SIAM J. SCI. COMPUT.
Vol. 35, No. 1, pp. A212–A230

SMOOTHNESS-INCREASING ACCURACY-CONSERVING FILTERS FOR DISCONTINUOUS GALERKIN SOLUTIONS OVER UNSTRUCTURED TRIANGULAR MESHES*

HANIEH MIRZAEE†, JAMES KING‡, JENNIFER K. RYAN§, AND ROBERT M. KIRBY‡

Abstract. The discontinuous Galerkin (DG) method has very quickly found utility in such diverse applications as computational solid mechanics, fluid mechanics, acoustics, and electromagnetics. The DG methodology merely requires weak constraints on the fluxes between elements. This feature provides a flexibility which is difficult to match with conventional continuous Galerkin methods. However, allowing discontinuity between element interfaces can in turn be problematic during simulation postprocessing, such as in visualization. Consequently, smoothness-increasing accuracy-conserving (SIAC) filters were proposed in [M. Steffen et al., *IEEE Trans. Vis. Comput. Graph.*, 14 (2008), pp. 680–692, D. Walfisch et al., *J. Sci. Comput.*, 38 (2009), pp. 164–184] as a means of introducing continuity at element interfaces while maintaining the order of accuracy of the original input DG solution. Although the DG methodology can be applied to arbitrary triangulations, the typical application of SIAC filters has been to DG solutions obtained over translation invariant meshes such as structured quadrilaterals and triangles. As the assumption of any sort of regularity including the translation invariance of the mesh is a hindrance towards making the SIAC filter applicable to real life simulations, we demonstrate in this paper for the first time the behavior and complexity of the computational extension of this filtering technique to fully *unstructured* tessellations. We consider different types of unstructured triangulations and show that it is indeed possible to get reduced errors and improved smoothness through a proper choice of kernel scaling. These results are promising, as they pave the way towards a more generalized SIAC filtering technique.

Key words. discontinuous Galerkin, SIAC filtering, accuracy enhancement, unstructured tessellations, triangle meshes, postprocessing

AMS subject classification. 65M60

DOI. 10.1137/120874059

1. Introduction and motivation.

The discontinuous Galerkin (DG) method has very quickly found utility in such diverse applications as computational solid mechanics, fluid mechanics, acoustics, and electromagnetics. It allows for a dual path to convergence through both elemental h and polynomial p refinement. Moreover, unlike the classic continuous Galerkin FEM, which seeks approximations that are piecewise continuous, the DG methodology merely requires weak constraints on the fluxes be-

*Submitted to the journal's Methods and Algorithms for Scientific Computing section April 18, 2012; accepted for publication (in revised form) September 14, 2012; published electronically January 10, 2013. NVIDIA hardware support was provided through a Faculty Equipment Fellowship (2010).

http://www.siam.org/journals/sisc/35-1/87405.html

†Corresponding author. School of Computing, University of Utah, Salt Lake City, UT 84112 (mirzaee@cs.utah.edu). This author's work was supported in part by the Air Force Office of Scientific Research (AFOSR), Computational Mathematics Program (Program Manager: Dr. Fariba Fahroo), under grant FA9550-08-1-0156, and by the Department of Energy (DOE NET DE-EE0004449).

‡School of Computing, University of Utah, Salt Lake City, UT 84112 (kirby@cs.utah.edu, jsking2@gmail.com). The work of these authors was supported in part by the Air Force Office of Scientific Research (AFOSR), Computational Mathematics Program (Program Manager: Dr. Fariba Fahroo), under grant FA9550-08-1-0156, and by the Department of Energy (DOE NET DE-EE0004449).

§Delft Institute of Applied Mathematics, Delft University of Technology, 2628 CD Delft, The Netherlands (J.K.Ryan@tudelft.nl). This author's work was supported by the Air Force Office of Scientific Research (AFOSR), Air Force Material Command, USAF, under grant FA8655-09-1-3055.

tween elements. This feature provides a flexibility which is difficult to match with conventional continuous Galerkin methods. However, allowing discontinuity between element interfaces can in turn be problematic during simulation postprocessing, such as in visualization, where there is often an implicit assumption that the field upon which the postprocessing methodology is acting is smooth. A class of postprocessing techniques was introduced in [3] and continued in [7, 8], with an application to uniform quadrilateral meshes, as a means of gaining increased accuracy from DG solutions by performing convolution of a spline-based kernel against the DG field. As a natural consequence of convolution, these filters also increased the smoothness of the output solution. Building upon these concepts, smoothness-increasing accuracy-conserving (SIAC) filters were proposed in [21, 24] as a means of introducing continuity at element interfaces while maintaining the order of accuracy of the original input DG solution.

Although the DG methodology can be applied to arbitrary triangulations, the typical application of SIAC filters with mathematically proven properties has been to DG solutions obtained over translation invariant meshes. One way of producing such meshes is to take some base tessellation and repeat integer multiples of it [1, 2]. In an attempt to make the SIAC filter applicable to arbitrary tessellations, Curtis et al. [9] proposed a computational extension of this filtering technique to smoothly varying meshes as well as random meshes. They provided numerical results in one dimension which confirmed the accuracy enhancement of $2k + 1$, proved in [8], for smoothly varying meshes when a kernel scaling equal to the largest element size was used. For random meshes there was no clear order improvement, which may have been due to an incorrect kernel scaling. To further expand the applicability of the SIAC filter, the authors in [14] extended the postprocessing results, both theoretically and numerically, to structured triangular meshes. However, the outlook for triangular meshes was actually much better than that presented in [14]. Indeed, the order improvement was not clear for filtering over a Union Jack mesh when a scaling $H = h$ equal to the uniform mesh spacing was used. The authors in [12] revisited that case, as well as a Chevron triangular mesh, and demonstrated that it is indeed possible to obtain superconvergence of order $2k + 1$ for these mesh types. They also introduced theoretical proof as well as numerical examples that these results could be extended to adaptive meshes whose elements are defined by $\frac{1}{\ell}H$, where H represents the minimum scaling for the convolution kernel in the SIAC filter and is equal to the largest element size. ℓ is a multi-index of a dimension equal to the number of elements in one direction. For example, if $\ell = [\ell_1, \ldots, \ell_N]$, where N is the number of elements in one direction, then the size of the element i will be $\frac{1}{\ell_i}H$.

As the assumption of any sort of regularity, including the translation invariance of the mesh, is a hindrance towards making the SIAC filter applicable to real life simulations, we herein demonstrate for the first time the mathematical behavior and computational complexity of the extension of this filter to *unstructured* tessellations. We consider four examples: a simple Delaunay triangulation, a Delaunay triangulation with obvious changes in element sizes where a refinement ratio of $\frac{1}{2}$ has been applied, a Delaunay triangulation with splitting, and a stretched (anisotropic) triangulation. We show that it is indeed possible to obtain reduced errors and increased smoothness through a proper choice of kernel scaling. These results are promising, as they pave the way towards a more generalized SIAC filtering technique that could be used for arbitrary unstructured tessellations. Additionally, we demonstrate how to achieve up to an 18× speedup over traditional CPU implementations when performing our filtering task on graphics processing units (GPUs). This gain in performance

is realized by taking advantage of the streaming architecture of these multiprocessors.

We proceed in this paper by providing the basics of DG methods over triangulations followed by a brief review of the SIAC filter. In section 2, we present the implementation details of the postprocessor for unstructured triangular meshes. There, we discuss the Sutherland–Hodgman clipping algorithm used to compute the mesh-kernel intersections as well as CUDA implementations of the postprocessor for GPUs. In section 3, we give numerical results confirming the usefulness of our SIAC filter for the proposed triangulated meshes. Finally, section 4 concludes this paper. We further note that throughout this paper, we use the terms *filtering* and *postprocessing* interchangeably.

1.1. The DG formulation for triangular mesh structures. The DG method makes use of the same function space as the continuous method, but with relaxed continuity at the interelement boundaries. It was first introduced by Reed and Hill [17] for the solution of the neutron transport equation, and its history and recent development have been reviewed by Cockburn, Karniadakis, and Shu [6] and Cockburn [5]. The essential idea of the method is derived from the fact that basis functions can be chosen such that either the field variable, or its derivatives or generally both, are considered discontinuous across the element boundaries, while the computational domain continuity is maintained. The DG scheme has the advantages of both the finite volume and the finite element methods, in that it can be effectively used in convection-dominant applications, while maintaining geometric flexibility and higher local approximations through the use of higher-order elements. This feature makes it uniquely useful for computational dynamics and heat transfer. Because of the local nature of a discontinuous formulation, no global matrix needs to be assembled; thus, this reduces the demand on the in-core memory. The effects of the boundary conditions on the interior field distributions then gradually propagate through the domain via the element-by-element connections. This is another important feature that makes this method useful for fluid flow calculations.

