorithm is the use of a look-up table to identify the intersection cases. The type of look-up table is standard in visualization algorithms **[7]**. Our look-up table is shown in Figure 2. It gives the number of vertices that are inside the line, the cell vertex ids, and the cell edge ids that the line intersects. Two cases, case 8 and case 12, are illustrated. Looking at case 8, {1,2,0,2,2,3,-1,-1} we see that there is 1 vertex located on the outside of the interface that has for vertex id 2.The line intersects two edges: the edge defined between vertex id 0 and vertex id 2, and the edge defined between vertex id 2 and vertex id 3. The "-1" values represent no entry. Looking at case 12 {2,3,2,0,2,1,3,-1}, we see that there are 2 vertices located on the outside of the interface and these vertices are vertex id 3 and vertex id 2. The line intersects two edges: the edge defined between vertex id 0 and vertex id 2, and the edge defined between vertex id 1 and vertex id 3.

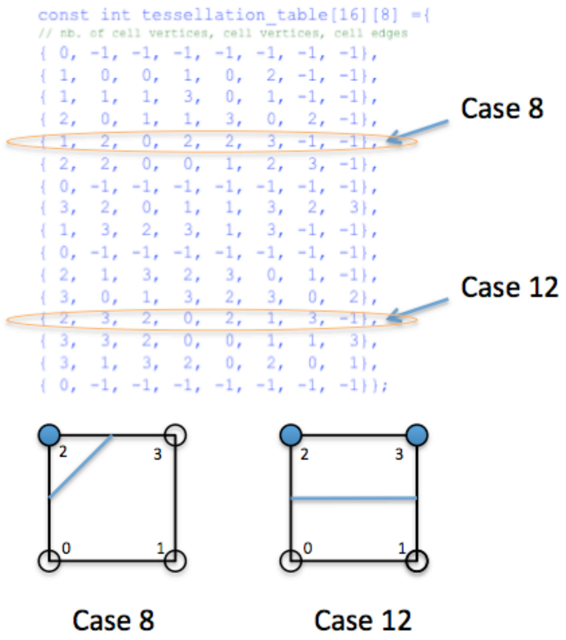
```
const int tessellation_table[16][8] ={
// nb. of cell vertices, cell vertices, cell edges
{ 0, -1, -1, -1, -1, -1, -1, -1},
{ 1,  0,  0,  1,  0,  2, -1, -1},
{ 1,  1,  1,  3,  0,  1, -1, -1},
{ 2,  0,  1,  1,  3,  0,  2, -1},
{ 1,  2,  0,  2,  2,  3, -1, -1},
{ 2,  2,  0,  0,  1,  2,  3, -1},
{ 0, -1, -1, -1, -1, -1, -1, -1},
{ 3,  2,  0,  1,  1,  3,  2,  3},
{ 1,  3,  2,  3,  1,  3, -1, -1},
{ 0, -1, -1, -1, -1, -1, -1, -1},
{ 2,  1,  3,  2,  3,  0,  1, -1},
{ 3,  0,  1,  3,  2,  3,  0,  2},
{ 2,  3,  2,  0,  2,  1,  3, -1},
{ 3,  3,  2,  0,  0,  1,  1,  3},
{ 3,  1,  3,  2,  0,  2,  0,  1},
{ 0, -1, -1, -1, -1, -1, -1, -1}};
```

Figure 2. Look-up table to identify intersection cases. Two intersection cases (case 8 and case 12) are given as examples.

CODE GENERATION

In PINION, we need only a single version of source code that runs on various architecture. In order to generate executables that run on different backends (single-thread, multi-thread and GPU) we simply construct different rules in our CMake file by using different compilers and flags:

```
    add_executable(Vof_2D_CPP Vof_2D.cpp)
    set_target_properties(Vof_2D_CPP PROPERTIES
COMPILE_FLAGS "-
DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_CPP -
std=c++0x")

    add_executable(Vof_2D_OMP Vof_2D.cpp)
    set_target_properties(Vof_2D_OMP PROPERTIES
COMPILE_FLAGS "-fopenmp -
DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_OMP -
std=c++0x")
    target_link_libraries(Vof_2D_OMP pthread gomp)
```

```
cuda_add_executable(Vof_2D_GPU Vof_2D.cu)
target_link_libraries(Vof_2D_GPU pthread)
```

The benefit of a single version of source code is to facilitate development and debugging. By using various debugging and memory checking tools, and assuming the backends are implemented correctly, the debugging process can be performed on the single threaded CPU and does not have to be repeated for other backends (multi-threaded CPU and the GPU), hence saving significant effort.

