A GPU Accelerated Discontinuous Galerkin Scheme for Advection

Zechariah J. Jibben [*†]

August 23, 2013

Abstract

This report describes the algorithms developed in OpenCL for solving a discontinuous Galerkin scheme to the linear advection equation used in interface transport. A sparse data structure was implemented to take full advantage of parallelism on the GPU, offering a final speedup ranging from 2-60x for a Nvidia Tesla C2050 vs. a 1.9GHz AMD Opteron 6168 running in serial. The algorithms take advantage of coalescence in global memory, as well as avoiding thread divergence.

[1]Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396.

[2]LA-UR-13-26528

[3]Supported by NSF grant NSF-CBET-1054272.

[*]Department of Mechanical and Aerospace Engineering, Arizona State University, Tempe, Arizona, 85287, E-mail: zjibben@asu.edu

[†]XCP-4 Methods and Algorithms Group, Los Alamos National Laboratory

1 Introduction

1.1 Motivation

A pressing problem in engineering is the modeling of fluid interactions involving an immiscible interface. For instance, inside a jet turbine where fuel is dispersed and atomized into air, the quality of the resulting mixture has a direct impact on the overall performance of the engine and pollutant production. Unfortunately, experiments are difficult or impossible to perform at operating conditions, simply because optical access is obstructed by engine structure and the "haze" of liquid droplets surrounds interesting structures. Furthermore, there is no known way of solving the nonlinear governing equations analytically. Therefore, numerical simulations become vital to the design process as well as to deepening our understanding of the physics, and having a high order solver becomes essential to accurately representing the material interface.

Here, the approach to this problem and benefit of utilizing GPUs to handle numerical algorithms is described. Here, sparsity becomes a noteworthy issue. Taking advantage of sparse data structures can be quite beneficial in CPU code. In a GPU implementation, however, it becomes crucial.

This paper begins by describing the problem in question, followed by a basic overview of the level set solution method and discontinuous Galerkin (DG) numerical method. The discussion of the scheme will involve noting the sparse nature of the system, followed by the GPU implementation (including some tricks that help in the process). Finally, results for the overall speedup are shown and discussed.

1.2 Governing Equations

Herrmann [3] gives a good overview for the governing equations of a fluid interaction involving immiscible interfaces. These are the Navier-Stokes' equations, along with a surface tension term $\boldsymbol{T}_\sigma$ that is nonzero only at the interface location $\boldsymbol{x}_f$.

$$\frac{\partial \boldsymbol{u}}{\partial t} + \boldsymbol{u} \cdot \nabla \boldsymbol{u} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla \cdot \left(\mu \left(\nabla \boldsymbol{u} + \nabla^{\mathrm{T}} \boldsymbol{u}\right)\right) + \boldsymbol{g} + \frac{1}{\rho}\boldsymbol{T}_\sigma \tag{1}$$

$$\boldsymbol{T}_\sigma(\boldsymbol{x}) = \sigma\kappa\delta\left(\boldsymbol{x} - \boldsymbol{x}_f\right)\hat{\boldsymbol{n}} \tag{2}$$

Here, $\boldsymbol{u}$ is the velocity, ρ is density, p is pressure, μ is dynamic viscosity, $\boldsymbol{g}$ is the gravitational body force, σ is the surface tension constant, κ is the local surface curvature, and $\hat{\boldsymbol{n}}$ is the local surface normal. As a result of this coupling from surface tension, an accurate method for tracking the phase interface location in such a way that allows us to also calculate the local curvature and normal at high order is vital. The level set method is selected to accomplish this goal.