In this paper we consider accuracy enhancement of numerical solutions to two-dimensional linear hyperbolic equations of the form

$$\begin{aligned} u_t + \sum_{i=1}^{2} \frac{\partial}{\partial x_i}(A_i(\mathbf{x})u) = 0, &\qquad \mathbf{x} \in \Omega \times [0,T], \\ u(\mathbf{x},0) = u_o(\mathbf{x}), &\qquad \mathbf{x} \in \Omega, \end{aligned} \tag{1}$$

where $\Omega \in \mathbb{R}^2$, and $A_i(\mathbf{x})$, $i = 1, 2$, is bounded in the L^∞-norm. We also assume smooth initial conditions are given along with periodic boundary conditions. We consider only the discretization of this equation in space. For full discretization of this equation, the reader should consult [4].

To discretize (1) in space using a DG method, we first triangulate the domain Ω such that it consists of N nonoverlapping triangular elements. We denote such a triangulation by Ω_h. We then seek a discontinuous approximate solution u_h which, in each element τ_e of the triangulation, belongs to the space V_h. There is no restriction on how to choose the space V_h, though a typical choice is the space of polynomials of degree at most k, i.e., $\mathbb{P}^k(\tau_e)$. We multiply (1) by a test function $v_h \in \mathbf{V}_h$ and substitute the approximate solution u_h to get the following weak form over τ_e:

$$\int_{\tau_e} \frac{\partial u_h}{\partial t} v_h \, d\mathbf{x} - \sum_{i=1}^{2} \int_{\tau_e} f_i(\mathbf{x},t) \frac{\partial v_h}{\partial x_i} \, d\mathbf{x} + \sum_{i=1}^{2} \int_{\partial \tau_e} \hat{f}_i \hat{n}_i v_h \, ds = 0, \tag{2}$$

where $f_i(\mathbf{x},t) = A_i(\mathbf{x})u_h(\mathbf{x},t)$ for $i = 1,2$ is the flux function, $\partial\tau_e$ is the boundary of the element τ_e, and $\hat{n}_i$ denotes the unit outward normal to the element boundary in the ith direction. Notice that the flux is multiply defined at element interfaces, and therefore we impose the definition that $\hat{f}_i\hat{n}_i = h(u_h(\mathbf{x}^{exterior},t), u_h(\mathbf{x}^{interior},t))$ is a consistent two-point monotone Lipschitz flux, as in [4].

The approximate solution, u_h, within a triangular element is given by

$$u_h(\mathbf{x},t) = \sum_{p=0}^{k}\sum_{q=0}^{k-p} u_{\tau_e(t)}^{(p,q)}\phi^{(p,q)}(\mathbf{x}), \qquad \mathbf{x}\in\tau_e, \tag{3}$$

where $u_{\tau_e}^{(p,q)}(t)$ represents the expansion coefficients and $\phi^{(p,q)}(\mathbf{x})$ are the given basis functions. We plug this approximation into (2), and we then solve for $u_{\tau_e}^{(p,q)}(t)$. We note here that for our DG solvers, we used the NEKTAR++ implementation given in [11] and available online from http://www.nektar.info.

1.2. A brief review of SIAC filters. SIAC filters were first introduced in [3] and later applied to DG solutions of linear hyperbolic equations in [7, 8]. The filtering technique was extended to a broader set of applications, such as being used for filtering within streamline visualization algorithms in [20, 21, 24]. Given a sufficiently smooth exact solution, the rate of convergence of a DG approximation of degree k is $k+\frac{1}{2}$ in general and $k+1$ for certain mesh structures (see [18, 4]). However, after convolving the approximation against a filter constructed from a linear combination of B-splines of order $k+1$, we can improve the order of convergence up to $2k+1$ as long as the necessary negative-order norm estimates can be proven. Additionally, we filter out the oscillations in the error and restore the $\mathcal{C}^{k-1}$-continuity at the element interfaces. The provability of these properties relies on the translation invariant nature of the mesh. Here we present a brief introduction to this postprocessing technique. For a more detailed discussion of the mathematics of the postprocessor, see [3, 8, 20].

The postprocessor itself is simply the DG solution at the final time T convolved against a linear combination of B-splines. That is, in one dimension,

$$u^\star(x) = \frac{1}{h}\int_{-\infty}^{\infty} K^{r+1,k+1}\left(\frac{y-x}{h}\right)u_h(y)dy, \tag{4}$$

where $u^\star$ is the postprocessed solution, h is the characteristic length, and

$$K^{r+1,k+1}(x) = \sum_{\gamma=0}^{r} c_\gamma^{r+1,k+1}\psi^{(k+1)}(x-x_\gamma) \tag{5}$$

is the convolution kernel. $\psi^{(k+1)}$ is the B-spline of order $k+1$, and $c_\gamma^{r+1,k+1}$ represent the kernel coefficients and are chosen such that the kernel reproduces polynomials of degree up to r. x_γ represent the positions of the kernel nodes and are defined by

$$x_\gamma = -\frac{r}{2}+\gamma, \qquad \gamma = 0,\ldots,r, \tag{6}$$

where $r = 2k$. The kernel in (5) has a *symmetric* form that can be used for postprocessing in the interior of the domain. To postprocess near boundaries or shocks we need to use a *one-sided* form of the kernel that requires information only from one side of the boundary or shock. In general, we can generate such a kernel by shifting the positions of the kernel nodes to one side. For more information, consult [19, 23, 13].

We note that in this work we always consider periodic meshes and solutions, and hence only the symmetric kernel is implemented. However, the ideas presented here can easily adapt to the one-sided kernel.

We additionally note that the superscript $r+1, k+1$ in (5) typically represents the number of kernel nodes as well as the B-spline order. In the following discussions we shall drop this superscript for the sake of a less cluttered notation and continue by investigating how the postprocessed solution given in (4) can be implemented in a computationally efficient manner.

2. SIAC filters for unstructured triangular meshes. In this section we provide the implementation details of the postprocessor over unstructured triangular meshes. The implementation discussed in this section is used to produce the results given in section 3.

In [14], we thoroughly discussed the extension of the SIAC filter to structured triangular meshes. Here, we simply take the existing implementation of the SIAC filter and apply the same ideas to unstructured triangular meshes.

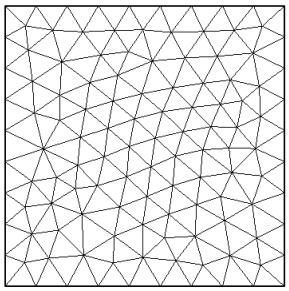


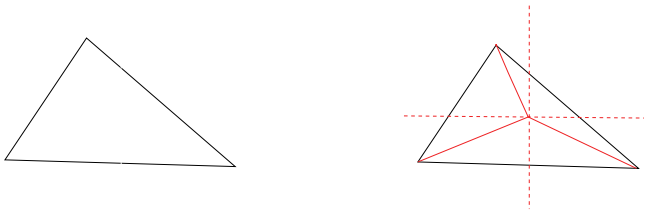
FIG. 1. *A sample unstructured triangular mesh.*

The postprocessor takes as input an array of the polynomial modes used in the DG method and produces the values of the postprocessed solution at a set of specified evaluation points. We assume these evaluation points correspond with specific quadrature points which can be used at the end of the simulation for such things as error calculations. We examine how to calculate the postprocessed value at a single evaluation point. Postprocessing of the entire domain is obtained by repeating the same procedure for all the evaluation points. Let us consider the case of a DG solution produced over an unstructured triangular mesh shown in Figure 1. We note that in two dimensions, the convolution kernel is the tensor product of one-dimensional kernels. Therefore, the postprocessed solution at $(x, y) \in T_i$ becomes

$$u^\star(x,y) = \frac{1}{h_{x_1} h_{x_2}} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} K\left(\frac{x_1 - x}{h_{x_1}}\right) K\left(\frac{x_2 - y}{h_{x_2}}\right) u_h(x_1, x_2) dx_1 dx_2, \tag{7}$$

where T_i is a triangular element, u_h is our approximate DG solution, and we have denoted the two-dimensional coordinate system as (x_1, x_2). The main difference in the implementation of (7) for unstructured triangulations versus structured meshes is in the choices of h_{x_1} and h_{x_2} used to scale the kernel in the x_1 and x_2 directions, respectively. In [12], it was shown that $h_{x_1} = H_{x_1}$ and $h_{x_2} = H_{x_2}$, where H_{x_1} and H_{x_2} represent the minimum scaling for translation invariance of the mesh. For instance, for a uniform quadrilateral mesh, $h_{x_1} = h_{x_2} = H$ and is simply the uniform mesh spacing. The authors in [12] also provided mathematical proofs demonstrating that

for an adaptive mesh whose elements are of size $h = \frac{1}{\ell}H$, ℓ a multi-index as given in section 1, and H the size of the largest element, we can obtain the correct convergence order and smoothness enhancement in the postprocessed results by choosing H as the scaling parameter. However, for an unstructured triangulation, neither the translation invariant property nor any interelement relation as in the adaptive mesh case holds. Therefore, we require another mechanism to find the proper scaling parameter. As it is not straightforward to speculate as to the width of the kernel support (and hence the corresponding neighboring information) necessary to generate accuracy conservation and smoothness enhancement, we will investigate how different choices of the scaling parameter lead to different postprocessing results. We start by considering a scaling equal to the largest side of all the triangular elements which we refer to as H. We then continue by investigating the impact of a kernel scaling smaller or larger than H on postprocessing DG solutions. In particular, in section 3 we present postprocessing results using $0.5H$, $0.75H$, H, $1.5H$, and $2H$ as kernel scaling values.