RESULTS

In this section, we present performance results of our single implementation of a piecewise linear interface reconstruction algorithm. The test case considered is a circle of radius 0.25 centered at (0.5,0.5) in a unit square domain. The computational mesh is varied from 256x256 up to 8192x8192. The tests are performed on various architectures: single-threaded CPU, multi-core CPU, GPU and MIC(Intel Xeon Phi).

Our OpenMP results were obtained using our modified version of Thrust, which includes our parallel OpenMP implementation of the scan (prefix sum) operator **[1]**.

Platform Characteristics

The CPU, OpenMP and GPU tests are performed on a HP Z800 workstation that has two Intel Xeon X5660 processors and a total of 32GB of memory. Each of the processors has 6 cores. Each of these cores can run two threads "simultaneously". Thus we can run up to 24 OMP threads.

The GPU installed on this workstation is a Nvidia Quadro 6000. There are 448 cores with 6GB of memory (http://www.nvidia.com/object/product-quadro-6000-us.html).

The MIC results are running in native mode on a single Intel Xeon Phi Coprocessor on the Stampede system at the Texas Advanced Computing Center (https://www.xsede.org/web/guest/tacc-stampede#overview). The Xeon Phi coprocessor has 61 cores and 512bit SIMD registers.

Weak Scaling

We have timed the three main steps of our algorithm: volume fraction initialization, volume fraction gradient, and interface reconstruction. The timings are performed following the standard timing procedure for the Thurst library. We execute ten trials per algorithm and within these ten trials, there are 100 repeated runs. The total time for each trial was measured by gettimeofday() on the CPU/OMP backends and by the CUDA event timer on the CUDA backend. The total time is then divided by the one thousand total iterations to obtain the reported average time. The results are shown in Figure 3, Figure 4, and Figure 5 for the volume fraction initialization algorithm, the volume fraction gradient algorithm and the interface reconstruction algorithm, respectively.

As expected, the run time increases roughly linearly with the domain size (grid size) on all platforms. The times on the single-threated CPU are the slowest. Using multiple threads (OMP) improves the performance. The MIC results are roughly comparable to the OMP results. The benefit of GPUs are clearly demonstrated for large grid size.

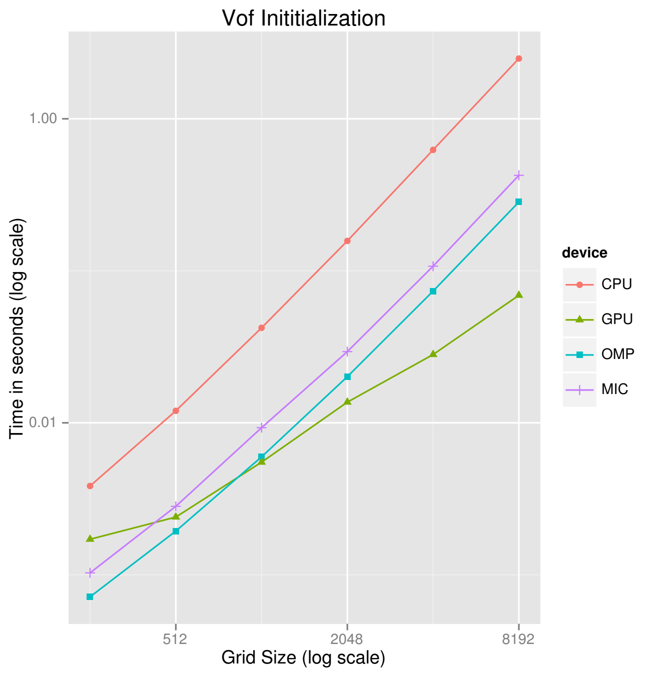

Figure 3. Weak scaling plots for the volume fraction initialization algorithm. The grid sizes range from 256^2 to 8192^2.

