LA-UR-18-28727
Approved for public release; distribution is unlimited.

Title: GPU ACCELERATION OF VOLUME FRACTION AND CENTROID COMPUTATION FROM GENERAL SHAPES ON UNSTRUCTURED MESH

Author(s): Ha, Quang-Thinh

Intended for: Report

Issued: 2018-09-13

Disclaimer:
Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

GPU ACCELERATION OF VOLUME FRACTION AND CENTROID COMPUTATION FROM GENERAL SHAPES ON UNSTRUCTURED MESH

Quang-Thinh Ha
Mentor: Rao Garimella

Los Alamos National Laboratory
Summer 2018

Abstract

One important requirement for coupling multi-physics models is a means to remap fields between meshes in an accurate and conservative manner. Since the remapping algorithm is embarrassingly parallel, this motivates the use of Graphical Processing Units (GPU) for such routine. However, since our remapping code, `portage`, is fairly complex, we explore the use of GPUs through a simpler problem of generating material's volume fraction and centroid inside each cell of the mesh. We use the same programming structure that is used in `portage`, namely, on-node parallelism via NVIDIA's `thrust` library. Using a similar programming structure enables us to draw lessons from this study that can be applied to the more complex problem. We have found that while the speed-up is pronounced (10-100 times), significant effort is required to re-factor the code so that `thrust` can be used with a `CUDA` backend. Moreover, `thrust` takes away the fine grain control that vanilla `CUDA` programming gives us.

1. Introduction

The `portage` library provides a framework for remapping of field data between meshes, between particles, and between meshes and particles [1]. The remapping algorithm within `portage` is divided into three phases, which are qualitatively labelled as:

- *Search* - find candidate cells/particles that will contribute to remap of a given target cell/particle.
- *Intersect* - Calculate the weight of each candidate's contribution to the remap of a given target cell/particle. This may include higher moments if requested.
- *Interpolate* - Using the weights and moments, along with appropriate limiters, reconstruct the field data for a given target cell/particle.

Intersection of unstructured meshes (particularly in 3D) is very expensive - accounts for 80% or more of total run time. Naturally, an attempt to speed up the routine via parallel programming was attempted. The first motivation comes from the fact that the algorithm is embarrassingly parallel, which is perfectly suited for thread parallelism. In order to ensure portability and flexibility within the library, NVIDIA's `thrust` was used as the back-end parallel driver [2].

Through `thrust`, different choices of back-end can be utilised. The current `portage` has reported good scaling using `MPI` and `OpenMP` **[CITATION NEEDED]**. To fully exploit the embarrassingly parallel's nature of the library, the next logical step would be utilising Graphical Processing Units (GPU). At the time of writing this report, the latest GPU - Volta V100 - contains 5120 cores [3], which is roughly more than 70 times the available number of cores on the top-of-the-line Central Processing Units (CPU) - Intel Xeon Phi Knights Landing 7290 [4]. Hence, it makes sense to spend efforts on extending `portage` to include GPU computing capability. The reduction in run-time could potentially be tremendous.

In order to fully understand the benefits and limitations of GPU acceleration through `thrust`, we start off using a simpler code base - namely `tangram` - which shares substantial similarities with `portage`. In this report, we focus on using `tangram` to calculate the volume fraction and centroid. To reduce the complexity even further, sampling strategy was used instead of intersection-based method. As expected, the speed-up achieved with GPU computation is really pronounced, and using the generated data, interfaces can be computed with the Moment-Of-Fluid's algorithm accurately [5].

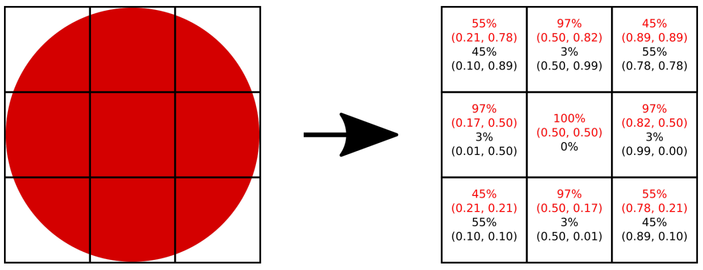

Figure 1: Generating volume fraction and centroid on a square mesh.