(a) Triangular element (b) Integration regions

FIG. 2. *Demonstration of integration regions resulting from the kernel-mesh intersection. Dashed lines represent the kernel breaks. Solid red lines represent a triangulation of the integration regions.*

To calculate the integral involved in the postprocessed solution in (7) exactly, we need to decompose the triangular elements that are covered by the kernel support into subelements that respect the kernel knots (which we also refer to as kernel breaks); the resulting integral is calculated as the summation of the integrals over each subelement. Figure 2 depicts a possible kernel-mesh intersection for a sample triangular element of the unstructured triangular mesh shown in Figure 1. As is shown in Figure 2(b), we divide the triangular region into subregions over which there is no break in regularity. Furthermore, we choose to triangulate these subregions for ease of implementation. Choosing H as the kernel scaling value in each direction, we can therefore rewrite (7) as

$$\begin{aligned} u^\star(x,y) &= \frac{1}{H^2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} K\left(\frac{x_1-x}{H}\right) K\left(\frac{x_2-y}{H}\right) u_h(x_1,x_2)dx_1dx_2 \\ &= \frac{1}{H^2}\sum_{T_j\in \mathrm{Supp}\{K\}} \int\int_{T_j} K\left(\frac{x_1-x}{H}\right) K\left(\frac{x_2-y}{H}\right) u_h(x_1,x_2)dx_1dx_2, \end{aligned} \tag{8}$$

where we have used the compact support property of the kernel to transform the infinite integral to finite local sums over elements. The extent of the kernel or $\mathrm{Supp}\{K\}$ is given by $(3k+1)H$ in each direction, where k is the degree of the polynomial approximation. Therefore, there will be $m(3k+1)^2H^2$ triangles covered by the kernel support, where m is an integer that depends on the unstructured mesh and the size of triangles relative to the scaling parameter H. Each of the integrals over a triangle

T_j then becomes

$$\begin{aligned}&\int\int_{T_j} K\left(\frac{x_1-x}{H}\right) K\left(\frac{x_2-y}{H}\right) u_h(x_1,x_2)dx_1dx_2\\ &=\sum_{n=0}^{N}\int\int_{\tau_n} K\left(\frac{x_1-x}{H}\right) K\left(\frac{x_2-y}{H}\right) u_h(x_1,x_2)dx_1dx_2,\end{aligned} \tag{9}$$

where N is the total number of triangular subregions formed in the triangular element T_j as the result of kernel-mesh intersection.

We note here that to evaluate the integrals in (9) to machine precision exactly, we first map via a Duffy transformation the triangular region τ_n to the standard triangular element defined as

$$T_{st} = \{(\xi_1,\xi_2)| -1 \le \xi_1,\xi_2; \xi_1+\xi_2 \le 0\}, \tag{10}$$

and then we apply Gaussian quadrature rules with the required number of quadrature points to integrate polynomials of degree $3k$ in each direction. For example, when using the Gauss–Jacobi rule, we need a total of $\left\lceil \frac{3k+1}{2} \right\rceil^2$ quadrature points. For more information regarding the Gaussian quadrature and the various mappings involved in the integration, consult [14, 15, 11].

We further add that in order to find the footprint of the kernel on the DG mesh, we first lay a regular grid over our unstructured mesh. Each regular grid element contains the information of the triangles that intersect with it. In this way, we can easily find the extent of the kernel support on this regular grid and consequently compute the integration regions by solving a geometric intersection problem. For this we apply the *Sutherland–Hodgman* clipping algorithm from computer graphics. In the following section we provide a brief overview of this algorithm.

2.1. Sutherland–Hodgman clipping algorithm. The Sutherland–Hodgman clipping algorithm finds the polygon that is the intersection between an arbitrary polygon (the *subject polygon*) and a convex polygon (the *clip polygon*) [22]. Figure 3 depicts a sample kernel-mesh overlap. We remind the reader that the convolution kernel used in the postprocessing algorithm is a linear combination of B-splines (see (5)) and therefore is a piecewise polynomial. Moreover, in two dimensions it is the tensor product of one-dimensional kernels. Consequently, for implementation purposes we will think of the two-dimensional kernel as an array of squares, as depicted with red dashed lines in Figure 3(left). Thereby, the problem of finding the integration regions becomes the problem of finding the intersection areas between each square of the kernel array (the clip polygon) and the triangular elements (the subject polygons) covered by the kernel support. Furthermore, it is clear that both the clip polygon and the subject polygons are convex.

To find the intersection area between a square of the kernel and a triangular element (Figure 3(right)), we follow the Sutherland–Hodgman clipping algorithm and use a divide-and-conquer strategy. First, we clip the polygon (in our case the triangular element) against the left clipping boundary (left side of the square in the kernel array). The resulting partially clipped polygon is then clipped against the top boundary, and then the process is repeated against the two remaining boundaries, as shown in Figure 4.

To clip against one boundary, the algorithm loops through all polygon vertices. At each step, it considers two of the vertices that we denote as *previous* and *current*.

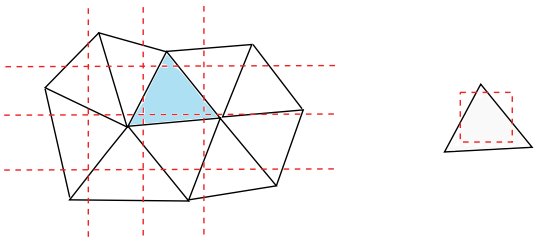


FIG. 3. *A sample kernel-mesh overlap (left). Dashed red lines represent the two-dimensional kernel as an array of squares. On the right, the intersection between a square of the kernel and a triangular element is shown.*

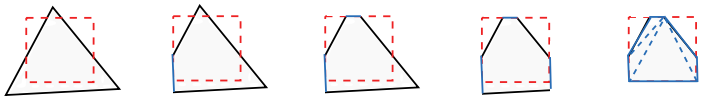


FIG. 4. *The Sutherland–Hodgman clipping. The final intersection area is triangulated for ease of implementation. Dashed red lines represent a square of the kernel, solid black lines represent the triangular DG element, solid blue lines represent the clipped area at each stage of the Sutherland–Hodgman algorithm, and dashed blue lines represent the final triangulation of the integration region. Color is distinquishable only in the online version.*

First, it determines whether these vertices are inside or outside the clipping boundary. This, of course, is a matter of comparing the horizontal or vertical position to the boundary's position. We then apply the following simple rules:

1. if the previous vertex and the current vertex are both inside the clipping boundary, output the current vertex;
2. if the previous vertex is inside the clipping boundary and the current vertex is outside the clipping boundary, output the intersection point of the corresponding edge with the clipping boundary;
3. if the previous vertex and the current vertex are both outside the clipping boundary, output nothing;
4. if the previous vertex is outside the clipping boundary and the current vertex in inside the clipping boundary, output the intersection point of the corresponding edge with the clipping boundary.

Following this procedure we obtain a new polygon, clipped against one boundary and ready to be clipped against the next boundary. Furthermore, we triangulate the resulting clipped area for ease of implementation of the quadrature rules (Figure 4(right)). As the final triangulation is merely used for performing numerical quadrature, there is no rigorous requirement on the triangle element quality. We require only well-formed (i.e., valid) triangles over which the approximate solution and the kernel can be evaluated and their product integrated.

2.2. CUDA implementation of the postprocessor. As we discussed in the previous sections, the postprocessor algorithm divides the triangular mesh into regularly spaced grid patches. The triangles in the intersected regular grid patches are checked for intersections with the kernel stencil using the Sutherland–Hodgman algorithm. The integrations over those intersected regions are computed and used to calculate the convolution operator for that evaluation point. In the following sections

we provide details on different aspects of an efficient GPU implementation of the postprocessor for unstructured tessellations.

