J Sci Comput
DOI 10.1007/s10915-013-9805-x

Exploiting Batch Processing on Streaming Architectures to Solve 2D Elliptic Finite Element Problems: A Hybridized Discontinuous Galerkin (HDG) Case Study

James King · Sergey Yakovlev · Zhisong Fu · Robert M. Kirby · Spencer J. Sherwin

Received: 14 March 2013 / Revised: 9 September 2013 / Accepted: 8 November 2013
© Springer Science+Business Media New York 2013

Abstract Numerical methods for elliptic partial differential equations (PDEs) within both continuous and hybridized discontinuous Galerkin (HDG) frameworks share the same general structure: local (elemental) matrix generation followed by a global linear system assembly and solve. The lack of inter-element communication and easily parallelizable nature of the local matrix generation stage coupled with the parallelization techniques developed for the linear system solvers make a numerical scheme for elliptic PDEs a good candidate for implementation on streaming architectures such as modern graphical processing units (GPUs). We propose an algorithmic pipeline for mapping an elliptic finite element method to the GPU and perform a case study for a particular method within the HDG framework. This study provides comparison between CPU and GPU implementations of the method as well as highlights certain performance-crucial implementation details. The choice of the HDG method for the case study was dictated by the computationally-heavy local matrix generation stage as well as the reduced trace-based communication pattern, which together make the method amenable to the fine-grained parallelism of GPUs. We demonstrate that the HDG method is

J. King · Z. Fu · R. M. Kirby (✉)
School of Computing and Scientific Computing and Imaging (SCI) Institute, University of Utah, Salt Lake City, UT, USA
e-mail: kirby@cs.utah.edu

J. King
e-mail: jsking@sci.utah.edu

Z. Fu
e-mail: zhisong@cs.utah.edu

S. Yakovlev
Scientific Computing and Imaging (SCI) Institute, University of Utah, Salt Lake City, UT, USA
e-mail: yakovs@sci.utah.edu

S. J. Sherwin
Department of Aeronautics, Imperial College London, London, UK
e-mail: s.sherwin@imperial.ac.uk

Published online: 26 November 2013

well-suited for GPU implementation, obtaining total speedups on the order of 30–35 times over a serial CPU implementation for moderately sized problems.

Keywords High-order finite elements · Spectral/*hp* elements · Discontinuous Galerkin method · Hybridization · Streaming processors · Graphical processing units (GPUs)

1 Introduction

In the last decade, commodity streaming processors such as those found in graphical processing units (GPUs) have arisen as a driving platform for heterogeneous parallel processing with strong scalability, power and computational efficiency [1]. In the past few years, a number of algorithms have been developed to harness the processing power of GPUs for a number of problems which require multi-element processing techniques [2,3]. This work is motivated by our attempt to find effective ways of mapping continuous and hybridized discontinuous Galerkin (HDG) methods to the GPU. Significant gains in performance have been made when combining GPUs with discontinuous Galerkin (DG) for hyperbolic problems (e.g. [4]); in this work, we focus on whether similar gains can be achieved when solving elliptic problems.

Note that within a hyperbolic setting, each time step of a DG method algorithmically consists of a single parallel update step where the inter-element communication is limited to the numerical flux computation that is performed locally. In the case of many elliptic operator discretizations, however, one is required to solve a linear system in order to find the values of globally coupled unknowns. The linear system in question can be reduced in size if static condensation (Schur Complement) technique is applied, but it has to be solved nevertheless. Depending on the choice of linear solver, the system matrix can either be explicitly assembled or stored as a collection of elemental matrices accompanied by the local-to-global mapping data. In this particular work we have chosen to explicitly assemble the system matrix on the GPU to match the CPU code used for comparison.

Due to the different structure of numerical methods for elliptic PDEs and the unavoidable global coupling of unknowns, one usually breaks the solution process into several of stages: local (elemental) matrix generation, global linear system matrix assembly, and global linear system solve. If static condensation is applied and the global linear system is solved for the trace solution (solution on the boundary of elements), there is an additional stage of recovering the elemental solution from the trace data. Each of the stages outlined above benefits from parallelization on the GPU to a different degree: the local matrix generation stage benefits from parallelization much more than the assembly and global solve stages, due to the fact that operations performed are completely independent for different elements.