2. Problem Definition

The problem is presented qualitatively in Figure 1. Given the configuration of a material, one would like to obtain the volume fraction of that material and where the centroid is. For this case, there are two materials: material 1, red circle, lying on top of material 2, white background. Inside the regular 3-by-3 square mesh, we would like to compute the volume fraction and the centroid of each material. Such information is neccessary for interfacial reconstruction technique including Volume-Of-Fluid and Moment-Of-Fluid (see [6] and [5]).

3. Sampling Points Algorithm

For rectangular meshes, points are generated from the joint uniform distributions which are scaled to each side of the cell, Figure 2. Each point will be labeled according to the material which they fall into. Once the labeling is done, the volume fraction of each material is approximated by the proportion of points which are labeled with that material. The centroid is calculated by averaging all of the labeled points' coordinates. With a large-enough number of points, this approximation should be close to actual values.

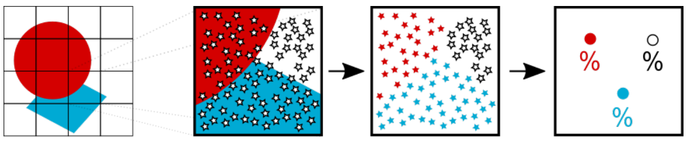

Figure 2: Overview of sampling point algorithm.

In case of non-rectangular cells, points are still generated using joint scaled uniform distributions, but those that are outside the cell are discarded in volume fraction and centroid calculation. As one would expect, the algorithm for checking whether a point is inside a polygon can get complicated, and this report is not meant to address such issue.

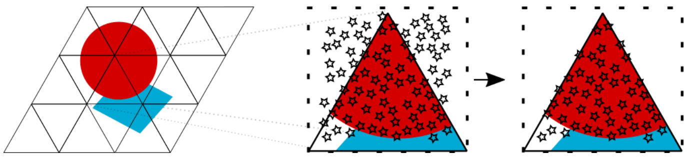

Figure 3: Discard points outside of non-rectangular cells.

The pseudocode for the sampling algorithm is as followed:

Algorithm 1 Pseudocode for sampling point algorithm.

```
 1: for cellID = 0 to num_cells do
 2:     for pointID = 1 to NPOINTS do
 3:         point ← generate_random_Point()
 4:         if point_in_cell(cellID, point) then
 5:             matID ← getmatID(point)
 6:             volume_fraction[matID] ← volume_fraction[matID] + 1
 7:             centroid[matID] ← centroid[matID] + point
 8:         end if
 9:     end for
10:     volume_fraction[matID] ← volume_fraction[matID]/NPOINTS
11:     centroid[matID] ← centroid[matID]/NPOINTS
12: end for
```

in which each cell includes the following struct:

```
#define MAXMATS 50
struct vfcen_t {
  int nmats = 0;
  int matids[MAXMATS] = {};
  double vf[MAXMATS] = {};
  Tangram::Point<dim> cen[MAXMATS];
};
```

which captures (1) how many materials are inside each cell `nmats`; (2) the list of all materials IDs inside each cell `matids`; (3) the volume fractions for the respective materials `vf` and (4) their centroids `cen`.

4. Using `thrust` with `OpenMP` and `CUDA`

4.1. Using *OpenMP* back-end

Both `tangram` and `portage` use `thrust` for thread parallelism, which offers high-level parallel analog of the `C++` Standard Template Library `STL`. One selling point of using `thrust` is its ability to switch between back-ends - `OMP`, `CUDA` or `TBB` - upon configuring with `cmake`. With `OMP` back-end, Algorithm 1 can be written using functors:

```
// Calling evaluator over cells
int main() {
[...]
Tangram::vector<int> cellID(NCELLS);
// Fill cellID and transform
Tangram::transform(cellID.begin(), cellID.end(),
                   vfcen.begin(),
                   vf_evaluator);
}
// Then inside vf_evaluator
struct vf_evaluator {
  operator()(int cellID) {
  Tangram::vector<Tangram::Point<dim>> ptID(NPOINTS);
  Tangram::transform(ptID.begin(), ptID.end(),
                     matID.begin(),
                     feature_evaluator);
  [...]
  return vfcen;
  }
}
// Then inside feature_evaluator
struct feature_evaluator {
  operator()(Tangram::Point<dim>) {
    [..]
    // Check with material ID it is
    return matID;
  }
}
```

where `dim` is the dimension of the problems, 2D or 3D; `Tangram::vector` is `thrust::device_vector` when compiled with `thrust` but is `std::vector` otherwise; `Tangram::transform` is `thrust::transform` with `thrust` but is `std::transform` otherwise; `vf_evaluator` takes a `cellID` as input and returns `vfcen_t` calculated for each cell; and `feature_evaluator` takes each point's coordinates as the input and compute its corresponding material `matID` in return.

