

A Scalable, Efficient Scheme for Evaluation of Stencil Computations over Unstructured Meshes

James King
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT
jsking@sci.utah.edu

Robert M. Kirby
School of Computing and
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT
kirby@cs.utah.edu

ABSTRACT

Stencil computations are a common class of operations that appear in many computational scientific and engineering applications. Stencil computations often benefit from compile-time analysis, exploiting data-locality, and parallelism. Post-processing of discontinuous Galerkin (dG) simulation solutions with B-spline kernels is an example of a numerical method which requires evaluating computationally intensive stencil operations over a mesh. Previous work on stencil computations has focused on structured meshes, while giving little attention to unstructured meshes. Performing stencil operations over an unstructured mesh requires sampling of heterogeneous elements which often leads to inefficient memory access patterns and limits data locality/reuse. In this paper, we present an efficient method for performing stencil computations over unstructured meshes which increases data-locality and cache efficiency, and a scalable approach for stencil tiling and concurrent execution. We provide experimental results in the context of post-processing of dG solutions that demonstrate the effectiveness of our approach.

1. INTRODUCTION

Stencil computations are common operations performed by many numerical methods in the context of scientific and engineering applications. A *stencil* is a geometric pattern which performs computations on a multi-dimensional grid using information from a localized region. Traditionally, computational methods have employed structured or block-structured techniques to compute solutions over regularly spaced grids of points in 2D and 3D. However, the need for accurately discretizing complex geometries has led to a rise in the use of unstructured grid techniques [15]. The terms stencil, grid, and mesh are often used in the HPC literature. In the context of this paper we are using specific definitions for these terms. A mesh is a convex polygon surface formed from collection of vertices, edges, and faces. A grid is a set of points defined over a mesh, which are not necessarily col-

located with the vertices of the mesh. A stencil is a fixed geometric pattern which defines a localized sampling region centered around some point. This definition does require that stencils have a fixed memory access pattern as is the case in many other works.

Numerical methods such as the Finite Volume Method (FVM) and the Finite Element Method (FEM) are defined to operate over a mesh. Solutions are often evaluated at a set of points extracted from the mesh which form a multi-dimensional grid. Meshes are classified as either structured or unstructured, with each having their own set of advantages and disadvantages. Structured meshes are easier to sample, allow for easy parallelization, and generally require no spatial data structure to manage. However, there is great difficulty in meshing some complex geometries with fully structured meshes. Unstructured meshes offer the advantage of easier discretization of complex geometries, but they are harder to sample, often requiring additional overhead in the form of some spatial data structure, and are generally more difficult to parallelize.

In context of dG post-processing, a grid of points is defined over the mesh which correspond to the numerical quadrature points for each polygon element in the mesh. The geometry of this grid depends upon the mesh's geometric structure. Structured meshes will lead to regular grid patterns, while unstructured grids will lead to irregular grid patterns. The regular access pattern used by structured grids generally leads to contiguous memory accesses, good memory layout patterns, and high cache efficiency. Efficient computation of stencil operations over structured meshes has been widely studied, and great gains have been made by exploiting parallelism and data locality. Stencil computations performed over unstructured grids is generally much harder than those performed over structure grids, and they often exhibit non-contiguous memory access patterns and lower cache efficiency.

One of the biggest challenges in computing stencil operations over unstructured meshes is efficiently sampling the underlying mesh in the mesh/stencil intersection. DG post-processing requires performing stencil operations that sample information from the neighborhood of mesh elements within the intersection of the stencil and the mesh. In cases such as this, the geometric nature of the mesh has a significant impact on cache efficiency and data locality, especially for many-core architectures.

Previous work on stencil computations has generally focused on optimizing for the case of structured meshes. Initial

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC '13, November 17 - 21 2013, USA

Copyright 2013 ACM 978-1-4503-2378-9/13/11 ...\$15.00.

<http://dx.doi.org/10.1145/2503210.2503214>

work has been conducted in the area of general methods for parallel applications on unstructured meshes/grids [24, 25], but there has been little research devoted to determining efficient methods for evaluating stencil computations over unstructured meshes. This paper contributes the following:

- An efficient method for evaluating stencil computations on unstructured meshes;
- A scalable approach for tiling and concurrent execution of stencil computations over unstructured meshes; and
- An experimental evaluation of our technique on GPU architecture over a variety of unstructured meshes.

The rest of the paper is organized as follows. Section 2 describes some of the related work dealing with stencil computations. In addition, it provides background on the post-processing of discontinuous Galerkin solutions, the motivation for this work, and describes the difficulties in efficiently performing the computations. Section 3 illustrates two major strategies for performing stencil computations over unstructured grids. Section 4 provides implementation details for the approach we used. Section 5 provides experimental results for GPU implementations of each approach. Finally, Section 6 concludes the paper and discusses future work and applications.

2. BACKGROUND

The interest in evaluating numerical methods over unstructured meshes has risen in recent years, in part due to the demand for ever more realistic meshes which conform to highly complex geometries.

2.1 Related Work

Stencil computations have been extensively studied as they have a wide range of applications in science and engineering. The research done in optimizing stencil computations falls into one of two categories. The first is compiler-based optimizations such as auto-tuning and domain-specific languages and compilers. The second is stencil-specific optimizations, which often provide greater performance increases than compiler-based optimizations due to their ability to take advantage of specific knowledge related to the computation. The complex nature of modern architectures often requires meticulous tuning to achieve high performance. There has been a large push towards auto-tuning architecture specific code. A framework for generating and auto-tuning architecture specific parallel stencil code was developed in [6]. Stencil-specific optimizations have included tiling mechanisms which take into account the memory access pattern of the stencil to improve load balance and concurrency. A method for generating tiling hyperplanes which allows for concurrent start without the need for pipe-lining was described in [2]. This technique provides perfect load balancing and increases parallelism.

Many-core architectures are very effective at exploiting parallelism in regular computations. Numerical methods performed over unstructured meshes are often classified as irregular. Work has been done to develop ways of classifying the amount of irregularity in a program [4]. Control-flow irregularity and memory-access irregularity are the two major types of irregularity that exist in programs. There has also been work done on run-time techniques, such as inspector/executor methods, for evaluating parallel irregular

computations and reordering/reassigning the operations to remove data dependence conflicts and increase load balance [1]. The amount of static optimization that can be achieved in irregular computations is often limited due to data dependencies that can only be determined at run-time.

The use of streaming many-core architectures to compute stencil operations has significantly increased in recent years. This is due to a number of factors, such as the inherent outer parallelism of stencil operations, the ability of many-core architectures to exploit fine-grained parallelism, and the high FLOP count often required by large scale calculations. Graphics Processing Units (GPUs) are massively-threaded, many-core streaming architectures that use the single program multiple data (SPMD) paradigm to increase memory bandwidth efficiency and computational throughput. Modern GPUs achieve peak single precision floating-point throughput of over 1 TFLOP/s. GPUs contain hundreds of cores arranged into compute grids, known as a streaming multi-processors (SM), which operate in a single instruction multiple data (SIMD) fashion. Logical threads are grouped together into blocks which are then assigned to a physical SM on the GPU. A single GPU can concurrently manage thousands of logical threads at one time and dynamically handle scheduling of their execution and context switching though hardware with little to no overhead [11].