2.2.1. Efficient computing on GPUs. In [12], we discussed how postprocessing of DG solutions on quadrilateral meshes was amenable to GPU parallelization. The scalability of the GPU stream processing architecture lends itself well to DG solutions and postprocessing. We continued this line of work with a GPU implementation of a DG postprocessor for unstructured triangular meshes.

Stream processing simplifies hardware by restricting the parallel computations that can be performed. Given a set of data (a stream), stream processors apply a series of operations (kernel functions) to each element in the data stream, often with one kernel function being applied to all elements in the stream (uniform streaming). This paradigm allows kernel functions to be pipelined, and local on-chip memory and data transmission are reused to minimize external memory needs. On modern GPUs each processing unit is termed a core. Sets of cores work together by performing identical sets of operations on a stream. This design paradigm allows for memory reuse between cores and reduces the need for cache hierarchies, which allows more space on each chip to be allocated to processing units [16]. Moreover, it also allows for very high bandwidth between cores and memory used for data streams.

Most current streaming processors rely on an SIMD (single-instruction multiple-data) programming model in which a single kernel is applied to a large collection of independent data streams. Parallel threads, operating on these streams, are logically grouped together with each group operating in a synchronous fashion. SIMD architectures perform at peak capacity when they are without thread divergence and when they achieve coalesced memory accesses. Thread divergence occurs when branching causes some threads in a group to execute instructions differing from the rest. A divergence in the logic forces other threads in that group to wait while the divergent threads execute, which reduces the level of parallelism achieved. Coalesced memory accesses occur when all the memory accesses performed by a thread group are to contiguous locations in memory. When this does not occur, more memory fetches are required to retrieve the data, effectively reducing memory bandwidth. Achieving coalesced memory accesses is easy for structured meshes but becomes much more challenging for unstructured meshes.

2.2.2. Suitability of the problem for the GPU. In theory, postprocessing of DG solutions should be highly amenable to parallelization since each evaluation point can be processed independently of the other evaluation points. However, the Sutherland–Hodgman algorithm is not the most suitable application for SIMD parallelization on the GPU. Thread divergence is very likely to occur in intersection processing due to the numerous branching logics in the Sutherland–Hodgman algorithm, as discussed in section 2.1. This divergence in thread execution may lead to suboptimal performance on the GPU. The polygon clipping that takes place within the Sutherland–Hodgman algorithm occurs at irregularly spaced intervals on an unstructured mesh. The GPU architecture relies on SIMD parallelism to gain efficiency, and this irregularity causes divergence between threads that are operating synchronously. This leads to noticeably poorer performance for unstructured meshes versus that of structured meshes. The nature of unstructured meshes also leads to nonconsecutive memory accesses during processing which reduces the bandwidth capacity for SIMD computations. To reduce thread divergence we implemented a modified regular grid binning for the GPU. We assign each triangle in the mesh to a unique regular grid element which allows us to check each triangle in the intersected regular

grid elements without verifying whether the triangle has been previously processed. This method avoids costly branching in the GPU code.

While the Sutherland–Hodgman algorithm is ill suited for the GPU, the numerical integrations performed over the intersection regions are more amenable to parallelization. The integration component of our processing scheme gained a 12×–14× speedup over the CPU version, while the Sutherland–Hodgman algorithm gained only a 4×–5× speedup over the CPU implementation. The memory allocation of intersections when performing the Sutherland–Hodgman algorithm is especially challenging due to the fact that the number of triangle/stencil intersections is not known until the computations are performed. For each evaluation point a convolution kernel is centered at that point and intersections between the underlying mesh and kernel are calculated. Performing this on an unstructured mesh may result in an arbitrary number of intersections per evaluation point. Memory for these intersections must be allocated ahead of time, which makes it difficult to efficiently allocate intersections to a block of memory when computing them in parallel. In Figure 5 we illustrate how intersections for an unstructured mesh might get mapped into memory. This diagram shows how some threads may use up all of their given memory space while others use only a small portion, leading to inefficient memory utilization.

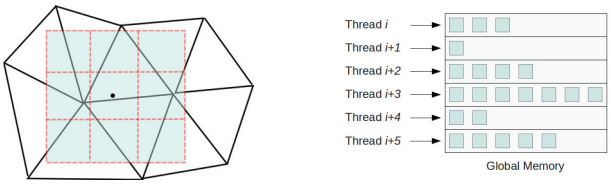


FIG. 5. *Left: Example intersections for a thread when the kernel is centered at an evaluation point (black circle) for postprocessing on a triangular mesh. Right: Global memory access patterns during intersection computations. For an unstructured mesh each thread finds a different set of intersections, resulting in different memory access patterns.*

2.2.3. Comparison of different approaches. The CPU version of the algorithm computes the integration of an intersected region immediately after calculating the intersections for a given triangle with the Sutherland–Hodgman algorithm. The level of parallelism that can be achieved on the GPU is bottlenecked by the Sutherland–Hodgman algorithm due to the fact that many threads are idling while another thread in that stream computes the integration over an intersection. In order to maximize throughput on the GPU we use two passes. In the first pass, we calculate a batch of intersection regions and store them. In the second pass, we compute the integrations over that batch of intersections. This increases computational density, which allows the GPU to decrease thread divergence and increase throughput.

We also investigated a CPU-GPU hybrid approach where the Sutherland–Hodgman intersections were calculated on the CPU and the integrations performed on the GPU. We found that the lengthy data transfer times between the CPU and GPU negated any performance gains from the increased computational power. The process is bandwidth limited, and transfer times between the CPU and GPU dominated the actual computation times. This led us to perform everything on the GPU.

In our GPU method each evaluation point was processed by a single warp (32 threads). Using a single warp per evaluation point allows each logical thread group to work separately and helps to prevent idling while threads are waiting on

TABLE 1

Timing results in seconds for CPU and GPU postprocessing over a variable-sized unstructured triangular mesh (see Mesh Example 1 *in section* 3.3*) using three different numbers of mesh elements.*

—	H			$1.5H$		
Mesh	CPU	GPU	Speedup	CPU	GPU	Speedup
$\mathbb{P}^2$						
1350	45.34	6.80	6.67	78.71	10.07	7.81
5662	302.94	47.26	6.41	496.95	71.19	6.98
22960	4201.53	644.13	6.52	5986.00	897.61	6.67
$\mathbb{P}^3$						
1350	273.86	25.24	10.85	447.48	36.39	12.29
5662	1350.33	184.56	7.31	2558.03	230.96	11.08
22960	10026.02	1424.15	7.04	23507.80	2203.35	10.67
$\mathbb{P}^4$						
1350	822.98	75.61	10.88	1157.59	111.46	10.38
5662	4058.99	447.02	9.07	6617.33	657.13	10.07
22960	24331.33	2886.19	8.43	45166.64	4675.63	9.66

results from other streams. We tried other configurations, including one thread per evaluation point and up to one block per evaluation point. Setting the granularity at the warp level for each evaluation point produced the best results. For the operations that need to be computed with respect to a given evaluation point, there is little overlapping data. Any gains from having shared data between all the threads in a block were mitigated by additional synchronization points in the code. However, gains were seen by reducing idle time. Warp level granularity allows for explicit synchronization calls between threads to be avoided. Synchronization is achieved implicitly due to the lock-step execution of threads in a warp.

The method can scale to suit out-of-core processes through the use of tiling. The mesh can be tiled into manageable sections and the postprocessing done seamlessly for each segmented tile. Due to the nature of the computations, each tile is independent of the rest and there is a minimal amount of global data that is needed by all tiles. The small amount of overhead data that is shared can easily be stored globally for the entirety of the program.

In Table 1 we present a comparison of the postprocessing times (in seconds) required by our CPU and GPU implementations of the SIAC filter for a variable-sized unstructured triangular mesh (see Mesh Example 1 in section 3.3). These timings are for two different kernel scaling values, namely H and $1.5H$, H being the largest side of all the triangular elements. Postprocessing was performed for the DG solutions of an advection equation (details in section 3) with three different polynomial degrees. As discussed previously, memory allocation for the Sutherland–Hodgman algorithm on the GPU is not straightforward, and this issue is even more pronounced for the variable-sized mesh when postprocessing around the areas of obvious spatial transition in element size. However, as seen in Table 1 we are still able to obtain $6\times$–$12\times$ speedup over the CPU implementation when running on the GPU. We note that for the Delaunay mesh in Figure 1 we observed up to an $18\times$ speedup. The CPU used in this comparison was an Intel Xeon X7542 running at 2.67GHz, and the GPU used was a Nvidia Tesla C2050.

3. Numerical results. In this section we provide postprocessing results that demonstrate the efficacy of the SIAC filter when applied to multiple unstructured triangulations. In all the examples we consider the solutions of the constant coefficient

TABLE 2
L_2-errors for various kernel scalings used in the SIAC filter for a simple Delaunay triangulation.