Unfortunately, the above code cannot be used as-is for `CUDA`, one of the main reason being illegal memory access between CPU and GPU. Further details on this matter shall be discussed later in this report.

4.2. Using *CUDA* back-end

Logistically, there are a number of rules that need to be followed in order to allow compilation of `thrust` with `cuda`. First, the code needs to have `*.cu` extension, as needed by

NVIDIA's compiler nvcc. Since nvcc can be fored to treat *.cc or *.cpp files as CUDA files, cmake can achieve similar behaviour by setting the LANGUAGE property of the executable target. This also means that CUDA has to be added as a language (besides CXX) within the cmake project. Then, nvcc will need to compile CUDA codes into object files, and host compilers (icc, mpicc etc.) will link the object files together to form the final executable file.

The principle of using CUDA as thrust's back-end is similar to how one would launch a CUDA kernel, except thrust automatically handles everything for the users. To be more specific, thrust will allocate and deallocate GPU (*device*) memory under the hood - without users explicitly calling cudaMemAlloc and cudaFree. Similarly, GPU's kernel size is determined inside thrust's routine.

One of the key principle to keep in mind, when using thrust with CUDA, is the distinctive location of the memory. Functions and variables defined on CPU or *host* can't be called for execution on GPU or *device*. Also, any input or output routines (i.e. std::cout etc) is not allowed to be called inside device code - it has to be executed on host. For functions executing on the device, decorators of either __device__ or __host__ __device__ need to be added. The former defines a device's function defined on device's memory and can only be called by other device's function. The latter one allows such device's function to be called from the host's side.

In terms of pointers and references, cross-memory access between host and device is forbidden when using thrust - CUDA itself has *unified memory* which allows for host memory to be accessed from the device in a seamless fashion. In short, device's pointers or references of host's memories will generate segmentation fault at run time, albeit error-free during compilation. To further illustrate these points, the following codes are provided. For thrust/OMP:

```
struct OMP_Functor {
 // Can pass and store vector
 std::vector<FEATURE> f_;
 int nf_;
 // Constructor doesn't need to be decorated
 OMP_Functor(std::vector<FEATURE> const& f_) :
   f_(f), nf_(f.size()) {}

 // Operator doesn't need to be decorated
 vfcen_t operator()(int cellID) {
   // Can declare Tangram::vector
   Tangram::vector<Point> points(NPOINTS);
   [...]
 }
}
int main(void) {
 [...]
 Tangram::vector<int> cellID(numCells);
 Tangram::vector<vfcen_t> vfcen(numCell);
 thrust::transform(cellID.begin(),
                   cellID.end(),
                   vfcen.begin(),
                   OMP_Functor);
 [...]
}
```

while for thrust/CUDA, certain modification is required. Besides decorators, memories and pointers require careful practice to avoid segmentation fault error:

```
struct CUDA_Functor {
 // Can NOT pass vector - use device pointer
 Tangram::pointer<FEATURE> f_ptr_;
 int nf_;
 // Constructor has to be decorated
 __host__ __device__
 CUDA_Functor(Tangram::pointer<FEATURE> f_ptr,
              int nf) :
 f_ptr_(f_ptr), nf_(nf) {}

 // Operator has to be decorated
 __host__ __device__
 vfcen_t operator()(int cellID) {
   // Have to use thrust::malloc with pointer
   thrust::pointer<Point,
     thrust::device_system_tag> pts_ptr_;
   pts_ptr_ = thrust::malloc<Point>(thrust::device,
                                    NPOINTS);
   [...]
 }
}
int main(void) {
 [...]
 Tangram::vector<int> cellID(numCells);
 Tangram::vector<vfcen_t> vfcen(numCell);
 Tangram::transform(cellID.begin(),
                    cellID.end(),
                    vfcen.begin(),
                    CUDA_Functor);
 [...]
}
```

In summary, the viable solution is to minimise the cross-memory access between host and device. For our problem, we can simply (1) copy the entire mesh from host to device, then (2) perform calculation on the device to obtain volume fraction and centroid, and finally (3) copy the results back to the host, Figure 4.