Each SM has a limited amount of register space and shared memory/cache for threads operating within a block. Threads within a block can pass information through this shared memory space, but threads between blocks must pass information through global memory. Global memory is device memory which is shared between all SMs and can be accessed by any thread. Synchronization can be achieved between threads within a logical block, but, in general, there is no way to synchronize threads across blocks. The low-level architectural model of the GPU presents a challenge in writing efficient programs. The Compute Unified Device Architecture (CUDA) programming model [9, 19] and the Open Computing Language (OpenCL) [12] have made strides towards lowering the barrier of programming GPUs.

Significant work has been devoted to the goal of achieving high performance from stencil computations on streaming SIMD architectures [10, 14, 20]. The SIMD architecture of the GPU fits well with stencil computations due to the inherent data level parallelism. Other researchers' work has shown promise for high performance of stencil computations on GPUs, using techniques such as auto-tuning and auto-generation of code [29]. Techniques such as data layout transformation and dynamic tiling at the thread level were demonstrated in [5].

Previous work on stencil computations has relied on exploiting regular geometric memory access patterns on structured grids to achieve high performance and maximize parallelism. This can not be relied upon in the case of unstructured meshes/grids, where a given stencil will often have different sampling patterns based on the point it is centered around. Discontinuous Galerkin post-processing is an example of a numerical method which requires performing stencil computations over unstructured meshes. We chose dG post-processing as a motivating example and demonstrator for our technique, although our technique is not specifically tailored to or limited to dG post-processing.

2.2 Post-Processing of Discontinuous Galerkin Solutions

The discontinuous Galerkin (dG) method has 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 classic continuous Galerkin FEM which seeks approximations that are piecewise continuous, 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, discontinuity between element interfaces can be problematic during post-processing, where there is often an implicit assumption that the field upon which the post-processing methodology is acting is smooth. A class of post-processing techniques was introduced 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 [26, 28] as a means of introducing continuity at element interfaces while maintaining the order of accuracy of the original input dG solution.

The post-processor itself is simply the discontinuous Galerkin solution u convolved against a linear combination of B-splines. That is, in one-dimension,

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

where u^* is the post-processed solution, h is the characteristic element length (elements are line segments in 1D) and

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

is the convolution kernel, which we refer to as the convolution stencil. $\psi^{(k+1)}$ is the B-spline of order $k+1$ and $c_{\gamma}^{r+1,k+1}$ represent the stencil coefficients. The term r is the upper bound on the polynomial degree that the B-splines are capable of reproducing through convolution. The stencil width increases proportionately with r . x_{γ} represent the positions of the stencil nodes and are defined by $x_{\gamma} = -\frac{r}{2} + \gamma$, $\gamma = 0, \dots, r$, where $r = 2k$. This will form a line and a square lattice of regularly spaced stencil nodes in 1D and 2D respectively.

The post-processor takes as input an array of the polynomial modes used in the discontinuous Galerkin method and produces the values of the post-processed solution at a set of specified grid points. We choose these grid points to correspond with specific quadrature points which can be used at the end of the simulation for such things as error calculations. Post-processing of the entire domain is obtained by repeating the same procedure for all the grid points. In two dimensions, the convolution stencil is the tensor product of one-dimensional kernels. Therefore, the post-processed solution at $(x, y) \in T_i$, becomes

$$u^*(x, y) = \frac{1}{h^2} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} K \left(\frac{x_1 - x}{h} \right) K \left(\frac{x_2 - y}{h} \right) u(x_1, x_2) dx_1 dx_2 \quad (1)$$

where T_i is a triangular element, u is our approximate dG solution, and we have denoted the two-dimensional coordinate system as (x_1, x_2) .

To calculate the integral involved in the post-processed solution in Equation (1) exactly, we need to decompose the triangular elements that are covered by the stencil support into sub-elements that respect the stencil nodes. The resulting integral is calculated as the summation of the integrals over each sub-element. Figure 1 depicts a possible decomposition of a triangular element based on the stencil-mesh intersection.

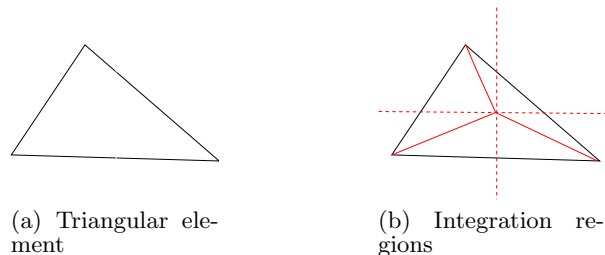


Figure 1: Demonstration of integration regions resulting from the stencil/mesh intersection. Dashed lines represent the breaks between stencil nodes. Solid red lines represent a triangulation of the integration regions.

As demonstrated in Figure 1(b), we divide the triangular region into sub-regions over which there is no break in regularity. Furthermore, we choose to triangulate these sub-regions for ease of implementation. The infinite integrals in Equation (1) may be transformed to finite local sums over elements, using the compact support property of the stencil ($T_j \in Supp\{K\}$). The extent of the stencil or $Supp\{K\}$ is given by $(3k+1)h$ in each direction, where k is the degree of the polynomial approximation. Each of the integrals over a triangle T_j then becomes

$$\begin{aligned} & \int \int_{T_j} K \left(\frac{x_1 - x}{h} \right) K \left(\frac{x_2 - y}{h} \right) u(x_1, x_2) dx_1 dx_2 \\ &= \sum_{n=0}^N \int \int_{\tau_n} K \left(\frac{x_1 - x}{h} \right) K \left(\frac{x_2 - y}{h} \right) u(x_1, x_2) dx_1 dx_2 \quad (2) \end{aligned}$$

where N is the total number of triangular sub-regions formed in the triangular element T_j as the result of stencil/mesh intersection, and τ_n is the n^{th} triangular subregion of the intersection. In the case that the stencil intersects a boundary of the domain, the stencil either wraps around the domain for periodic solutions, or an asymmetric (one-sided) stencil is used [21]. For further details on the discontinuous Galerkin method and post-processing, see [16, 17, 18, 8, 22].