–	Before filtering		$m = 0.5$		$m = 0.75$		$m = 1.0$		$m = 1.5$	
Mesh	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order
	$\mathbb{P}^2$									
878	3.26E-04	—	1.14E-04	—	5.34E-05	—	6.54E-05	—	not valid	—
3588	3.55E-05	3.19	1.40E-05	3.02	9.61E-06	2.47	6.20E-06	3.40	7.48E-06	—
14888	4.27E-06	3.05	1.50E-06	3.22	1.02E-06	3.23	6.30E-07	3.30	2.30E-07	5.02
59264	5.46E-07	2.96	2.07E-07	2.85	1.56E-07	2.70	1.09E-07	2.53	4.51E-08	2.35
	$\mathbb{P}^3$									
878	1.10E-05	—	1.71E-06	—	8.09E-07	—	2.95E-06	—	not valid	—
3588	5.84E-07	4.23	1.05E-07	4.02	3.09E-08	4.71	1.46E-08	7.65	2.43E-07	—
14888	3.78E-08	3.94	8.55E-09	3.61	3.45E-09	3.16	1.20E-09	3.60	9.06E-10	8.06
59264	2.44E-09	3.95	5.21E-10	4.03	2.48E-10	3.79	1.08E-10	3.47	1.87E-11	5.59
	$\mathbb{P}^4$									
878	3.32E-07	—	2.37E-08	—	1.27E-08	—	not valid	—	not valid	—
3588	9.32E-09	5.15	3.77E-09	2.65	3.69E-09	1.78	3.69E-09	—	not valid	—
14888	2.62E-10	4.57	3.57E-11	6.72	7.39E-12	8.96	6.07E-12	9.24	1.01E-11	—
59264	8.69E-12	4.91	2.44E-12	3.87	1.55E-12	2.25	1.03E-12	2.55	6.94E-13	3.86

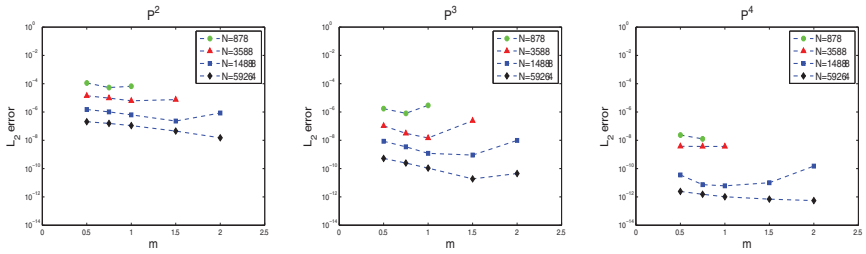


FIG. 6. *L_2-errors versus different kernel scalings when postprocessing over a simple Delaunay triangulation. Left: $\mathbb{P}^2$; middle: $\mathbb{P}^3$; and right: $\mathbb{P}^4$ polynomials. N represents the number of triangular elements in the mesh.*

linear advection equation given below:

$$u_t + u_x + u_y = 0, \qquad (x, y) \in (0, 1) \times (0, 1), \qquad T = 12.5 \tag{11}$$

with initial condition $u(0, x, y) = \sin(2\pi(x+y))$. We further note that to generate the various unstructured meshes we used the Gmsh finite element mesh generator [10].

3.1. Simple Delaunay triangulation. For this example we consider a simple Delaunay triangulation of the domain given in Figure 1. As we discussed in section 2, we consider the largest side of all the triangular elements and denote it with H. We then perform postprocessing using mH as the kernel scaling values, where $m = 0.5, 1.0, 1.5, 2.0$. Table 2 and Figure 6 provide the L_2-error results and plots for these scaling values. The $m = 2.0$ case is purposely omitted from the table, as this scaling is *not valid* for many of the meshes; i.e., the kernel would become larger than the domain size. However, the $m = 2.0$ data is provided in the error plots when available. This omission has been done for all the tables presented herein. Generally speaking, as we increase the kernel width by using a larger m, the error decreases until it reaches a point where we obtain the minimal error value and it increases afterwards. For coarser mesh structures it is often more beneficial to use a smaller kernel to avoid the not valid scenarios. We further note that comparing the L_2-errors before and after filtering, we always obtain accuracy enhancement after the application of the SIAC filter.

In Figure 7 we present the pointwise error contour plots. As can be observed from these plots, the errors are highly oscillatory before the application of the postprocessor

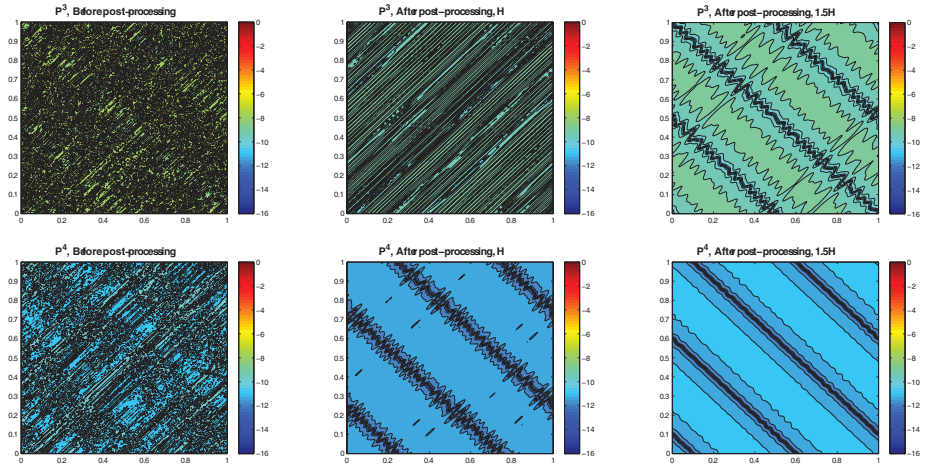
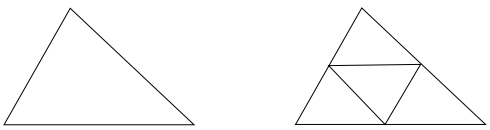


FIG. 7. *Pointwise error contour plots before and after postprocessing over a simple Delaunay triangulation with* $N = 14888$ *elements. Left column: before postprocessing; middle column:* H *scaling; right column:* $1.5H$ *scaling. Top row:* $\mathbb{P}^3$ *polynomials; bottom row:* $\mathbb{P}^4$ *polynomials.*

(left column). However, the postprocessor filters out the oscillations, and this effect is noticeably visible when using $\mathbb{P}^4$ polynomials. The magnitude of the error also decreases after postprocessing. Moreover, we get better results in terms of smoothness with a larger kernel; however, the error might increase in some cases.

3.2. Delaunay triangulation with element splitting. For this case, we took the unstructured mesh in Figure 1 and refined it with splitting (Figure 8). We suspected that this would give better postprocessing results due to the natural hierarchy of solution spaces generated. Table 3 and Figure 9 provide the L_2-error values with respect to different sizes of kernel scalings. Again, it is noted that, generally, there is an optimal kernel scaling value for which we obtain the minimum L_2-error. In addition, Figure 10 presents the pointwise error contour plots. Compared to the contour plots in Figure 7, these provide much smoother error values.



(a) Triangular element (b) Splitting

FIG. 8. *Refining a sample triangular element by splitting.*

3.3. Delaunay triangulation with variable-sized elements. In this example we examine two variants of the Delaunay triangulation of the domain, shown in Figure 11, where there is an obvious spatial transition in element size in the interior of the domain (a refinement ratio of 0.5 has been applied). This change was made in the middle of the mesh in order to maintain the periodic boundary conditions and simplify the application of the postprocessor. Table 4 and Figure 12 present the L_2-errors for postprocessing over Mesh Example 1 in Figure 11 using different kernel

TABLE 3

L_2-errors for various kernel scalings used in the SIAC filter for a triangulation with element splitting.