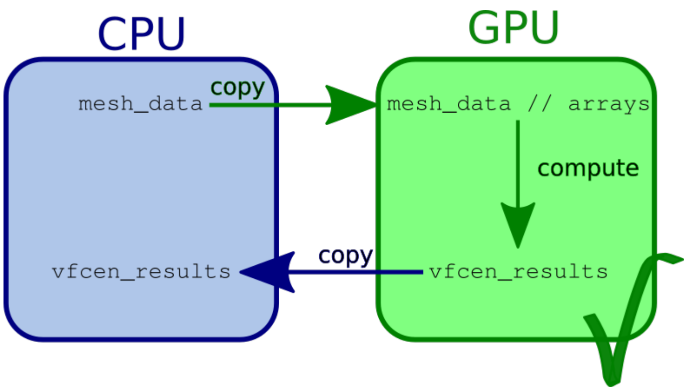

Figure 4: Working scheme for data transfer between CPU and GPU.

Additionally, the data residing on device should be contiguous arrays and should not exploit C++ Standard Template Library (STL) classes. Hence, traditional mesh wrappers provided in tangram and portage cannot be used since they utilise std::vector data structure from C++. To tackle this problem, we have implemented a new mesh wrapper, cuda_mesh_wrapper, which explicitly copies all the mesh's information upon initialisation:

```
public:
  // Copy mesh to GPU
  __host__
  void initialize(Mesh_Wrapper& input) {
    nodeCoords_ =
      thrust::device_malloc<Tangram::Point<D>>(numNodes);
    cellNodeCounts_ =
      thrust::device_malloc<int>(numCells);
    cellNodeOffsets_ =
      thrust::device_malloc<int>(numCells);
    cellToNodeList_ =
      thrust::device_malloc<int>(totCellNodes);
    faceNodeCounts_ =
      thrust::device_malloc<int>(numFaces);
    faceNodeOffsets_ =
      thrust::device_malloc<int>(numFaces);
    cellFaceCounts_ =
      thrust::device_malloc<int>(numCells);
    cellFaceOffsets_ =
      thrust::device_malloc<int>(numCells);
  }
  // Functions to use as other mesh wrappers
  __host__ __device__
  void cell_get_coordinates(int const cellid,
   thrust::pointer<Tangram::Point<D>,
               thrust::device_system_tag> *cnode_ptr,
   int& ncnode) const {[...]}
  __host__ __device__
  void cell_get_facetization(int const cellid,
   thrust::pointer<int,
               thrust::device_system_tag> *facetpoints,
   int &nfacets,
   thrust::pointer<Tangram::Point<3>,
                   thrust::device_system_tag> *points,
   int &npoints) const {[...]}
```

The key point behind cuda_mesh_wrapper is mainly striding 2D arrays containing rows of varying sizes into two 1D arrays: one containing the 'flatten' version of the 2D arrays (by concatenating a row to the end of the previous one), the other one collects the 'offset', which is the length of each row in the original 2D array. Two functions cell_get_coordinates and cell_get_facetization helps with readability within the code base. Each of them will return the corresponding cell's coordinates or facets given the cell's ID cellID.

4.3. *Underlying difficulties with thrust/CUDA*

Data race condition is one of the main concern in CUDA programming. The situation can be illustrated via the following snippet:

```
__global__ void collect(int *vfcen.matids) {
  // Generate point and get point's material ID
  Point p;
  int pid = getMatID(p);
  // Check if matID is already collected
  if (pid not in vfcen.matids[nmats]) {
    nmats++;
    vfcen.matids->pop(pid);
  }
}
```

The obtained array of matids for the list of materials inside the specific cell will be incorrect. This problem is commonly referred as *stream compaction*: some, but not all, threads will create a new value which needs to be stored in an array without gaps. One solution would be using stream compaction algorithm available in thrust, but this will require two passes of transform. The solution implemented inside this report is simply *bucketing*:

```
__global__ void collect(int *vfcen.matids) {
  // Generate point and get point's ID
  Point p;
  int pid = getMatID(p);
  // Collect all the points
  vfcen.matids[pid]++;
  // Then perform a count at the end
 }
```

All the values of pid is assumed to fall within the size of vfcen.matids, each occurence of pid triggers an increment of vfcen.matids[pid]. A final count and evaluation is required as a post-processing step.