3. ALGORITHM

In previous work, stencil computations are often defined as a method that updates each point in a structured grid according to an expression which depends upon the values of neighboring points in a fixed geometric pattern. For the case of discontinuous Galerkin (dG) post-processing, we use a more general definition of stencil computations, which is the localized sampling area centered around a grid point which intersects the mesh geometry. We now define the *key concepts* used in the context of stencil computations over

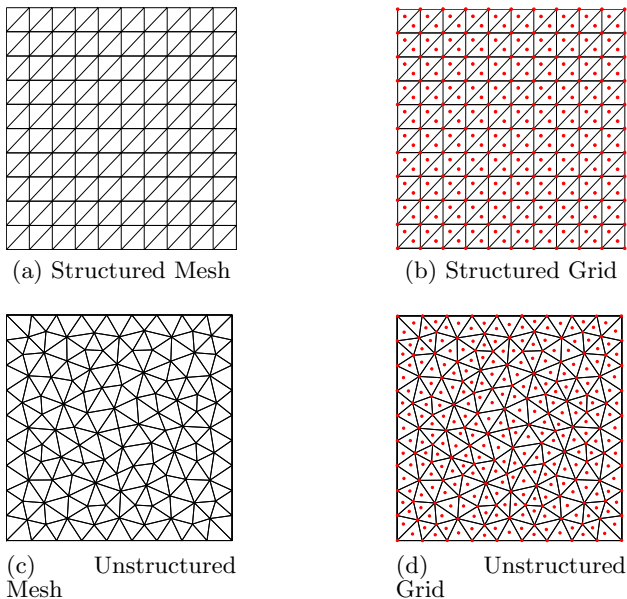


Figure 2: Structured and unstructured meshes and their respective structured and unstructured grids

unstructured meshes: computation grids, stencil operations, spatial data structures, and buffered vs. in-place stencils. All of our tests were conducted over 2D unstructured triangular meshes, therefore we use the terms element and triangle interchangeably.

When evaluating stencil operations over a mesh, a set of evaluation points must be derived in relation to the underlying geometry. This set of points over which the stencil computations are evaluated is denoted as the computation grid. In the case of post-processing of dG solutions, the evaluation points are the quadrature points of the polynomial interpolant defined over each element. Figure 2 illustrates an example of structured and unstructured 2D triangular meshes along with the set of grid points derived from them. In the case of structured meshes, the layout of the quadrature points will follow a regular pattern. For unstructured meshes, the layout of the quadrature points will depend on the size, shape, and orientation of the elements. Post-processing of dG solutions requires sampling the discontinuous piecewise functions that exist over the elements of the mesh.

We define stencil operations to be computations performed which update the value of a grid point at which the stencil is centered, using information within the localized sampling region. The computations depend upon function values of sampled points that lie within the stencil. The stencil may differ for each grid point when computing stencil operations over unstructured grids. This is due to the fact that the set of sample points within the stencil depend upon the intersection between the stencil and the underlying geometry. The varying intersection spaces between grid points will lead to a non-regular sampling pattern that must be calculated independently for each grid point.

As stencil operations rely on local neighborhood relationships between evaluation points, it is a common operation to query all elements within some distance of a given point. Therefore, an efficient method for accessing elements or points

within some spatial region is required. There exist a number of data structures used for spatially decomposing an unstructured grid or mesh in an efficient manner, such as k-d trees, uniform hash grids, quad/oct trees, and bounding volume hierarchies [23]. Given that the stencils, in this case, are square and grid points are roughly uniformly distributed, a uniform hash grid was the most applicable choice [3].

We differentiate between stencil types based on how they operate over their solution memory space. In-place stencils sample from the same memory locations to which the solutions are written. This is often the case with time dependent iterative stencil computations. In-place stencils must be tiled in some fashion as to avoid race conditions. Buffered stencils write the solution to a separate memory space from the space which is sampled to compute the stencil. As such, buffered stencil operations can be processed independently of each other without concern for race conditions. Post-processing of dG solutions is a buffered stencil operation.

3.1 Stencil Evaluation

The most straightforward method for post-processing is a per-point evaluation method which iterates over the grid of points, and for each point finds all of the elements that intersect with the stencil centered around that point. Those intersected regions are then integrated and the values summed to produce the value of post-processed solution at that grid point. We propose an alternate method which is a per-element evaluation method that iterates over each element, and for every element finds all of the points whose stencil intersects with that element. Each of those individual intersections are then integrated, which produces a number of partial solutions that are scattered to multiple grid points. Figure 3 illustrates these two methods. In per-point evaluation, integrations are all partial sums of the same grid point. In per-element evaluation, every grid point whose stencil intersects with the given element will have its value updated with a partial solution.

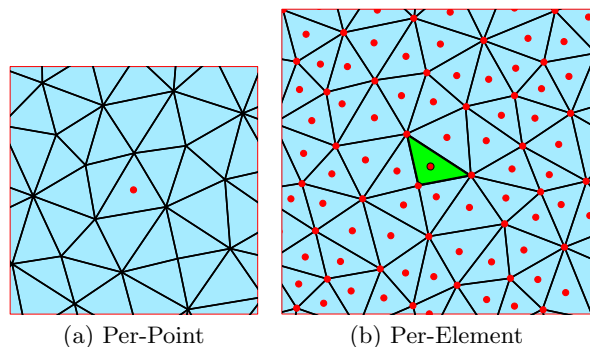


Figure 3: Per-point vs per-element evaluation. Red points indicate grid points that will be updated by this evaluation. The bounds indicate the area covered by the stencil. In the per-point case, the red dot indicates the point whose solution is being evaluated. In the per-element case, the partial solutions are evaluated with respect to the green highlighted element.

Post-processing of dG solutions over unstructured meshes requires finding the intersections between the B-spline stencil and the mesh geometry. We use the Sutherland-Hodgman algorithm [27] to find and triangulate these intersections.

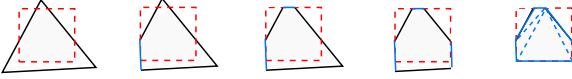


Figure 4: A sample triangulation of an intersection region by the Sutherland-Hodgman algorithm.

Algorithm 1: SutherlandHodgman (SH) Algorithm

```

input : clipPolygon, subjectPolygon
output: intersectionPolygon
List outputList = subjectPolygon;
for Edge clipEdge in clipPolygon do
  List inputList = outputList;
  outputList.clear();
  Point S = inputList.last;
  for Point E in inputList do
    if E inside clipEdge then
      if S not inside clipEdge then
        outputList.add(Intersection(S,E,clipEdge));
      end
      outputList.add(E);
    end
    else if S inside clipEdge then
      outputList.add(Intersection(S,E,clipEdge));
    end
    S ← E;
  end
end
intersectionPolygon ← outputList;

```

This clipping algorithm finds the polygon that is the intersection between two given arbitrary convex polygons and divides the intersection into triangular subregions. Figure 4 illustrates this triangulation process. The convolution stencil used in the post-processing algorithm is broken down into an array of squares as depicted with red dashed lines. Consequently, the problem of finding the integration regions becomes the problem of finding the intersection areas between each square of the stencil array and the triangular elements covered by the stencil support. Figure 5 depicts a sample stencil/mesh overlap.