The goals this paper pursues are the following: (a) to provide the reader with an intuition regarding the overall benefit that parallelization on streaming architectures provides to numerical methods for elliptic problems as well as per-stage benefits and the runtime trends for different stages; (b) to propose a pipeline for solving 2D elliptic finite element problems on GPUs and provide a case study to understand the benefits of GPU implementation for numerical problems formulated within the HDG framework; (c) to propose a per-edge assembly as a more efficient approach than the traditional per-element assembly, given the structure of the HDG method and the restrictions of the current generation of SIMD hardware. The key ingredients to our proposed approach are the mathematical nature of the HDG method and the batch processing capabilities (and algorithmic limitations) of the GPU. The choice of method for our case study is motivated by the fact that the local matrix generation stage, which benefits the most from parallelization, is much more computationally intensive for the

HDG method as opposed to the CG method. We now provide background concerning the HDG method and discuss the batch processing capabilities of the GPU.

1.1 Background

DG methods have seen considerable success in a variety of applications due to ease of implementation, ability to use arbitrary unstructured geometries, and suitability for parallelization. The local support of the basis functions in DG methods allows for domain decomposition at the element level which lends itself well to parallel implementations (e.g. [5,6]). A number of recent works have demonstrated that DG methods are well-suited for implementation on a GPU [7,8], for reasons of memory reference locality, regularity of access patterns, and dense arithmetic computations. Computational performance of DG methods is closely tied to polynomial order. As polynomial order increases on DG methods, memory bandwidth becomes less of a bottleneck as the floating point arithmetic operations become the dominant factor. The increase in floating point operation throughput on GPUs has led to implementations of high-order DG methods on the GPU [9].

However DG methods still suffer from and are often criticized for the need to employ significantly more degrees of freedom than other numerical methods [10], which results in a bigger global linear system to solve. The introduction of the HDG method in Cockburn et al. [11] successfully resolved this issue by providing a method within the DG framework whose only globally coupled degrees of freedom were those of the scalar unknown on the borders of the elements. The HDG method uses a formulation which expresses all of the unknowns in terms of the numerical trace of the hybrid scalar variable λ. This method greatly reduces the global linear system size, while maintaining properties that make DG methods apt to parallelization. The elemental nature of DG methods have encouraged many to assert that they should be "easily parallelizable" (e.g. [4,12,13]). Due to weak coupling between elements in the HDG method, there is less inter-element communication needed which is advantageous for scaling the method to a parallel implementation. The combination of a batch collection of local (elemental) problems which needs to be computed and the reduced trace-based communication pattern of HDG conceptually makes this method well-suited to the fine-grained parallelism of streaming architectures such as modern GPUs. It is the local (elemental) batch nature of the decomposition which directs us to investigate the GPU implementation of the method. In the next subsection we provide an overview of batched operations, describe the current state of batch processing in existing software packages, and explain why it was relevant to create our own batch processing framework.

1.2 Batched Operations

Batch processing is the act of grouping some number of like tasks and computing them as a "batch" in parallel. This generally involves a large set of data whose elements can be processed independently of each other. Batch processing eliminates much of the overhead of iterative non-batched operations. "Batch" processing is well-suited to GPUs due to the SIMD architecture which allows for high parallelization of large streams of data. Basic linear algebra subprograms (BLAS) are a common example of large scale operations that benefit significantly from batch processing. The HDG method specifically benefits from batched BLAS Level 2 (matrix–vector multiplication) and BLAS Level 3 (matrix–matrix multiplication) operations.