Additionally, thrust hides explicit access to different types of memory and provides no mechanism for allocating kernel's size. Specifically, if one needs to parallel process more elements than the number of available CUDA cores, thrust does not provide a convenient way to schedule them. Instead, we process the work in groups of N elements at a time, where N is the number of available cores on the GPU architecture (N = 5120 for Volta P100). This way, we can 'force' thrust to launch a kernel where each thread is executed on a single CUDA core.

5. Performance benchmark

The code is implemented as part of tangram's application. A simple square (2D) or cube (3D) mesh is implemented inside app/simple-vfgen-cuda, while app/vfgen-cuda can take Exodus [7] meshes as input.

5.1. *Timing on normal GPU*

The performance using thrust with CUDA is compared against the original version using thrust with 32-thread OMP. Both of these run cases, CUDA and 32-thread OMP generates 50,000 particles per cell, and each test is performed five times to capture the averages and the error bands. Initially, the result was collected on non-Votla GPUs.

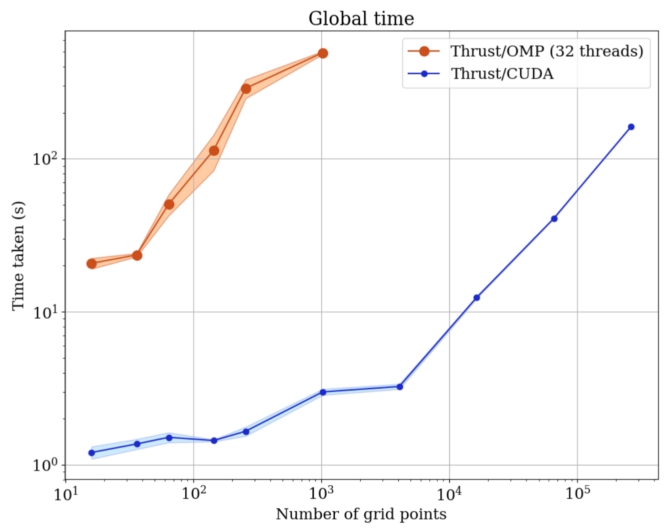

Figure 5: Timing comparison between OMP and CUDA - 2D cases.

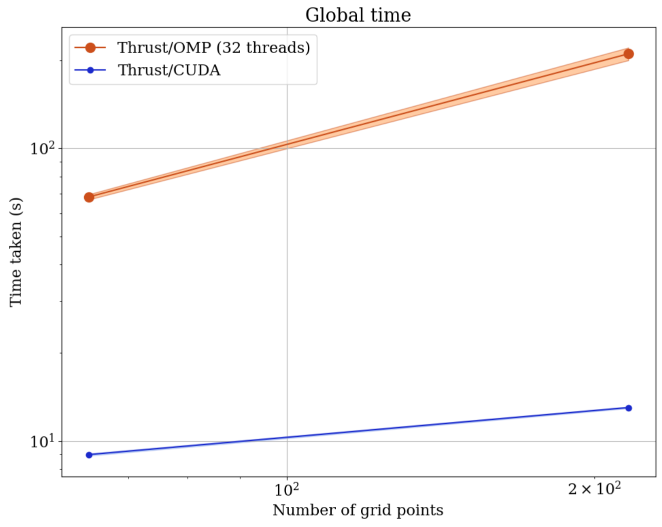

Figure 6: Timing comparison between OMP and CUDA - 3D cases.

It is obvious from Figure 5 and 6 that CUDA performs significantly better than OMP. Using thrust with CUDA offers between 10 to 100 times of faster run-time. This can be due to both the fact that GPU has more threads than CPU and the problem is strongly embarrassingly parallel.

5.2. *Timing on Volta V100*

To achieve better understanding on specific timing of individual tasks, the algorithm implemented with thrust/CUDA was executed on Volta V100. From both Figure 7 and 8, the time taken to copy the mesh from CPU to GPU takes roughly one-fourth of the overall run time. At small problem size (2D mesh with dimension 4-by-4), the global run-time is dominated by the CPU-to-GPU transfer time. As previously stated, the entire mesh needs to be copied over to GPU to avoid run-time error. The larger the mesh size, the more data is required to transfer. Data transfer between host and device still contributes significantly to the global run-time of GPU programs[8]. Figure 7 and Figure 8 both show that roughly one-fourth of the overall routine is consumed by data movement alone.

On the other hand, the major task remains calculating the volume fraction and centroid of the materials. For large problem size, it takes between 65% to 75% of total run-time.