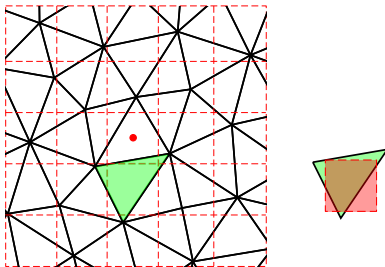


Figure 5: A sample stencil/mesh overlap. Dashed lines represent the two-dimensional stencil as an array of squares. The intersections of the dashed lines are stencil node locations. The subfigure on the right illustrates the intersection of the green highlighted element and the overlapping stencil square.

DG post-processing consists of two main steps. The first is finding and triangulating the stencil/mesh intersections. This will create a set of triangulated subregions. The second is integrating those subregions according to Equation (2) and summing the results. The resulting sum is the post-processed value of the solution u^* at that point.

3.2 Grid Construction

A spatial data structure is needed to efficiently search the elements of an unstructured mesh in order to determine in which element a given point lies. We perform a uniform subdivision of the mesh and each element/point is stored in a hash grid cell based on its spatial coordinates. For per-point sampling the hash grid stores the centroid location of each element. On unstructured meshes, a triangle may be located in a cell while parts of its area extend into neighboring cells. To ensure enclosure (*i.e.* no triangle spans more than two cells in any one dimension), a minimum size on the cells of the hash grid is imposed. The minimum size used in our computation to guarantee enclosure is the length of the longest edge amongst all triangles in the mesh. In the per-element case the hash grid stores the grid points instead of the triangular elements of the underlying mesh. The decomposition in this case has no minimum size restriction on the cells of the grid.

When evaluating the intersection of a stencil and the triangular mesh we first evaluate the intersection of the stencil and the uniform hash grid. The intersected cells store indices of the elements/points that must be tested for intersections with the given element/point being evaluated. We choose the domain of the hash grid to be $[0, 1]$ in both dimensions, with per-point and per-element cell spacings c_p and c_e respectively. We set c_p and c_e to be some factor of s , which represents the longest side amongst all triangles in the mesh. For our tests we used $c_p = s$ and $c_e = \frac{s}{2}$. Construction of the uniform hash grid follows from dividing the mesh into $\lceil \frac{1}{c_p} \rceil$ and $\lceil \frac{1}{c_e} \rceil$ cells in each dimension.

Given an element with vertices (A, B, C) and a grid point (x, y) , we construct a bounding box around the element with corners being defined as

$$\begin{aligned} \min_x &= \min(A_x, B_x, C_x) & \min_y &= \min(A_y, B_y, C_y) \\ \max_x &= \max(A_x, B_x, C_x) & \max_y &= \max(A_y, B_y, C_y). \end{aligned}$$

The bounds are extended by half of the stencil width, which is defined to be $w = s(3P + 1)$, where P is the polynomial order. The bounds of the per-element and per-point stencils (e, p) are defined as

$$\begin{aligned} \text{left}_e &= \lfloor \frac{\min_x - \frac{w}{2}}{c_e} \rfloor & \text{left}_p &= \lfloor \frac{x - \frac{w}{2}}{c_p} \rfloor - 1 \\ \text{right}_e &= \lfloor \frac{\max_x + \frac{w}{2}}{c_e} \rfloor & \text{right}_p &= \lfloor \frac{x + \frac{w}{2}}{c_p} \rfloor + 1 \\ \text{top}_e &= \lfloor \frac{\max_y + \frac{w}{2}}{c_e} \rfloor & \text{top}_p &= \lfloor \frac{y + \frac{w}{2}}{c_p} \rfloor + 1. \\ \text{bottom}_e &= \lfloor \frac{\min_y - \frac{w}{2}}{c_e} \rfloor & \text{bottom}_p &= \lfloor \frac{y - \frac{w}{2}}{c_p} \rfloor - 1 \end{aligned} \quad (3)$$

The hash grid is constructed in a similar manner for both methods, with the per-point hash grid storing the triangle elements and the per-element hash grid storing the grid points.

The size of the intersection search space, in each dimension, for the per-point method is the sum of the stencil width

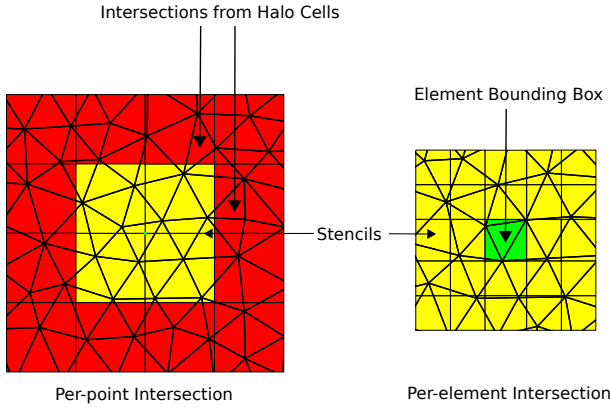


Figure 6: Per-point vs per-element mesh intersections on hash grid. The yellow areas denote the stencil regions, the red area denotes the halo region, and the green area is the element bounding box.

and the width of the cells surrounding the stencil, known as the *halo* region [11]. The size of the intersection space for the per-element scheme is sum of the width of element bounding box and the stencil width. The resulting size of the intersection search space has an upper bound of $2s + w$ for the per-point scheme, and $s + w$ for the per-element scheme. Figure 6 illustrates the difference in the intersection search spaces between the two methods. Elements that lie within the *halo* cells around the stencil but do not intersect the stencil are also tested. This results in additional unnecessary stencil/triangle intersection tests in the per-point case. Data about the number of intersection tests performed with the per-point and per-element hash grids are detailed in Table 1.

Since a single point cannot span more than one cell, this allows for smaller cells which form a tighter bound around the stencil, and additionally, the elimination of the *halo* region. We found that setting the cell size equal to half the maximum triangle edge size produced good results. This method makes a tradeoff by reducing uncoalesced reads from sampling the unstructured mesh and increasing coalesced writes by splitting the solution in parts. Note that not every triangle tested will intersect with the stencil around the grid point. Only true positive intersections will be integrated.

Mesh Size	# of Per-Point Intersection Tests	# of Per-Element Intersection Tests
4k	6647394	3525297
16k	26492809	14235618
64k	110778427	59277119
256k	455614318	243245703
1024k	1919070326	1017924543

Table 1: Number of intersection tests performed with the per-point and per-element methods using linear polynomials

3.3 Per-Point Evaluation

To evaluate a stencil computation with the per-point method, a stencil is centered around each grid point and the intersections between that stencil and the underlying mesh geometry are found. When determining the mesh/stencil intersection,

we first determine the intersection between the hash grid and the stencil. A bounded region on the hash grid is determined by centering the stencil at the grid point and expanding the borders to nearest cell boundary in each dimension, as denoted in Equation (3). Next, each element within the bounded cells is tested for intersections. Intersected regions are then triangulated with the Sutherland-Hodgman algorithm and integrated. The set of *halo* cells around the bounded region must be included to ensure that all intersecting regions are tested. Algorithm 2 provides psuedo-code for the per-point evaluation method. The element data requires a minimum of $\frac{(P+1)(P+2)}{2} + 3$ values to be read from memory per integration, where P is the polynomial order.