–	Before filtering		$m = 0.5$		$m = 0.75$		$m = 1.0$		$m = 1.5$	
Mesh	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order
	$\mathbb{P}^2$									
776	3.63E-04	—	1.10E-04	—	7.08E-05	—	1.25E-04	—	not valid	—
3104	4.46E-05	3.02	1.22E-05	3.17	7.84E-06	3.17	6.45E-06	4.27	not valid	—
12416	5.68E-06	2.97	1.46E-06	3.06	8.24E-07	3.25	5.02E-07	3.68	1.98E-06	—
49664	7.16E-07	2.98	1.80E-07	3.15	1.09E-07	2.20	5.97E-08	3.07	8.11E-08	4.60
	$\mathbb{P}^3$									
776	1.31E-05	—	1.47E-06	—	9.88E-07	—	8.52E-06	—	not valid	—
3104	9.03E-07	3.85	1.23E-07	3.57	2.71E-08	5.18	1.30E-07	6.03	not valid	—
12416	5.98E-08	3.92	1.17E-08	3.39	3.28E-09	6.02	1.99E-09	6.02	4.58E-08	—
49664	4.32E-09	3.79	1.05E-09	3.47	2.34E-10	3.80	5.85E-11	5.08	6.20E-10	6.20
	$\mathbb{P}^4$									
776	4.29E-07	—	2.68E-08	—	4.48E-08	—	not valid	—	not valid	—
3104	1.56E-08	4.78	4.84E-10	5.79	2.52E-10	7.47	4.07E-09	—	not valid	—
12416	6.09E-10	4.68	2.76E-11	4.13	6.66E-12	5.24	2.05E-11	7.63	not valid	—
49664	2.72E-11	4.48	1.49E-12	4.21	7.81E-13	3.09	7.12E-13	4.85	5.17E-12	—

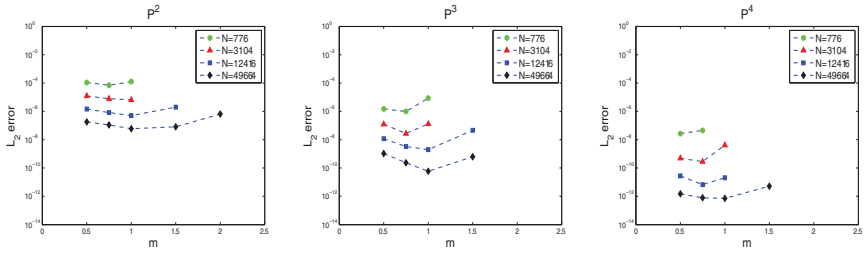


FIG. 9. *L_2-errors versus different kernel scalings when postprocessing over a triangulation with element splitting. Left: $\mathbb{P}^2$; middle: $\mathbb{P}^3$; and right: $\mathbb{P}^4$ polynomials. N represents the number of triangular elements in the mesh.*

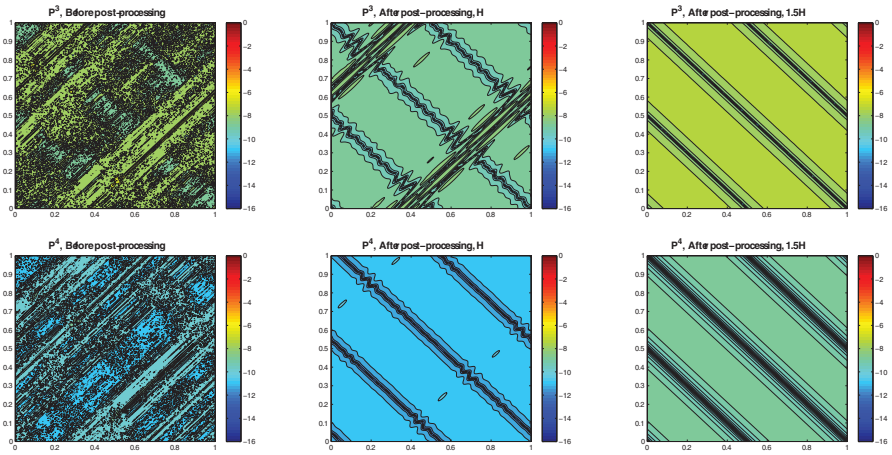
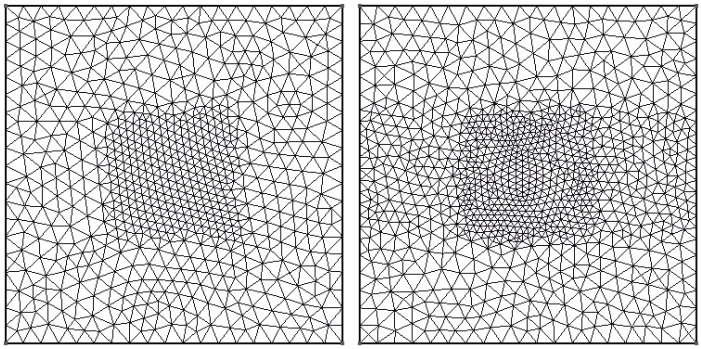


FIG. 10. *Pointwise error contour plots before and after postprocessing over a triangulation with element splitting with $N = 12416$ elements. Left column: before postprocessing; middle column: H scaling; right column: $1.5H$ scaling. Top row: $\mathbb{P}^3$ polynomials; bottom row: $\mathbb{P}^4$ polynomials.*

scaling values. Moreover, Figure 13 provides the pointwise error contour plots for this mesh. We observe postprocessing behavior similar to that of previous mesh examples. Postprocessing results for Mesh Example 2 are provided in Table 5 and Figures 14 and 15.



(a) Mesh Example 1 (b) Mesh Example 2

FIG. 11. *Examples of variable-sized unstructured Delaunay triangulation.*

TABLE 4
L_2-errors for various kernel scalings used in the SIAC filter for Mesh Example 1.

–	Before filtering		$m = 0.5$		$m = 0.75$		$m = 1.0$		$m = 1.5$	
Mesh	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order
	$\mathbb{P}^2$									
1350	2.36E-04	—	6.74E-05	—	4.68E-05	—	5.30E-05	—	not valid	—
5662	3.26E-05	2.85	1.30E-05	2.37	7.59E-06	2.62	3.75E-06	3.82	6.78E-06	—
22960	4.02E-06	3.01	1.66E-06	2.96	1.19E-06	2.67	7.43E-07	2.33	2.91E-07	4.54
90682	4.93E-07	3.02	1.95E-07	3.08	1.44E-07	3.04	9.53E-08	2.96	3.37E-08	3.11
	$\mathbb{P}^3$									
1350	8.50E-06	—	9.99E-07	—	3.90E-07	—	2.06E-06	—	not valid	—
5662	5.34E-07	3.99	9.84E-08	3.34	2.32E-08	4.07	1.14E-08	7.49	2.20E-07	—
22960	2.59E-08	4.36	7.59E-09	3.69	2.70E-09	3.10	9.15E-10	3.63	8.53E-10	8.01
90682	2.20E-09	3.55	5.04E-10	3.91	1.87E-10	3.85	5.93E-11	3.94	1.05E-11	6.34
	$\mathbb{P}^4$									
1350	2.23E-07	—	1.33E-08	—	8.41E-09	—	not valid	—	not valid	—
5662	7.50E-09	4.89	6.22E-10	4.41	1.01E-10	6.37	1.41E-10	—	not valid	—
22960	2.34E-10	5.00	2.56E-11	4.60	7.12E-12	3.82	6.08E-12	4.53	9.60E-12	—
90682	7.73E-12	4.91	2.17E-12	3.56	1.38E-12	2.36	1.04E-12	2.55	6.83E-13	3.81

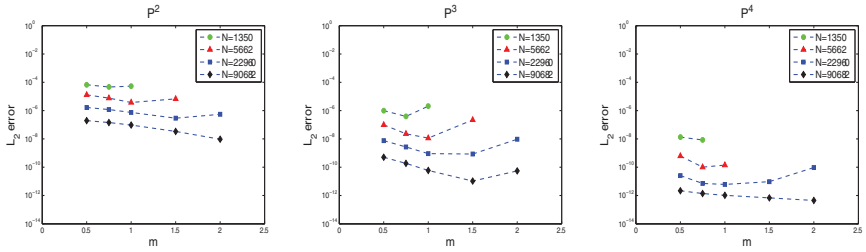


FIG. 12. *L_2-errors versus different kernel scalings when postprocessing over Mesh Example* 1. *Left: $\mathbb{P}^2$; middle: $\mathbb{P}^3$; and right: $\mathbb{P}^4$ polynomials. N represents the number of triangular elements in the mesh.*

3.4. Delaunay triangulation with stretched elements. Here we consider a sample Delaunay mesh with stretched elements in the x direction (Figure 16). This is the so-called *anisotropic* unstructured mesh and is often the type of mesh we see in practice when simulating flows which have strong preferential direction. Table 6 and Figures 17 and 18 provide the postprocessing results. Again, through the application of the SIAC filter we are able to smooth out the oscillations in the error and obtain lower error values.