6. Test result with interface reconstruction

The sampling algorithm was tested by using the results to reconstruct material interfaces using algorithms in tangram. In Figure 9, from left to right the number of cells increases from 3-by-3 to 50-by-50 to 1000-by-1000. The last image on the right show the result obtained from using the test case bailey128.

In 3D, Figure 10, the reconstructed interface using moment-of-fluid shows reasonable agreement with the reference. Both co-centric spheres in the middle is captured and reconstructed nicely. This is also similar for the inclined half-space plane between materials.

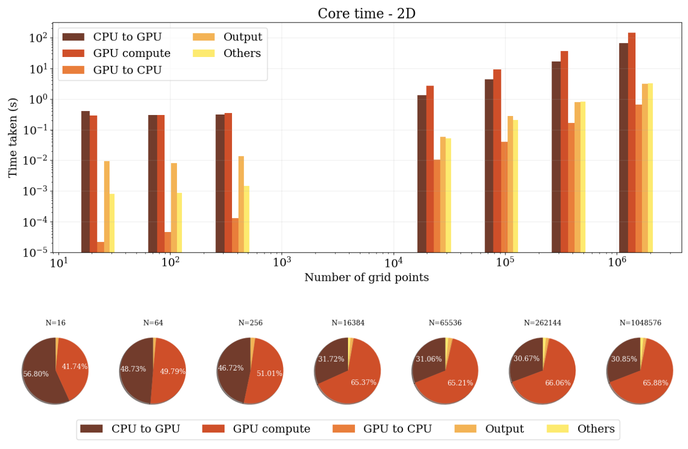

Figure 7: Timing breakdown for 2D cases on Volta V100.

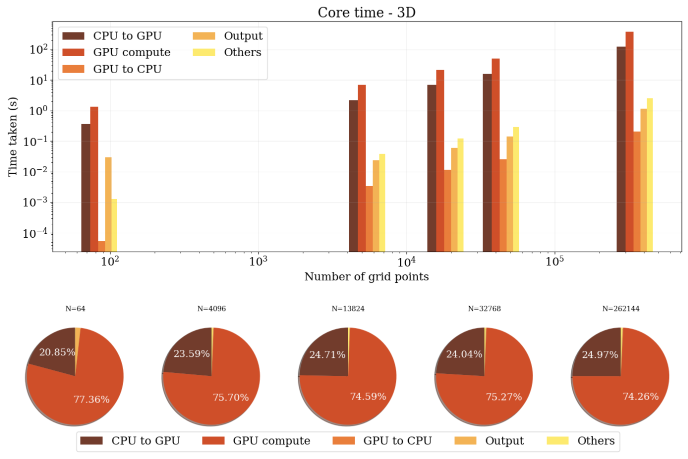

Figure 8: Timing breakdown for 3D cases on Volta V100.

7. Summary and Further Work

In this project, we have demonstrated the advantages of porting the volume fraction and centroid generation routine onto GPU. With the multi-core nature of GPU hardware in combination with the embarrassingly parallel nature of the algorithm, the overall run-time benefits tremendously.

An important point that is worth nothing is that thrust allows CUDA and OMP to be mixed within a code. A quick trial of using OMP for the first transform (inside main) while using CUDA for the second transform (inside vf_evaluator) does *not* yield favorable results. Further investigation exposes that each OMP thread will *sequentially* launch its own GPU kernel, which ends up creating a severe bottleneck.

There are certain drawbacks which make spending more effort on using thrust with CUDA remaining questionable. First, the number of lines that are specific to thrust/CUDA is noticeable. To the point that it poses a question whether the project could benefit better from using CUDA specific code instead (i.e. without relying on STL benefits of thrust). The counter argument for such case is, of course, being unable to switch to OMP when necessary. On the other hand, committing fully to