Algorithm 2: Per-Point Post Processing

```

foreach Point  $p$  do
  // Compute hash grid bounds
   $L, R, T, B \leftarrow$  PointHashGridBounds( $p$ );
  foreach Cell  $j$  within bounds  $L, R, T, B$  do
    foreach Element  $e$  in Cell  $j$  do
      // Compute and store per-element data
       $ED \leftarrow$  ElementData();
      // Compute and triangulate
      stencil/element intersections
       $Regions \leftarrow$  SH(Stencil( $p$ ),  $e$ );
      // Integrate triangulated regions
       $Solution[p] \leftarrow$  Solution[ $p$ ] +
      Integrate(Regions,  $ED$ );
    end
  end
end

```

3.4 Per-Element Evaluation

The per-element evaluation scheme groups sample points by the underlying geometric element in which they happen to fall. The per-element stencil bounds, denoted in Equation (3), enclose an area that includes all grid points which have stencil intersections with the bounding box of the triangle. From this bounded area, the set of grid points whose stencils intersect the triangle are determined. Each grid point that falls within this region is tested for a stencil/triangle intersection using the given triangle element. The evaluation points within the triangle are then processed concurrently. The per-element scheme breaks up Equation (2) into partial solutions. The partial solutions are grouped together by triangular element, and each element will contribute partial solutions to every grid point whose stencil intersects that triangle. We divide the mesh into patches, the details of which are described in the next section, with the solution of each patch being accumulated into a separate memory space. Algorithm 3 provides the psuedo-code for the per-element evaluation method.

Data associated with the given element, such as the elemental coefficients and the bounds of the triangle, can be stored and reused for all evaluations. This takes advantage of data locality and leads to more coalesced memory accesses than in the per-point scheme. In the per-element case, only the spatial offset of the grid point (two values in 2D) are required to be read per integration, since the $(\frac{(P+1)(P+2)}{2} + 3)$ values associated with the triangle are reused for all integrations over that element.

Algorithm 3: Per-Element Post Processing

```
foreach Element  $e$  do
  // Compute hash grid bounds
   $L, R, T, B \leftarrow \text{ElementHashGridBounds}(e)$ ;
  // Compute and store element data in Shared
  Memory
   $ED \leftarrow \text{ElementData}()$ ;
  foreach Cell  $j$  within bounds  $L, R, T, B$  do
    foreach Point  $p$  in Cell  $j$  do
      // Compute and triangulate
      stencil/element intersections
       $\text{Regions} \leftarrow \text{SH}(\text{Stencil}(p), e)$ ;
      // Integrate triangulated regions
       $\text{PSolution}[\text{patch}(e), p] \leftarrow \text{PSolution}[\text{patch}(e), p] + \text{Integrate}(\text{Regions}, ED)$ ;
    end
  end
end
// Perform reduction on solution by patch
 $\text{Solution} \leftarrow \text{Reduction}(\text{PSolution})$ 
```

4. IMPLEMENTATION

The Sutherland-Hodgman algorithm presents a challenge in efficiently post-processing on many-core architectures. The highly divergent nature of the intersection processing, caused by branching logic, may lead to sub-optimal performance on streaming SIMD architectures. 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, due to noncontiguous memory accesses and thread divergence. Minimizing the total number of intersection tests is key to achieving high performance with stencil computations over unstructured meshes on SIMD architectures.

In the per-point method we assign a block to compute the solution for a given grid point. On the GPU we use a number of blocks equal to the SM count on the GPU (N_{SM}). The blocks then iterate over the points in a strided fashion (*i.e.* P_{i+k*N_B} , where P_i is the i^{th} point, N_B is the number of concurrent blocks, and k is an incrementing integer). Within a block, we assign threads to iterate over the element indices that lie within intersected cells of the hash grid in a similar strided fashion. The stencil/element intersections are then tested and integrated. There is no contention between stencils, as each stencil updates a discrete grid point. In this case it is trivial to achieve perfect load balancing between all processing groups. In the per-element method we assign a block to each patch. The threads within the blocks iterate over the points stored within the intersected cells of the hash grid in a strided manner. To maximize parallelism, we chose a number of blocks equal to the number of SMs per card. For multi-GPU decomposition we divide the mesh into $N_{GPU} \times N_{SM}$ patches, where N_{GPU} is the number of GPUs. In the multi-GPU implementation we use a two stage reduction. In the first stage, each GPU computes a reduction on the patches that it processed. This is followed by a final reduction of those resulting solutions in the second stage.

The per-element evaluation scheme requires that concurrent execution of stencil tiles acting on the same memory space do not overlap. Overlapping stencils may introduce race conditions where the value of a grid point is being updated by multiple stencils. To solve this problem, we assign a separate scratch pad memory space to each concurrent stencil tile where the partial solutions are accumulated. After all the stencils have finished their computations, the final solution is summed together from all the partial solutions. This requires additional memory space, but allows for maximum parallelism without the need for pipe-lining of the stencils.

We implemented a spatially overlapped tiling scheme, introduced in [13], where each tile uses a disjoint memory working set. Each logical block is assigned to process stencils in a localized patch of the mesh. The partial solutions of each patch are stored in a separate scratch pad memory space. This requires that grid points lying along the borders of patches have multiple partial solutions. Grid points that fall within the intersection of stencils from multiple patches will have a partial solution stored in each of those patches memory sets. Grid points that lie in the interior of a patch and only fall within stencils from that patch will have a single solution in memory. Figure 7 illustrates an example patch division and the partial solutions formed from the patches. The overlapped regions that lie within the intersection of stencils from multiple patches are summed together to produce the final result for those respective grid points. This leads to a relatively low amount of storage overhead. The memory overhead, relative to the memory requirement for the total solution, decreases as the mesh size increases.