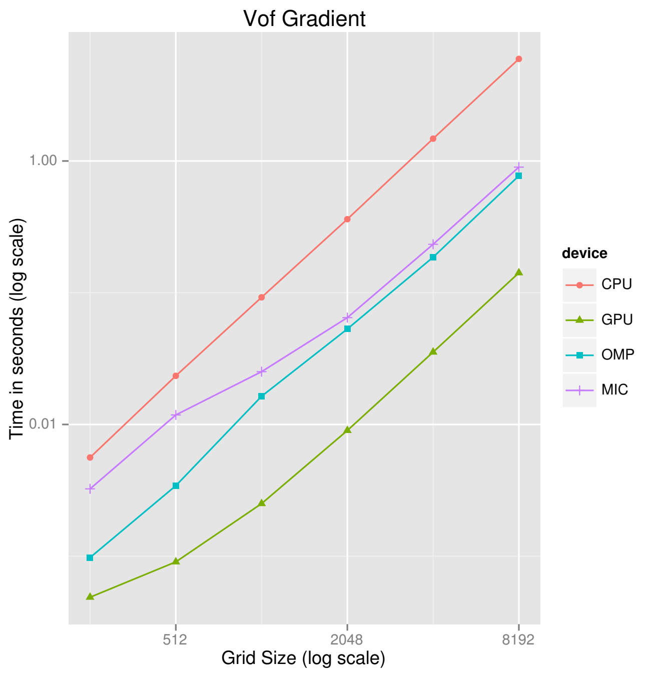

Figure 4. Weak scaling plots for the calculation of the volume fraction gradient (interface normal). The grid sizes range from 256^2 to 8192^2.

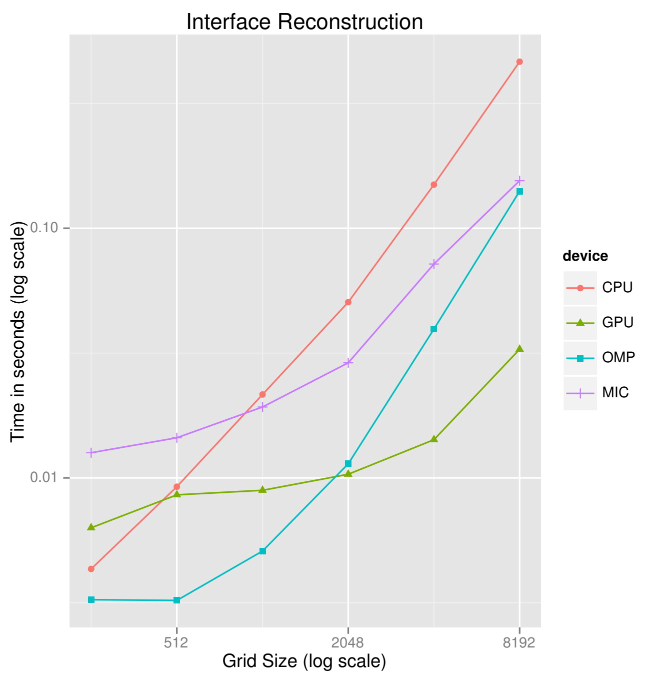

Figure 5. Weak scaling plots for the interface reconstruction (iterative volume matching procedure) algorithm. The grid sizes range from 256^2 to 8192^2.

Strong Scaling

Figure 6 shows the strong scaling (for a constant grid size problem) of the three algorithms with the number of OpenMP threads. This demonstrates that our algorithms make efficient use of the available parallelism.

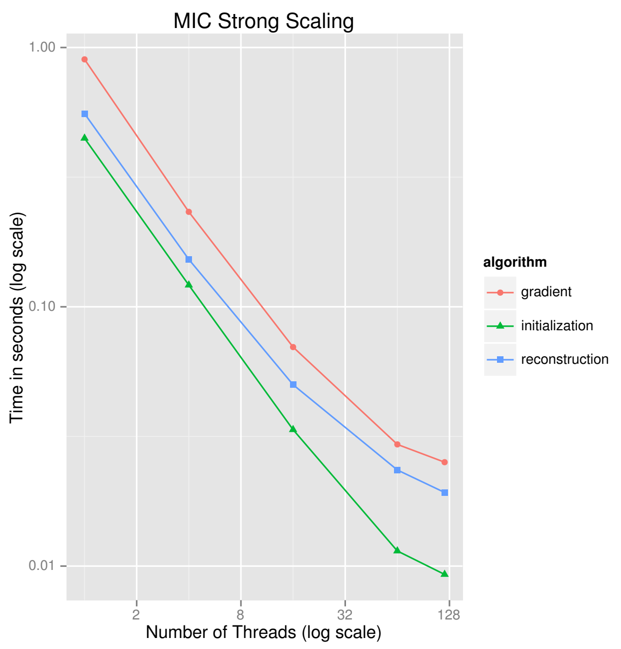

Figure 6. Strong scaling obtained on the MIC for a constant grid size of $1,024^2$.