Finding efficient implementations for solving linear algebra problems is one of the most active areas of research in GPU computing. The NVIDIA CUBLAS [14] and AMD

APPML [15] are well-known solutions for BLAS functions on GPUs. While CUBLAS is specifically designed for the NVIDA GPU architecture based on CUDA [14], the AMD solution using OpenCL [16] is a more general cross platform solution for both GPU and multi-CPU architectures. CUBLAS has constantly improved based on a successive number of research attempts by Volkov [17], Dongarra [18,19] *etc.* This led to a speed improvement of one to two orders of magnitude for many functions from the first release version till now. In recent releases, CUBLAS and other similar packages have been providing batch processing support to improve processing efficiency on multi-element processing tasks. The support is, however, not complete as currently CUBLAS only supports batch mode processing for BLAS Level 3, but not for functions within BLAS Level 1 and BLAS Level 2.

It is due to the these limitations of existing software that the authors were prompted to create a batch processing framework. We developed a batch processing framework for the GPU which uses the same philosophy present in CUBLAS. However, we augmented it with additional operations such as matrix-vector multiplication and matrix inversion. The framework is generalized such that it is not limited specifically to linear algebra operations; however, due to the finite element context of this paper, we restricted our focus to linear algebra operations.

1.3 Outline

The paper is organized as follows. In Sect. 2 we present the mathematical formulation of the HDG method. In Sect. 3 we introduce all the necessary implementation building blocks: polynomial expansion bases, matrix form of the equations from Sect. 2, trace assembly and spread operators, *etc.* Sect. 4 and its subsections present details that are specific to GPU implementation of the HDG method. First we describe the implementation pipeline followed by the description of the local matrix generation in Sect. 4.1, the global system matrix assembly in Sect. 4.2, and the global solve and subsequent local solve in Sect. 4.3. In Sect. 5 we present numerical results which include a comparison of CPU and GPU implementations of HDG method. Finally, in Sect. 6 we conclude with potential directions for future research along with a summary of the results.

2 Mathematical Formulation of HDG

In this section we introduce the HDG method for the following elliptic diffusion problem with mixed Dirichlet and Neumann boundary conditions:

$$-\nabla^2 u(\boldsymbol{x}) = f(\boldsymbol{x}) \qquad \boldsymbol{x} \in \Omega, \tag{1a}$$

$$u(\boldsymbol{x}) = g_D(\boldsymbol{x}) \qquad \boldsymbol{x} \in \partial\Omega_D, \tag{1b}$$

$$\boldsymbol{n} \cdot \nabla u(\boldsymbol{x}) = g_N(\boldsymbol{x}) \qquad \boldsymbol{x} \in \partial\Omega_N, \tag{1c}$$

where $\partial\Omega_D \bigcup \partial\Omega_N = \partial\Omega$ and $\partial\Omega_D \bigcap \partial\Omega_N = \emptyset$. The formulation above can be generalized in many ways which can be treated in a similar manner. For example, by considering a diffusion tensor which is given by a symmetric positive definite matrix and by adding convection and reaction terms.

In Sects. 2.2–2.4 we define the HDG methods. We start by presenting the global weak formulation in Sect. 2.2. In Sect. 2.3, we define *local problems*: a collection of elemental operators that express the approximation inside each element in terms of the approximation at its border. Finally, we provide a *global* formulation with which we determine the approx-

imation on the border of the elements in Sect. 2.4. The resulting global boundary system is significantly smaller than the full system one would solve without solving *local problems* first. Once the solution has been obtained on the boundaries of the elements, the primary solution over each element can be determined independently through a forward-application of the elemental operators. However before proceeding we first define the partitioning of the domain and the finite element spaces in Sect. 2.1.

2.1 Partitioning of the Domain and the Spectral/*hp* Element Spaces

We begin by discretizing our domain. We assume $\mathcal{T}(\Omega)$ is a two-dimensional tessellation of Ω. Let $\Omega^e \in \mathcal{T}(\Omega)$ be a non-overlapping element within the tessellation such that if $e_1 \neq e_2$ then $\Omega^{e_1} \bigcap \Omega^{e_2} = \emptyset$. By N_{el}, we denote the number of elements (or cardinality) of $\mathcal{T}(\Omega)$. Let $\partial\Omega^e$ denote the boundary of the element Ω^e (*i.e.* $\bar{\Omega}^e \setminus \Omega^e$) and $\partial\Omega^e_i$ denote an individual edge of $\partial\Omega^e$ such that $1 \leq i \leq N^e_b$ where N^e_b denotes the number of edges of element e. We then denote by Γ the set of boundaries $\partial\Omega^e$ of all the elements Ω^e of $\mathcal{T}(\Omega)$. Finally, we denote by N_Γ the number of edges (or cardinality) of Γ.