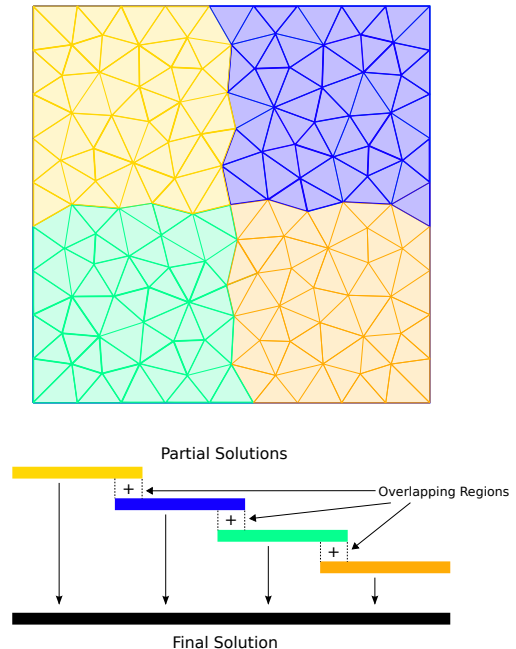


Figure 7: Example of mesh division into four patches

Patch construction follows from simple recursive bisection of the mesh elements until there are k patches of roughly equal size, with k being the number of concurrently executing blocks. This method easily scales with the mesh size. As the domain size increases, the number of concurrent stencils can be increased. Patch perimeter distance should be min-

imized in order to minimize the overall memory overhead. Increasing the number of tiles while decreasing the tile size has the effect of increasing overall memory overhead, but allows for higher parallelism. The number of concurrent executing tiles has a maximum upper bound equal to the number of geometric elements in the mesh. As the surface area of a patch grows at a faster rate than the perimeter, the memory overhead tends to be relatively low for large meshes. This also naturally extends to 3D with the memory overhead determined by the surface area to volume ratios of the patches.

The baseline memory consumption is the minimum amount of memory required to store the solution at all the evaluation grid points. The patch based tiling method adds additional memory overhead based on the number of grid points that fall within the intersection of stencils from multiple patches. Each patch stores partial solutions for every grid point that falls within the union of intersections spaces of the elements contained within the patch. Thus only points near the boundaries of patches will require storing multiple partial solutions. The ratio of boundary length to patch area decreases inversely proportional to mesh size for a fixed number of patches. Figure 8 illustrates the scaling of memory overhead across the range of test meshes. The perimeter of a patch grows linearly while the surface area grows quadratically. As the results demonstrate, this adds relatively little overhead memory consumption for larger meshes.

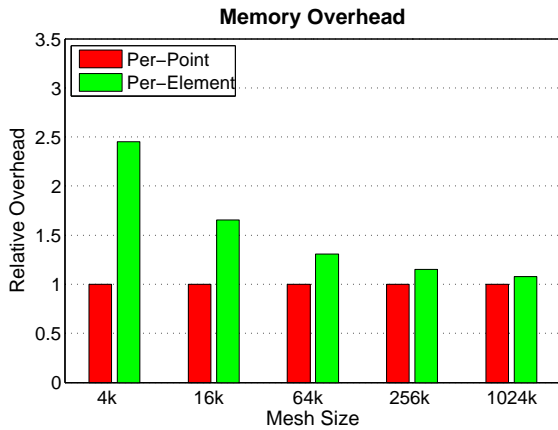


Figure 8: Memory overhead of per-element method using 16 patches with linear polynomials

The final summation of the partial solutions only requires a linear reduction based on the memory offset of each patch solution. In the reduction phase, we divide up the grid points based on the patch they fall within. We then assign a block to each patch which performs the reduction on the partial solutions for those grid points. This eliminates write contention to the final solution space. The process contributes a minimal amount of time to the overall process. We also explored a pipe-lined tiling method, but this introduces additional synchronizations between pipeline stages. There is no additional memory overhead introduced by pipe-lining, but there is reduction in overall performance.

5. EXPERIMENTAL RESULTS

In this section we evaluate the performance of GPU implementations of the per-point and per-element methods. In

addition, we demonstrate the scalability of our approach on 1, 2, 4, and 8 GPUs. We ran our tests on a node with two Intel Xeon E5630 processors (4 cores each) running at 2.53GHz, 128GB of memory, and eight NVIDIA Tesla M2090 GPUs using CUDA 5.0. We executed the tests across a series of 2D unstructured triangular meshes created using Delaunay triangulation. We tested our implementations on two different types of meshes. The first was an unstructured mesh with roughly uniform sized triangles, shown in Figure 9. The second type was an unstructured mesh with highly varying element sizes, shown in Figure 10. We tested each of these mesh types across mesh sizes on the order of 4k, 16k, 64k, 256k, and 1024k triangles. We used periodic boundary conditions with linear, quadratic, and cubic polynomials, which have three, six, and ten coefficients respectively for triangular elements. All tests were conducted with double precision floating point values.

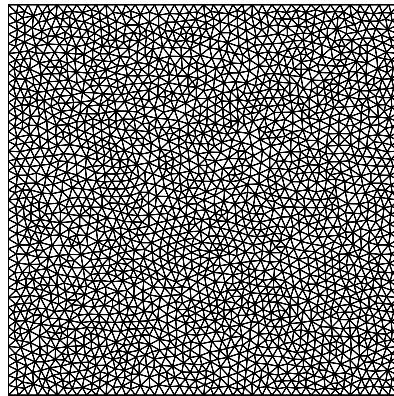


Figure 9: Unstructured mesh with low variance

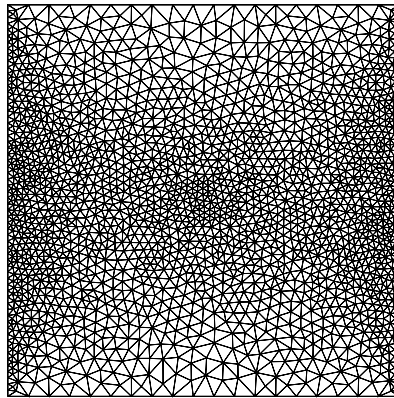


Figure 10: Unstructured mesh with high variance

The post-processing is divided into two main components. The first of which finds the intersections between the stencils and the underlying mesh geometry, and the second which integrates those subregions and accumulates the results. The intersection finding has linear complexity with respect to the number of intersection tests performed, while the integral calculation has a computational complexity on the order of $O((P + 1)^d)$, where P is the polynomial order used in the post-processing of the finite element solution and d is the dimension. The higher computational complexity of integration calculation dominates the overall run-time as the

polynomial order increases. This is demonstrated by the smaller performance increase between the per-point and per-element evaluation scheme for quadratic and cubic polynomials.

5.1 Metrics

Figure 11 provides FLOP metrics for the GPU over low-variance meshes. The per-element method achieves a peak FLOP rating of 345 GFLOP/s for linear polynomials on the 1024k mesh. For quadratic and cubic polynomials, the FLOP ratings are lower, but the relative difference between the methods is larger. For quadratic polynomials, the methods achieve a peak FLOP rating between 50 - 120 GFLOP/s, while for cubic polynomials a peak rating of 30 - 60 GFLOP/s is seen. The computational complexity of the integral kernel grows quadratically with respect to the polynomial order. As polynomial order grows, the integral kernel occupies a larger percent of the total run-time and the ratio of time spent computing intersections to time spent performing integrations decreases. In addition the integration kernel requires storage of a large number of intermediate values that grow on the order of $O((P + 1)^2)$. These constraints lead to a lower FLOP performance at higher polynomial orders.