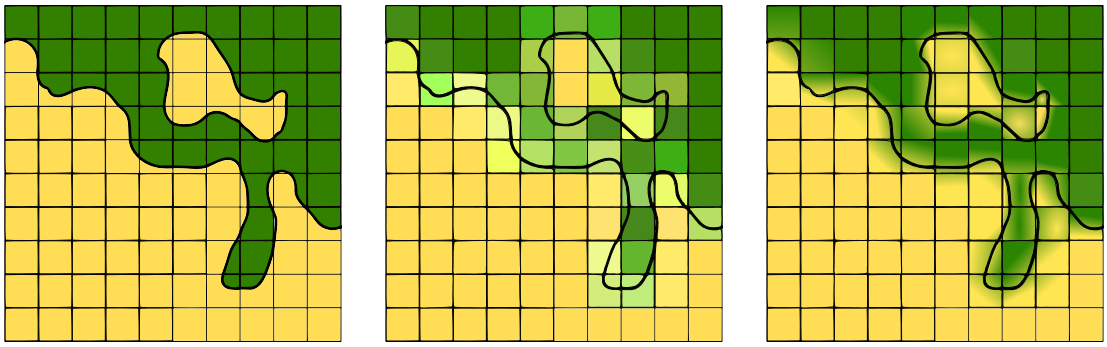
1.3 The Level Set Method

There are several approaches to interface tracking, volume of fluid methods (VOF) and level set methods being the most common. The VOF approach has the benefit of discretely conserving mass, while traditional level sets do not share this property. On the other hand, level sets have the benefit of high order accuracy and having the ability to compute high order normals and curvature. Recently, Olsson and Kreiss [7] and Olsson et al. [8] developed a conservative level set method that treats the level set scalar as a conserved variable, greatly improving the mass conservation of the method.

The concept of level sets is to model the fluid interface, shown in Fig. 1, as the 0.5-isosurface of some scalar function $G(\boldsymbol{x}, t)$. Then, $G > 0.5$ on one side of the interface and $G < 0.5$ on the other. The 0.5-isosurface is then transported via the advection equation, which is found from the fact that the material derivative of $G\left(\boldsymbol{x}_f\right)$ is equal to zero. For incompressible flows, this can be written in conservative form as

$$\frac{\partial G}{\partial t} + \nabla \cdot (G\boldsymbol{u}) = 0. \tag{3}$$

Combining this approach with an arbitrary order discontinuous Galerkin method further improves the accuracy and mass conservation of level set methods.



(a) Fluid Interface (b) Finite Volume (c) Discontinuous Galerkin

Figure 1: Interface Discretization

2 The Discontinuous Galerkin Method

The numerical approach used is an arbitrary-order discontinuous Galerkin method, as described by Cockburn and Shu [1]. It can be thought of as a generalization of the finite volume method, which assigns average values of the solution variables to each cell. The discontinuous Galerkin method, on the other hand, allows sub-cell variation by performing a spectral decomposition of the solution variables in each cell. That is, we project $\boldsymbol{G}$ and $\boldsymbol{u}$ into the basis $\{b_i\}$ as

$$G_{\text{ic}} = \sum_{i=1}^{N_{\text{g}}} g_{i,\text{ic}} b_i, \quad \boldsymbol{u}_{\text{ic}} = \sum_{i=1}^{N_{\text{u}}} \hat{\boldsymbol{u}}_{i,\text{ic}} b_i, \tag{4}$$

where the series is truncated at N_g and N_u terms for $\boldsymbol{G}$ and $\boldsymbol{u}$, respectively (however, for this paper, we take $N_u = N_g$). In this sense, a finite volume method is equivalent to a discontinuous Galerkin method with $N_g = N_u = 1$. The normalized Legendre polynomial basis is selected for their orthonormality property, and they are constructed by performing Gram-Schmidt orthonormalization on the space of 3D monomials $x^\alpha y^\beta z^\gamma$. Then, for a maximum monomial degree k, we find $N_g = (k+1)^3$. It has been shown by LeSaint and Raviart [5] that this method can then formally achieve a $k+1$ convergence rate.