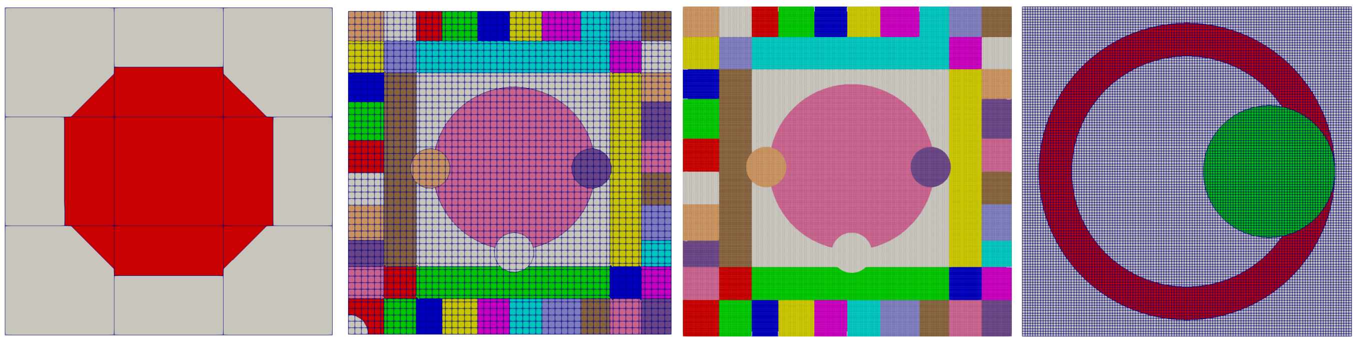
Figure 9: 2D interface reconstruction using moment-of-fluid.

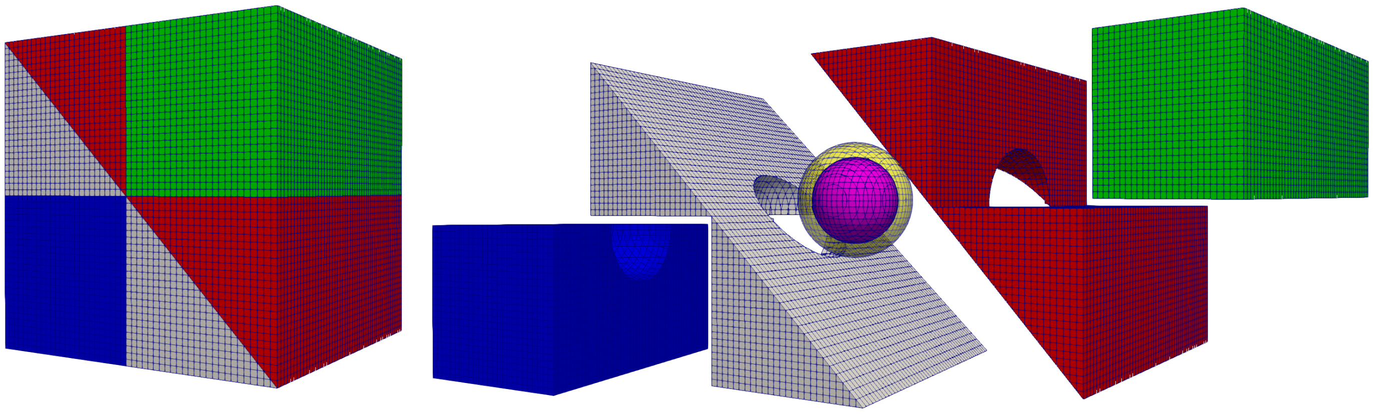
Figure 10: 3D interface reconstruction using moment-of-fluid.

CUDA code should allow more explicit controls on the GPU (i.e. launching kernel size, multiple level of memory access etc.). If the main motivation for the project remains reaching exa-scale, it is inevitable that a compromise solution that optimise both CPU and GPU usage is necessary.

Reference

[1] "Welcome to portage!." `https://laristra.github.io/portage/index.html`. Accessed: 14-08-2018.

[2] "What is thrust?." `https://thrust.github.io/`. Accessed: 14-08-2018.

[3] "List of NVIDIA Graphics Processing Units." `https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units#Volta_series`. Accessed: 14-08-2018.

[4] "Xeon Phi." `https://en.wikipedia.org/wiki/Xeon_Phi`. Accessed: 14-08-2018.

[5] H. T. Ahn and M. Shashkov, "Adaptive moment-of-fluid method," *J. Comput. Phys.*, vol. 228, pp. 2792–2821, May 2009.

[6] C. Hirt and B. Nichols, "Volume of fluid (vof) method for the dynamics of free boundaries," *Journal of Computational Physics*, vol. 39, no. 1, pp. 201 – 225, 1981.

[7] "Sandia Engineering Analysis Code Access System - SEACAS." `https://github.com/gsjaardema/seacas`. Accessed: 24-08-2018.

[8] "CUDA Toolkit - Best Practices Guide." `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`. Accessed: 14-08-2018.