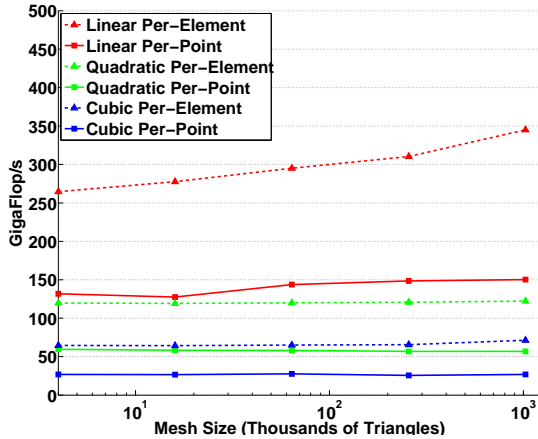


Figure 11: GPU Flop/s

Figure 12 provides GPU flop ratings for high variance meshes. The difference in FLOP performance between the two methods is more noticeable on meshes with high variance in element size. This is due in part to the fact that the search area for the per-point method includes a halo region that has a cell width equal to the largest element size. This has significantly more impact on performance than in the case of meshes with low variance in element size.

The results in Figure 13 illustrate the relative performance of the per-point and per-element method for low and high variance meshes. The timings of the per-point methods have been normalized. The performance difference between the per-element and per-point methods is greater on meshes with high variance in element sizes. The per-element method achieves over a $2\times$ speedup for the low-variance mesh with cubic polynomials, and over a $3\times$ speedup for the high-variance mesh.

The results demonstrate a significant performance improvement of the per-element evaluation scheme over the per-point scheme for many-core architectures. Local data associated with each element is accessed only once and reused

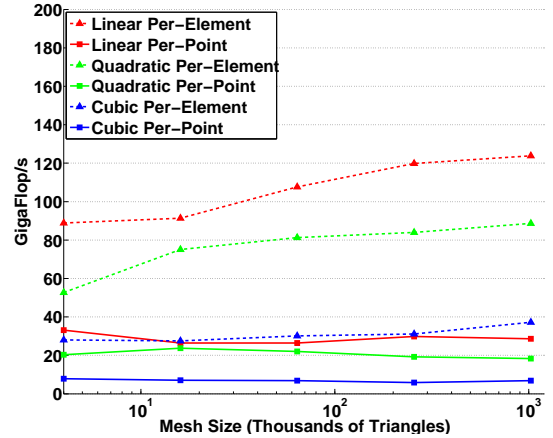


Figure 12: GPU Flop/s

for all evaluations within the element. The heterogeneity of the unstructured mesh leads to irregular memory access patterns and uncoalesced memory accesses. Fewer intersection tests combined with increased data reuse contribute to increased performance. The results provide insight into the performance of each evaluation method on many-core architectures. The streaming many-core architecture of the GPU benefits greatly from reduced intersection tests and increased data reuse of the local element information, in part due to the relatively low amount of cache per core. We also implemented a single threaded CPU version of the methods. We noticed that implementations with low levels of concurrency see less benefit from data reuse. The improvement of per-element evaluation over per-point evaluation is less significant, and in a few cases even worse due to the increased overhead.

5.2 Scaling

To demonstrate the scaling of the per-element method, we tested the per-element evaluation method on 1, 2, 4, and 8 GPUs across the entire range of our test meshes. The results demonstrate that the method has perfect linear scaling with respect to increased mesh size. This is to be expected for a problem with outer parallelism where there is no inherent dependencies between grid points. Figure 14 illustrates the scaling of the GPU per-element method across the range of test meshes for linear polynomials. Parallelization across GPUs was achieved by subdividing the mesh into the $N_{GPU} \times N_{SM}$ patches and evenly distributing them between the GPUs.

6. CONCLUSION

In this paper, we have introduced an efficient, scalable scheme for evaluating stencil computations over unstructured meshes. We present two general strategies for evaluating stencil computations over unstructured meshes, per-point and per-element. In addition, we present a scalable overlapped tiling method which allows for concurrent execution of stencils. We implemented a discontinuous Galerkin post-processor for 2D unstructured triangular meshes using both per-point and per-element evaluation schemes. We compare each approach in the context of memory efficiency and overall performance. Further, we compared the per-

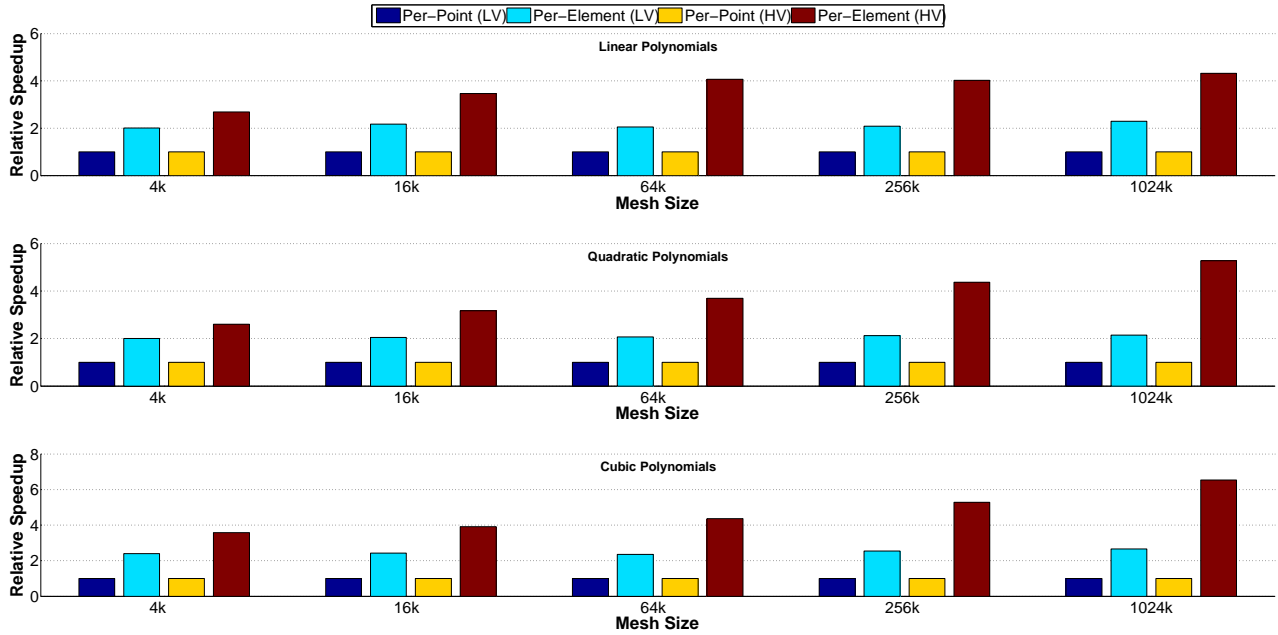


Figure 13: Relative speedup compared to a normalized per-point methods for low variance (LV) and high variance (HV) meshes

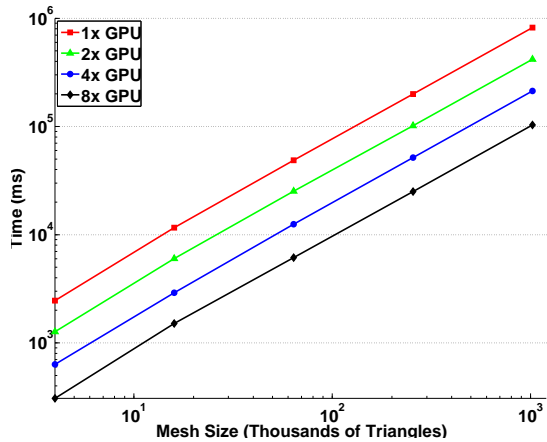


Figure 14: Scaling of the per-element method on 1, 2, 4, and 8 GPUs with linear polynomials

point and per-element evaluation schemes across unstructured meshes with low and high variance, and we demonstrated the scalability of the per-element scheme to multiple GPUs.