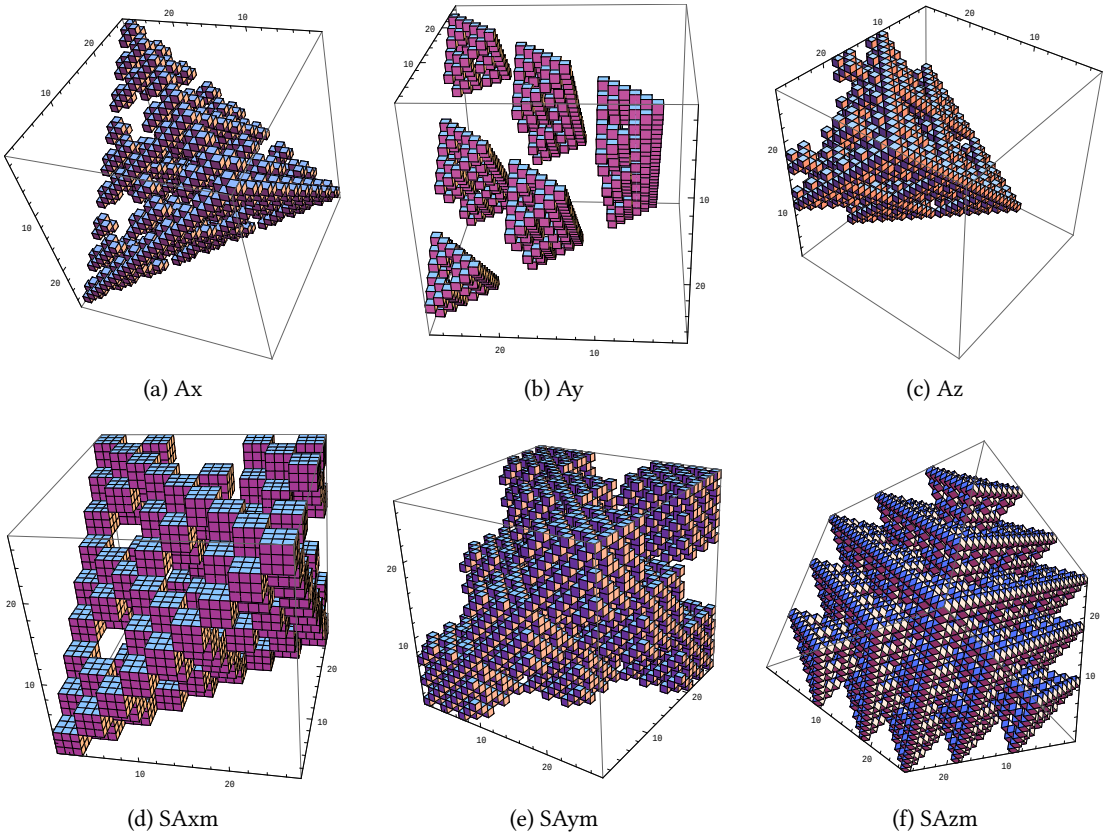
These expansions are then substituted into Eq. (3). By performing an inner product with b_n (integrate over the cell domain Ω), taking advantage of orthonormality, and using the divergence theorem, we arrive at a system of coupled ordinary differential equations describing the time evolution the coefficients g_n for all cells. A simple upwind flux is used to handle integration along cell interfaces, and a $k+1$ order Runge-Kutta (RK) time stepping mechanism is used.

$$\frac{\mathrm{d}g_{n,\text{ic}}}{\mathrm{d}t} = u^j_{k,\text{ic}} g_{i,\text{ic}} \int_\Omega b_k b_i \frac{\partial b_n}{\partial x_j} \,\mathrm{d}V + u^{j,\text{up}}_{k,\text{ic}} g^{\text{up}}_{i,\text{ic}} \int_{\partial\Omega} N_j b^{\text{up}}_k b^{\text{up}}_i b_n \,\mathrm{d}S \tag{5}$$

Note that the two integrals are entirely in terms of our basis functions and the cell domain. These can therefore be pre-computed analytically using symbolic software such as Mathematica, Maple, or SymPy, and stored in a 3D array for reference later. This avoids the use of quadrature, saving computation time. Furthermore, note that the orthogonality of the Legendre polynomial basis produces sparse arrays (see Table 1 and Fig. 2). For reference, the volume integrals are denoted Ax, Ay, and Az, and the "-" face surface integrals are denoted SAxm, SAym, and SAzm. Together, the high number of operations with comparatively few solution variables and the sparsity of the integral arrays make this method ideal for GPU computation.

Table 1: Matrix Fill Fraction

Polynomial Degree	2D Simulation		3D Simulation	
	Volume Integrals	Surface Integrals	Volume Integrals	Surface Integrals
1	12.5%	50.0%	6.25%	25.0%
2	10.6%	40.7%	4.30%	16.6%
3	10.1%	35.9%	3.63%	12.9%
4	9.68%	33.6%	3.25%	11.3%



(a) Ax (b) Ay (c) Az

(d) SAxm (e) SAym (f) SAzm

Figure 2: Sparsity illustration for 2^{nd} degree polynomials in 3D. Cubes are placed at array locations containing nonzero elements. Visualized by Mathematica.

It is highly beneficial to store these arrays in a manner that takes advantage of their sparse structure. A format similar to compressed row storage (CRS) [2] was chosen, allowing the integrals to be stored in 1D arrays along with three corresponding 1D arrays of ints giving nonzero element locations. By doing so, we limit the amount of data that must be sent to the GPU, and make parallelization on the GPU a simpler matter.