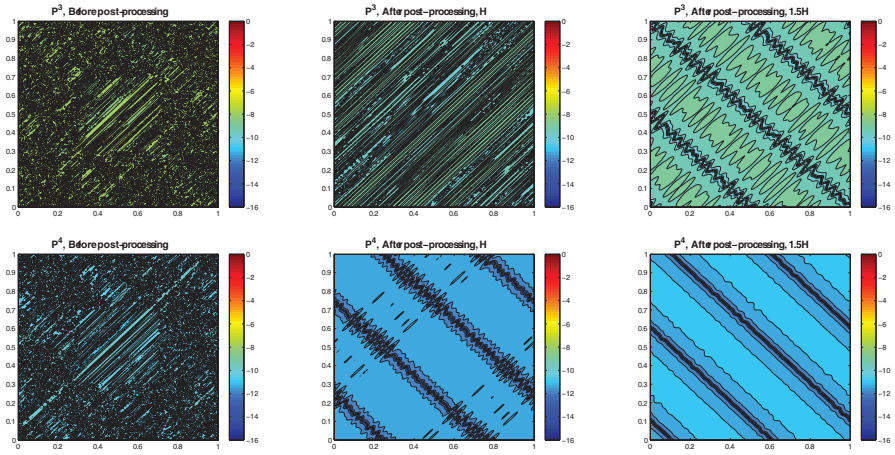


FIG. 13. *Pointwise error contour plots before and after postprocessing over Mesh Example* 1 *with* $N = 22960$ *elements. Left column: before postprocessing; middle column: H scaling; right column:* $1.5H$ *scaling. Top row:* $\mathbb{P}^3$ *polynomials; bottom row:* $\mathbb{P}^4$ *polynomials.*

TABLE 5
L_2-errors for various kernel scalings used in the SIAC filter for Mesh Example 2.

–	Before filtering		$m = 0.5$		$m = 0.75$		$m = 1.0$		$m = 1.5$	
Mesh	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order
					$\mathbb{P}^2$					
1972	1.87E-04	—	6.61E-05	—	4.40E-05	—	5.02E-05	—	not valid	—
8240	2.34E-05	3.00	9.06E-06	2.86	4.69E-06	3.22	2.31E-06	4.44	6.52E-06	—
34562	2.91E-06	3.01	1.18E-06	2.94	7.99E-07	2.55	4.68E-07	2.30	1.68E-07	5.27
138254	3.47E-07	3.06	1.38E-07	3.09	9.60E-08	3.05	6.14E-08	2.93	2.32E-08	2.84
					$\mathbb{P}^3$					
1972	6.29E-06	—	1.04E-06	—	3.97E-07	—	2.27E-06	—	not valid	—
8240	3.74E-07	4.07	7.02E-08	3.88	2.01E-08	4.30	1.10E-08	7.68	2.17E-07	—
34562	2.53E-08	3.89	6.35E-09	3.46	1.70E-09	3.56	4.01E-10	4.77	8.11E-10	8.06
138254	1.51E-09	4.06	4.58E-10	3.79	1.65E-10	3.36	4.27E-11	3.23	7.08E-12	6.83
					$\mathbb{P}^4$					
1972	1.65E-07	—	8.71E-09	—	9.29E-09	—	not valid	—	not valid	—
8240	4.64E-09	5.15	3.75E-10	4.53	2.34E-10	5.31	2.69E-10	—	not valid	—
34562	1.67E-10	4.79	2.06E-11	4.18	1.46E-11	4.00	1.45E-11	4.21	1.62E-11	—
138254	5.57E-12	4.90	2.15E-12	3.26	1.27E-12	3.52	9.04E-13	4.00	6.28E-13	4.68

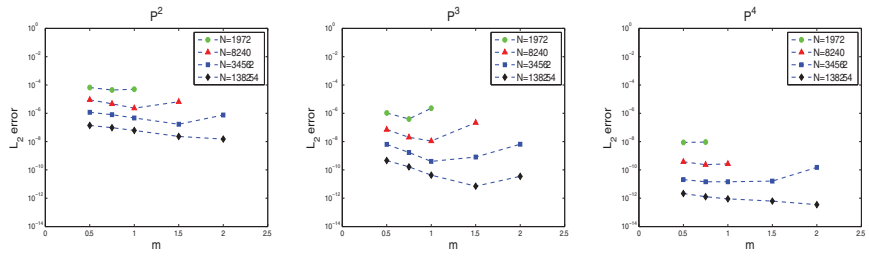


FIG. 14. *L_2-errors versus different kernel scalings when postprocessing over Mesh Example* 2. *Left:* $\mathbb{P}^2$*; middle:* $\mathbb{P}^3$*; and right:* $\mathbb{P}^4$ *polynomials. N represents the number of triangular elements in the mesh.*

4. Conclusion. The SIAC filtering technique has traditionally been applied to translation invariant meshes. In some cases, random meshes in one dimension and structured smoothly varying meshes in two dimensions were also considered. However, as the assumption of any sort of regularity will restrict the application of this filtering technique to more complex simulations, we provided in this paper the behavior

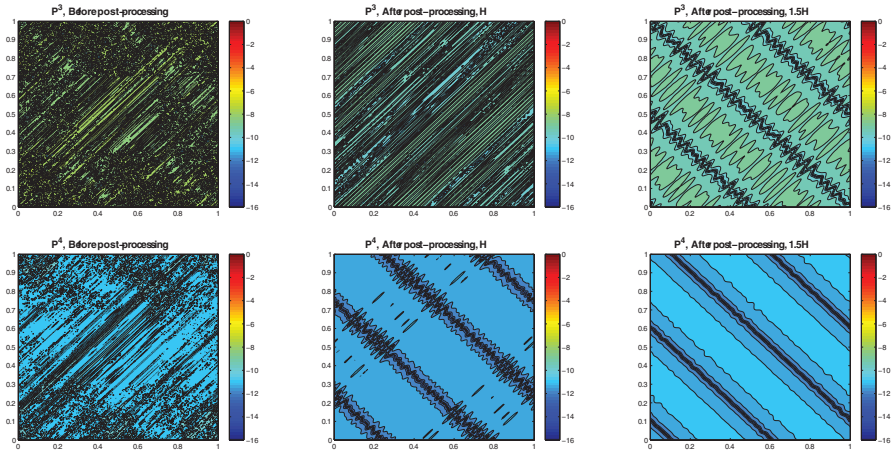


FIG. 15. *Pointwise error contour plots before and after postprocessing over Mesh Example* 2 *with* $N = 34562$ *elements. Left column: before postprocessing; middle column:* H *scaling; right column:* $1.5H$ *scaling. Top row:* $\mathbb{P}^3$ *polynomials; bottom row:* $\mathbb{P}^4$ *polynomials.*

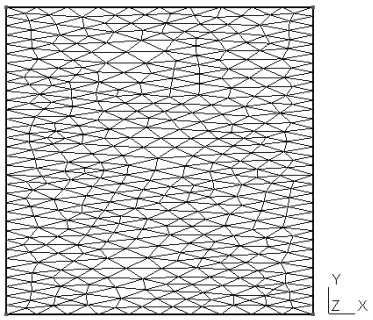


FIG. 16. *A sample unstructured triangular mesh with stretched elements in the* x *direction.*

TABLE 6
L_2-errors for various kernel scalings used in the SIAC filter for a stretched triangulation.

—	Before filtering		$m = 0.5$		$m = 0.75$		$m = 1.0$		$m = 1.5$	
Mesh	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order	L_2-error	Order
	$\mathbb{P}^2$									
796	8.36E-04	—	3.38E-04	—	2.96E-04	—	not valid	—	not valid	—
3232	1.07E-04	2.96	3.48E-05	3.27	1.86E-05	3.99	3.31E-05	—	not valid	—
12816	1.24E-05	3.10	4.10E-06	3.08	2.51E-06	2.88	1.62E-06	4.35	5.54E-06	—
51430	1.62E-06	2.93	6.79E-07	2.59	4.12E-07	2.60	2.13E-07	2.92	1.62E-07	5.09
	$\mathbb{P}^3$									
796	5.47E-05	—	5.25E-06	—	not valid	—	not valid	—	not valid	—
3232	3.47E-06	3.98	5.13E-07	3.35	2.04E-07	—	1.72E-06	—	not valid	—
12816	2.00E-07	4.11	3.10E-08	4.04	6.46E-09	4.98	7.44E-09	7.85	1.79E-07	—
51430	1.32E-08	3.92	2.23E-09	3.79	5.05e-10	3.67	1.80E-10	5.36	1.45E-09	6.94
	$\mathbb{P}^4$									
796	2.10E-06	—	1.07E-07	—	not valid	—	not valid	—	not valid	—
3232	7.38E-08	4.83	4.40E-09	4.60	7.07E-09	—	not valid	—	not valid	—
12816	2.14E-09	5.10	7.38E-11	5.89	9.61E-12	9.52	1.08E-10	—	not valid	—
51430	6.72E-11	4.99	2.60E-12	4.82	8.17E-13	3.55	7.00E-13	7.26	1.49E-11	—

and complexity of the computational extension of this filtering technique to totally unstructured tessellations. We note that this is an important step towards a more generalized SIAC filter. We considered various unstructured tessellations and demonstrated that it is indeed possible to get reduced errors and increased smoothness