The results of our tests show that increased data-reuse and data locality has a significant impact on the performance of stencil computations over unstructured meshes with high levels of concurrency. On the GPU, the per-element method exhibits between a $2\times$ - $6\times$ performance improvement over the per-point counterpart. The technique of homogenizing similar operations by their associated geometric element on unstructured meshes leads to significantly increased performance on many-core architectures like the GPU. The per-element method demonstrates perfect linear scaling as the number of computing cores increases. The overlapped tiling

method we employ allows for nearly perfect linear scaling with minimal synchronization overhead. The per-element method adds some memory overhead to the process, but significantly improves overall performance.

Future opportunities for research include the extension of these ideas to 3D over unstructured tetrahedral meshes. The overlapped tiling methodology with partial solutions could be extended to 3D as the volume of the patches grows at a faster rate than the surface area. In addition, the methodology we present is general, and need not be constrained to only dG post-processing. Our technique could be extended to methods operating over unstructured meshes which compute the values over an element based upon linear or non-linear combinations of values from spatially neighboring elements. This includes methods such as weighted essentially non-oscillatory (WENO) spatial filtering, radial basis function finite differences (RBF-FD), and narrow-band schemes for solving level set equations in parallel.

Acknowledgments

The authors would like to thank Dr. Sergey Yakovlev for comments and suggestions. This work is funded in part by the Air Force Office of Scientific Research (AFOSR), Computational Mathematics Program (Program Manager: Dr. Fariba Fahroo), under grant number FA9550-12-1-0428 and by the Department of Energy (DOE NETL DE-EE0004449).

7. REFERENCES

- [1] M. Arenaz, J. Touriño, and R. Doallo. An inspector-executor algorithm for irregular assignment parallelization. In *In Proc. of the 2nd International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, 2005.
- [2] V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [3] J. L. Bentley and J. H. Friedman. Data Structures for Range Searching. *ACM Comput. Surv.*, 11(4):397–409, Dec. 1979.
- [4] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. In *Proceedings of the IEEE International Symposium on Workload Characterization, IISWC '12*, 2012.
- [5] L.-W. Chang, J. A. Stratton, H.-S. Kim, and W.-M. W. Hwu. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, pages 27:1–27:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [6] M. Christen, O. Schenk, and H. Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687, 2011.
- [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*, pages 291–300. Springer, 1999.
- [8] B. Cockburn, M. Luskin, C.-W. Shu, and E. Süli. Enhanced accuracy by post-processing for finite element methods for hyperbolic equations. *Mathematics of Computation*, 72:577–606, 2003.
- [9] N. Corporation. *CUDA C Best Practices Guide*. NVIDIA, 2012.
- [10] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08*, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.
- [11] J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing, ICS '12*, pages 311–320, New York, NY, USA, 2012. ACM.
- [12] Khronos Group. *The OpenCL Specification*, Sept. 2011.
- [13] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07*, pages 235–244, New York, NY, USA, 2007. ACM.
- [14] T. Malas, A. J. Ahmadi, J. Brown, J. A. Gunnels, and D. E. Keyes. Optimizing the performance of streaming numerical kernels on the IBM Blue Gene/P PowerPC 450 processor. *International Journal of High Performance Computing Applications*, 27(2):193–209, May 2013.
- [15] D. J. Mavriplis. Unstructured Grid Techniques. *Annual Review of Fluid Mechanics*, 29(1):473–514, 1997.
- [16] H. Mirzaee, L. Ji, J. K. Ryan, and R. M. Kirby. Smoothness-Increasing Accuracy-Conserving (SIAC) Post-Processing for Discontinuous Galerkin solutions over structured Triangular Meshes. *SIAM Journal of Numerical Analysis*, 49:1899–1920, 2011.
- [17] H. Mirzaee, J. King, J. Ryan, and R. Kirby. Smoothness-Increasing Accuracy-Conserving Filters for Discontinuous Galerkin Solutions over Unstructured Triangular Meshes. *SIAM Journal on Scientific Computing*, 35(1):A212–A230, 2013.
- [18] H. Mirzaee, J. K. Ryan, and R. M. Kirby. Efficient Implementation of Smoothness-Increasing Accuracy-Conserving (SIAC) Filters for Discontinuous Galerkin Solutions. *Journal of Scientific Computing*, 2011.
- [19] NVIDIA. *CUDA C Programming Guide v5.0*. NVIDIA, 2012.
- [20] M. Rietmann, P. Messmer, T. Nissen-Meyer, D. Peter, P. Basini, D. Komatitsch, O. Schenk, J. Tromp, L. Boschi, and D. Giardini. Forward and adjoint simulations of seismic wave propagation on emerging large-scale GPU architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, pages 38:1–38:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
- [21] J. K. Ryan and C.-W. Shu. On a one-sided post-processing technique for the discontinuous Galerkin methods. *Methods and Applications of Analysis*, 10:295–307, 2003.
- [22] J. K. Ryan, C.-W. Shu, and H. L. Atkins. Extension of a post-processing technique for the discontinuous Galerkin method for hyperbolic equations with application to an aeroacoustic problem. *SIAM Journal on Scientific Computing*, 26:821–843, 2005.
- [23] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [24] L. Solano-Quinde, B. Bode, and A. K. Somani. Techniques for the parallelization of unstructured grid applications on multi-GPU systems. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM '12*, pages 140–147, New York, NY, USA, 2012. ACM.
- [25] L. Solano-Quinde, Z. J. Wang, B. Bode, and A. K. Somani. Unstructured grid applications on GPU: performance analysis and improvement. In *Proceedings*

- of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-4*, pages 13:1–13:8, New York, NY, USA, 2011. ACM.
- [26] 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 Transactions on Visualization and Computer Graphics*, 14(3):680–692, 2008.
- [27] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.
- [28] 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. *Journal of Scientific Computing*, 38(2):164–184, 2009.
- [29] Y. Zhang and F. Mueller. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12*, pages 155–164, New York, NY, USA, 2012. ACM.