3 GPU Programming Model

Using OpenCL terminology, a GPU operates by executing a function called a *kernel* in parallel on a cluster of *work-items*, which are organized into *work-groups* with an associated memory space we call *tiles*. Each work-item then has a global id and local id, and each work-group has a group id. There are several nuances of this model to take advantage of, the most important of which is how workloads are managed. For instance, if work-items inside the same work-group are given drastically different workloads, the entire work-group must wait for the slowest member to complete its task before moving on to the next portion of the problem. As a result, it is highly advantageous to

assign uneven workloads to different work-groups, rather than within work-groups. Although the code is written with GPUs in mind, it can be effectively implemented on other architectures using the OpenCL model.

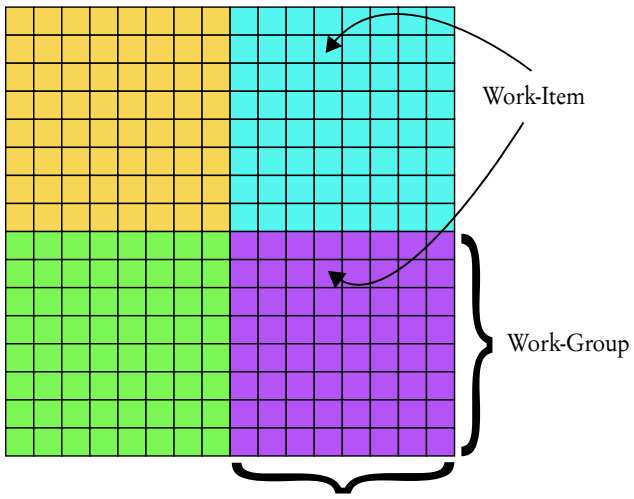


Figure 3: OpenCL Execution Model

To take advantage of this aspect of parallelism, and avoid thread divergence, Eq. (5) is solved by assigning a single $g_{n,\mathrm{ic}}$ to each tile. Then, work-items share the workload of tensor-vector multiplication and summation, avoiding uneven workload distributions between work-items (called thread divergence). This way, if an integral array has a largely zero row n, the entire work-group is given a lighter workload. This allows it to finish earlier and execute a new work-group rather than waiting for individual work-items to finish their work.

In Listing 1, Eq. (5) is considered a series of equations of the form $\Delta g_{n,\mathrm{ic}} \mathrel{+}= \sum_{k=1}^{N_u} u_{k,\mathrm{ic}} \sum_{i=1}^{N_g} g_{i,\mathrm{ic}} Z_{n,k,i}$. Each work-item has its own instance of the variable `my_dg`. Work-items then proceed to sum together a subset of the above equation, that is, products of elements of velocity `u`, level set scalar `g`, and the integral array, denoted `Z` for generalization in the code. Instead of looping over both k and i, we loop over a single integer l that corresponds to nonzero elements of the compressed array `Z`. Two arrays `Zi2` and `Zi3` give the values of k and i associated with l for each iteration. Finally, each work-group has its own value of $\Delta g_{n,\mathrm{ic}}$ to compute, with a unique combination of n and ic. Since each work-group only has one value of n, it only needs to loop through a subset of the integral array `Z`. As a result, it is necessary to pass in two integers `Znstart` and `Znend` that give the bounds of this subsection.

In order to take advantage of memory coalescence and evenly distribute the workload, and hence reduce runtime, the local group of work-items align their access to the array `Z` by their local id number. For example, the work-item with local id 7 will access the array element immediately after the work-item with local id 6 and immediately before the work-item with local id 8.

Listing 1: GPU Implementation

```
1   const uint tiX = get_local_id(0);   // get local work-item id
    const uint ntX = get_local_size(0); // get local work-group size
3
    // initialize summation variables
5   double my_dg=0.0;
    __local double partialsum[TILE_SIZE];
7
    for (uint l=Znstart+tiX; l<Znend; l+=ntX) {
9     uint k = Zi2[l]; uint i = Zi3[l];            // multiply by associated u and g
      my_dg += u[k]*g[i]*Z[l];
11  }
    partialsum[tiX] = term;                        // save private result to local array
13  my_dg = reduction_sum_within_tile(partialsum); // sum the partialsum elements
```

On the next iteration, l is updated with a step size equal to the number of work-items in the work-group. Finally, after each work-item has saved its result in an array stored in local memory, the elements of the array are summed together via a simple parallel reduction routine.