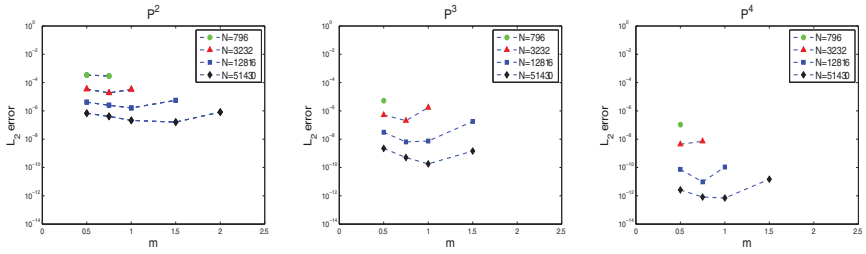


FIG. 17. *L_2-errors versus different kernel scalings when postprocessing over a stretched triangulation. Left: $\mathbb{P}^2$; middle: $\mathbb{P}^3$; and right: $\mathbb{P}^4$ polynomials. N represents the number of triangular elements in the mesh.*

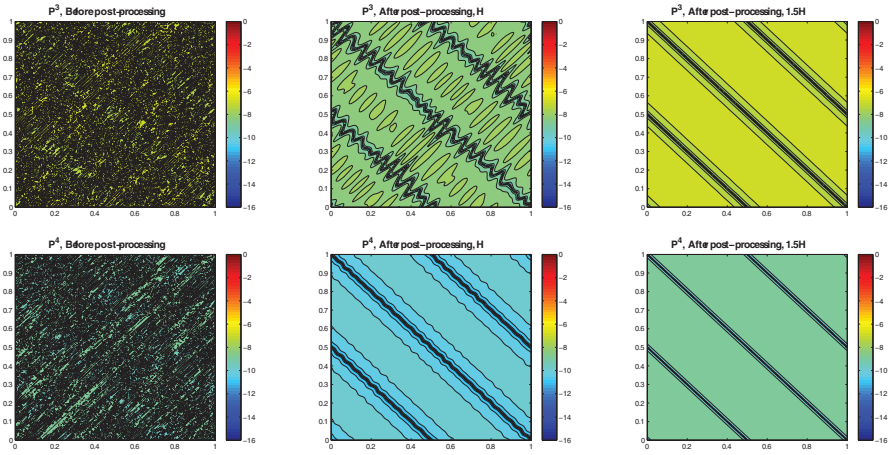


FIG. 18. *Pointwise error contour plots before and after postprocessing over a stretched triangulation with $N = 12816$ elements. Left column: before postprocessing; middle column: H scaling; right column: $1.5H$ scaling. Top row: $\mathbb{P}^3$ polynomials; bottom row: $\mathbb{P}^4$ polynomials.*

through a proper choice of kernel scaling. Furthermore, we provided numerical results which confirm the smoothness-increasing capability of the SIAC filter when applied to unstructured tessellations. Finally, GPU implementations of this SIAC filter were described. Using a single GPU, up to an 18× reduction in computational costs over the traditional CPU implementations can be obtained for these unstructured tessellations. We documented why care must be taken with the programming of the GPUs to obtain such a reduction when applied to SIAC postprocessing of DG solutions.

REFERENCES

[1] M. AINSWORTH AND J. ODEN, *A Posteriori Error Estimation in Finite Element Analysis*, Wiley-Interscience, New York, 2000.

[2] I. BABUŠKA AND R. RODRIGUEZ, *The problem of the selection of an a posteriori error indicator based on smoothing techniques*, Internat. J. Numer. Methods Engrg., 36 (1993), pp. 539–567.

[3] J. H. BRAMBLE AND A. H. SCHATZ, *Higher order local accuracy by averaging in the finite element method*, Math. Comp., 31 (1977), pp. 94–111.

[4] B. COCKBURN, *Discontinuous Galerkin methods for convection-dominated problems*, in High-Order Methods for Computational Physics, Lect. Notes Comput. Sci. Eng. 9, Springer, Berlin, 1999, pp. 69–224.

[5] B. Cockburn, *Devising discontinuous Galerkin methods for non-linear hyperbolic conservation laws*, J. Comput. Appl. Math., 128 (2001), pp. 187–204.
[6] B. Cockburn, G. Karniadakis, and C.-W. Shu, *The development of the discontinuous Galerkin methods*, in Discontinuous Galerkin Methods: Theory, Computation and Application, Lect. Notes Comput. Sci. Eng. 11, Springer-Verlag, Berlin, 2000, pp. 3–50.
[7] B. Cockburn, M. Luskin, C.-W. Shu, and E. Süli, *Post-processing of Galerkin methods for hyperbolic problems*, in Proceedings of the International Symposium on Discontinuous Galerkin Methods (Newport, RI, 1999), Springer, Berlin, 2000, pp. 291–300.
[8] B. Cockburn, M. Luskin, C.-W. Shu, and E. Süli, *Enhanced accuracy by post-processing for finite element methods for hyperbolic equations*, Math. Comp., 72 (2003), pp. 577–606.
[9] S. Curtis, R. M. Kirby, J. K. Ryan, and C.-W. Shu, *Postprocessing for the discontinuous Galerkin method over nonuniform meshes*, SIAM J. Sci. Comput., 30 (2007), pp. 272–289.
[10] C. Geuzaine and J.-F. Remacle, *Gmsh: A three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*, Internat. J. Numer. Methods Engrg., 79 (2009), pp. 1309–1331.
[11] G. E. Karniadakis and S. J. Sherwin, *Spectral/hp Element Methods for Computational Fluid Dynamics*, 2nd ed., Oxford University Press, New York, 2005.
[12] J. King, H. Mirzaee, J. K. Ryan, and R. M. Kirby, *Smoothness-increasing accuracy-conserving (SIAC) filtering for discontinuous Galerkin solutions: Improved errors versus higher-order accuracy*, J. Sci. Comput., 53 (2012), pp. 129–149.
[13] X. Li, J. K. Ryan, R. M. Kirby, and K. Vuik, *Computationally Efficient Position-Dependent Smoothness-Increasing Accuracy-Conserving (SIAC) Filtering: The Uniform Mesh Case*, manuscript, 2012.
[14] H. Mirzaee, L. Ji, J. K. Ryan, and R. M. Kirby, *Smoothness-increasing accuracy-conserving (SIAC) postprocessing for discontinuous Galerkin solutions over structured triangular meshes*, SIAM J. Numer. Anal., 49 (2011), pp. 1899–1920.
[15] H. Mirzaee, J. K. Ryan, and R. M. Kirby, *Efficient implementation of smoothness-increasing accuracy-conserving (SIAC) filters for discontinuous Galerkin solutions*, J. Sci. Comput., 52 (2011), pp. 85–112.
[16] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell, *A survey of general-purpose computation on graphics hardware*, Comput. Graphics Forum, 26 (2007), pp. 80–113.
[17] W. Reed and T. Hill, *Triangular Mesh Methods for the Neutron Transport Equation*, Tech. report, Los Alamos Scientific Laboratory, Los Alamos, NM, 1973.
[18] G. Ritcher, *An optimal-order error estimate for the discontinuous Galerkin method*, Math. Comp., 50 (1988), pp. 75–88.
[19] J. K. Ryan and C.-W. Shu, *On a one-sided post-processing technique for the discontinuous Galerkin methods*, Methods Appl. Anal., 10 (2003), pp. 295–307.
[20] J. Ryan, C.-W. Shu, and H. Atkins, *Extension of a postprocessing technique for the discontinuous Galerkin method for hyperbolic equations with application to an aeroacoustic problem*, SIAM J. Sci. Comput., 26 (2005), pp. 821–843.
[21] M. Steffen, S. Curtis, R. M. Kirby, and J. K. Ryan, *Investigation of smoothness enhancing accuracy-conserving filters for improving streamline integration through discontinuous fields*, IEEE Trans. Vis. Comput. Graph., 14 (2008), pp. 680–692.
[22] I. E. Sutherland and G. W. Hodgman, *Reentrant polygon clipping*, Comm. ACM, 17 (1974), pp. 32–42.
[23] P. van Slingerland, J. K. Ryan, and C. Vuik, *Position-dependent smoothness-increasing accuracy-conserving (SIAC) filtering for improving discontinuous Galerkin solutions*, SIAM J. Sci. Comput., 33 (2011), pp. 802–825.
[24] D. Walfisch, J. K. Ryan, R. M. Kirby, and R. Haimes, *One-sided smoothness-increasing accuracy-conserving filtering for enhanced streamline integration through discontinuous fields*, J. Sci. Comput., 38 (2009), pp. 164–184.