We have also implemented distributed versions of these algorithms based on our distributed extension of Thrust as presented in **[8]**. This allows them to be run on multiple instances of any of the supported architectures (for example, across multiple CPUs, GPUs, or Xeon Phi coprocessors). In our distributed backend, the data-parallel operators are implemented by calling the Thrust shared-memory parallel operators on each node, while using MPI to communicate between nodes. Ghost cells for structured grids are handled transparently by our distributed backend. We plan to present the details of this implementation and performance results in an upcoming paper.

CONCLUSION

We have implemented a two-dimensional piecewise linear interface reconstruction algorithm using the NVIDIA Thrust library and tested the performance of our algorithm for single and multiple threads, as well as on a GPU and MIC architectures. The interface reconstruction algorithm is currently iterative; in the future we plan to implement an analytical algorithm [3] to avoid the iteration procedure and compare computational performance. We also intend to further optimize a MIC-specific OpenMP backend to more fully take advantage of the properties of this architecture, such as its wide vector width. Future work will also include implementation in 3D and on other types of meshes**.**

NOMENCLATURE

α	volume fractions
ρ	constant in line equation
A	area
e	edge index
f	function
$\hat{n}$	unit normal
V	volume
CPU	Central Processing Unit
GPU	Graphics Processing Unit
MIC	Many Integrated Core
OMP	Open Multi-Processing

ACKNOWLEDGMENTS

This work was performed under the auspices of the National Nuclear Security Administration of the US Department of Energy at Los Alamos National Laboratory under Contract No. DE-AC52-06NA25396. This work is supported by the Laboratory Directed Research and Development Exploratory Research program under the project “Physics-Based Data Models and Architecture-Optimized Backends for a Portable Data-Parallel Computation Library”. We acknowledge the use of XSEDE, supported by National Science Foundation award number OCI-1053575. We would also like to thank Jim Ahrens of Los Alamos National

Laboratory for his guidance of the PISTON and PINION projects. LA-UR-14-20777.

REFERENCES

[1] Blelloch G., Vector Models for Data-Parallel Computing, MIT Press. ISBN 0-262-02313-X. 1990.

[2] Codyer S., Raessi M., and Khanna G., Using Graphics Processing Units to accelerate numerical simulations of interfacial incompressible flows, ASME Fluid Engineering Conference, Puerto Rico, USA, 2012.

[3] Diot S., Francois M.M., Dendy E.D., An interface reconstruction method based on analytical formulae for 2D planar and axisymmetric arbitrary convex cells, submitted to Journal of Computational Physics, 2103.

[4] Lo L-T., Sewell C., Ahrens J., PISTON: A Portable Cross-Platform Framework for Data-Parallel Visualization Operators, Eurographics Symposium on Parallel Graphics and Visualization, May 13-14, 2012.

[5] Nagatake T., Kunugi T., Application of GPU to computational multiphase fluid dynamics, IOP Conference Series: Materials Science and Engineering, 10(1), p012024, 2010.

[6] Rider W. Kothe D.B., Reconstructing volume tracking, Journal of Computational Physics, 112-152, 1998.

[7] Schroeder W., Martin K., Lorensen B., Visualization Toolkit, An Object-Oriented Approach to 3D Graphics, Kitware, 2006.

[8] Sewell C, Lo L-T., Ahrens J., Portable Data-Parallel Visualization and Analysis in Distributed Memory Environments, IEEE Symposium on Large-Scale Data Analysis and Visualization, October 13-14, 2013.

[9] Thrust Library, https://developer.nvidia.com/Thrust

[10] Zaspel P., Griebel M., Solving incompressible two-phase flows on multi-GPU clusters, Computers and Fluids, 80, 356-364, 2013.