4 Tricks and Workarounds

Programming GPUs comes with the added challenge that OpenCL (and CUDA) currently do not support Fortran [9]. It is, however, possible for Fortran to call C functions. Since OpenCL readily supports host code written in C, functions in C can easily act as a staging area between Fortran and OpenCL. This is done by first giving C access to data allocated in Fortran, which is achieved by creating pointers to that data and sending them as arguments to a C function. Since Fortran natively sends pointers rather than the data itself, this is as simple as writing the first element of an array as the argument to a C function, which C then receives as a pointer to an array contiguous in memory.

Passing more complex data structures, such as arrays nested within arrays of derived data types, is more complicated, but still manageable. Because Fortran pads arrays in such a way that can be difficult to predict, it is simplest to send to C a pointer to the start of each array. This process can be made more compact by defining a derived data type in Fortran containing only a pointer, thereby allowing Fortran to generate an array of pointers (which is not natively available). A pointer to the first element of this array of pointers is then sent to C, which allows C to find the data associated with each variable within an array of derived data types.

Finally, OpenCL does not accept multidimensional arrays. To avoid bulky or obscure code, multidimensional arrays are sent to the GPU as 1D arrays (so long as they are contiguous in memory), and a macro is defined on the OpenCL side to simulate multidimensional behavior.

5 Results

The OpenCL algorithm for DG advection was executed on a Nvidia Tesla C2050 GPU (with a work-group size of 128) and compared to the original algorithm running in serial on a a 1.9GHz AMD Opteron 6186 CPU. Both algorithms take advantage of sparsity and are implemented on equidistant Cartesian meshes in unit sized domains. Verification of the method has been performed for the CPU algorithm previously [4], so the emphasis of this work is limited to compute times and assurance that the CPU and GPU give equivalent results (within 10^5 times machine epsilon at double precision). As such, the test problem is arbitrary. For robustness, the solution variable coefficients are randomized, and activating or deactivating terms of the equation can be used for debugging and additional assurance that each term is being evaluated correctly.

Table 2: Results for Compute Time of One RK-Step

Polynomial Degree	$1/\Delta x$	2D Simulation			3D Simulation		
		CPU time (s)	GPU time (s)	Speedup	CPU time (s)	GPU time (s)	Speedup
1	10	6.37e-4	2.12e-3	0.30x	2.25e-2	6.96e-3	3.23x
	20	2.52e-3	3.11e-3	0.81x	1.79e-1	3.99e-2	4.49x
	40	9.47e-3	6.28e-3	1.51x	6.18e-1	2.83e-1	2.18x
2	10	2.45e-3	2.45e-3	1.00x	3.24e-1	2.56e-2	12.7x
	20	9.63e-3	4.57e-3	2.11x	1.18	1.41e-1	8.37x
	40	3.85e-2	1.15e-2	3.35x	8.82	1.05	8.40x
3	10	8.81e-3	2.87e-3	3.07x	1.08	8.30e-2	13.0x
	20	3.38e-2	6.52e-3	5.18x	8.34	6.20e-1	13.5x
	40	1.34e-1	1.89e-2	7.09x	6.67e+1	4.78	14.0x
4	10	3.47e-2	4.34e-3	8.00x	1.15e+1	4.16e-1	27.6x
	20	1.38e-1	1.03e-2	13.4x	1.32e+2	3.03	43.6x
	40	3.92e-1	3.06e-2	12.8x	1.36e+3	2.40e+1	56.7x

These tests produce several interesting trends. First, low degree polynomials show little benefit from the GPU, and sometimes even slower runtimes. This is simply because of the parallelization scheme, where we delegate work

for a single $g_{n,\text{ic}}$ coefficient across a work-group. If there are not enough terms for all of the work-items, we lose much of the benefit of parallelization on the GPU, since some of the threads then do no work. One way to remedy this in practice is to use smaller work-group sizes when dealing with smaller polynomials. However, this solution has limitations since GPUs are most efficient when the work-group size is a multiple of 32 [6]. A similar drawback arises if sparsity is not exploited, where the GPU sees a ∼2x slow-down for 3$^{\text{rd}}$ order polynomials. This results from parallelizing along the tensor multiplication loop, where many threads end up multiplying zeros together and appending them to a sum, again wasting effort.

Second, the data indicates the GPU is increasingly advantageous as it is given more work. As the degrees of freedom and number of operations increases, whether from refining the grid or increasing the number of polynomials, the speedup increases. This reflects the streaming memory model on the GPU, where floating-point operations are almost free.

Using OpenCL event timers, we can further probe the GPU runtime to investigate the execution time and rank routines by their overall cost. This is shown in Table 3 for the degree 3 polynomial, 40x40x40 grid case. As a perhaps

Table 3: GPU Event Timing

Event	Time (ms)
Kernel Create	0.1628
Data Send	336.9
Compute	4763
Data Receive	20.14
Total	4784

unexpected result, the computations overwhelmingly dominate the execution time. In other applications, memory transfer operations take up a significant portion of the runtime. However, this case involves a high work to data ratio, since the method involves a high number of operations relative to the amount of data transferred. As a result, optimizations that focus on decreasing compute time are more beneficial than memory optimizations, contrary to the usual case for GPU algorithms.

Note: these times may overlap, so the total compute time is not necessarily the sum of event times.

6 Concluding Remarks

This work has demonstrated that taking advantage of sparsity, whenever possible, is crucial to developing efficient algorithms, especially on the GPU. Furthermore, an overall speedup ranging from 2-60x for GPUs over CPUs was found, advocating the applicability and benefit of GPUs in numerical algorithms, especially for independent segments of code that benefit from parallelism. These speedups indicated that more work given to the GPU results in more speedup, especially when increasing the number of basis functions used. This compliments the results in [4], where it was found that the discontinuous Galerkin conservative level set method is more effective when more basis functions are used. Therefore, accelerating that method via GPUs reaps benefits from multiple angles, making it an excellent candidate for high order interface tracking and modeling atomization.

Future work to further develop this approach would involve an improved implementation, for instance, ensuring memory alignment. Porting more segments of code to OpenCL would also be valuable. This is especially true when there is a direct benefit from GPU architectures, but is also true for code that would not be improved by parallelism, since it avoids passing data to and from the GPU as much as possible. For instance, it may be beneficial to handle ghost cell updates on the GPU, allowing multiple time steps to be executed before returning to the CPU. Finally, to complete the conservative level set method it is necessary to implement a similar scheme for reinitialization, Eq. (6).

$$\frac{\partial G}{\partial \tau} + \nabla \cdot (G\,(1 - G)\,\hat{\boldsymbol{n}}) = \nabla \cdot (\varepsilon\,(\nabla G \cdot \hat{\boldsymbol{n}})\,\hat{\boldsymbol{n}}) \tag{6}$$

Reinitialization is best described as a nonlinear companion to the advection equation solved in pseudo-time which greatly improves mass conservation.

Acknowledgments

I would like to acknowledge the support of my mentor, Bob Robey, for his invaluable advice and teaching through the summer. I would also like to thank Scott Runnels, who organized the Los Alamos National Laboratory Computational Physics Student Summer Workshop 2013, where this work was performed. Finally, I would like to thank the teams managing the Darwin CCS-7 experimental cluster and the Moonlight ASC cluster, where these algorithms were tested and benchmarked.

References

[1] B. Cockburn and C.-W. Shu. "Runge–Kutta discontinuous Galerkin methods for convection-dominated problems". *J. Sci. Comput.* 16 (2001), pp. 173–261.

[2] I. Duff, R. Grimes, and J. Lewis. *User's Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)*. 1992.

[3] M. Herrmann. "A balanced force refined level set grid method for two-phase flows on unstructured flow solver grids". *J. Comput. Phys.* 227 (2008), pp. 2674–2706.

[4] Z. Jibben and M. Herrmann. "An arbitrary high-order conservative level set Runge-Kutta discontinuous Galerkin method for capturing interfaces". *ILASS Americas 25th Annual Conference on Liquid Atomization and Spray Systems* (2013).

[5] P. LeSaint and P. A. Raviart. "On a finite element method for solving the neutron transport equation". In: *Mathematical Aspects of Finite Elements in Partial Differential Equations*. Ed. by C. de Boor. Academic Press, NY, 1974, pp. 89–123.

[6] *NVIDIA OpenCL Best Practices Guide*. ver. 1.0. NVIDIA Corporation. 2009.

[7] E. Olsson and G. Kreiss. "A conservative level set method for two phase flow". *J. Comput. Phys.* 210 (2005), pp. 225–246.

[8] E. Olsson, G. Kreiss, and S. Zahedi. "A conservative level set method for two phase flow II". *J. Comput. Phys.* 225 (2007), pp. 785–807.

[9] *The OpenCL Specification*. ver. 1.2. Khronos Group, Inc. 2011.