For simplicity, we assume that the tessellation $\mathcal{T}(\Omega)$ consists of conforming elements. Note that HDG formulation can be extended to non-conforming meshes. We do not consider the case of a non-conforming mesh in this work, as it would complicate the implementation while not enhancing the contribution statement in any way. We say that Γ^l is an *interior edge* of the tessellation $\mathcal{T}(\Omega)$ if there are two elements of the tessellation, Ω^e and Ω^f, such that $\Gamma^l = \partial\Omega^e \cap \partial\Omega^f$ and the length of Γ^l is not zero. We say that Γ^l is a *boundary edge* of the tessellation $\mathcal{T}(\Omega)$ if there is an element of the tessellation, Ω^e, such that $\Gamma^l = \partial\Omega^e \cap \partial\Omega$ and the length of Γ^l is not zero.

As it will be useful later, let us define a collection of index mapping functions, that allow us to relate the local edges of an element Ω^e, namely, $\partial\Omega^e_1, \ldots, \partial\Omega^e_{N^e_b}$, with the global edges of Γ, that is, with $\Gamma^1, \ldots, \Gamma^{N_\Gamma}$. Thus, since the jth edge of the element Ω^e, $\partial\Omega^e_j$, is the lth edge Γ^l of the set of edges Γ, we set $\sigma(e, j) = l$ so that we can write $\partial\Omega^e_j = \Gamma^{\sigma(e,j)}$.

Next, we define the finite element spaces associated with the partition $\mathcal{T}(\Omega)$. To begin, for a two-dimensional problem we set

$$
\begin{aligned}
V_h &:= \{v \in L^2(\Omega) : \ v|_{\Omega^e} \in P(\Omega^e) \quad \forall\, \Omega^e \in \mathcal{T}(\Omega)\}, && \text{(2a)}\\
\boldsymbol{\Sigma}_h &:= \{\tau \in [L^2(\Omega)]^2 : \ \tau|_{\Omega^e} \in \boldsymbol{\Sigma}(\Omega^e) \quad \forall\, \Omega^e \in \mathcal{T}(\Omega)\}, && \text{(2b)}\\
\mathcal{M}_h &:= \{\mu \in L^2(\Gamma) : \ \mu|_{\Gamma^l} \in P(\Gamma^l) \quad \forall\, \Gamma^l \in \Gamma\}, && \text{(2c)}
\end{aligned}
$$

where $P(\Gamma^l) = \mathcal{S}_P(\Gamma^l)$ is the polynomial space over the standard segment, $P(\Omega^e) = \mathcal{T}_P(\Omega^e)$ is the space of polynomials of total degree P defined on a standard triangular region and $P(\Omega^e) = \mathcal{Q}_P(\Omega^e)$ is the space of tensor-product polynomials of degree P on a standard quadrilateral region, defined as

$$
\begin{aligned}
\mathcal{S}_P(\Gamma^l) &= \{s^p; \quad 0 \leq p \leq P; (x_1(s), x_2(s)) \in \Gamma^l; -1 \leq s \leq 1\},\\
\mathcal{T}_P(\Omega^e) &= \{\xi_1^p \xi_2^q; 0 \leq p + q \leq P; (x_1(\xi_1, \xi_2), x_2(\xi_1, \xi_2)) \in \Omega^e; -1 \leq \xi_1 + \xi_2 \leq 0\},\\
\mathcal{Q}_P(\Omega^e) &= \{\xi_1^p \xi_2^q; 0 \leq p, q \leq P; (x_1(\xi_1, \xi_2), x_2(\xi_1, \xi_2)) \in \Omega^e; -1 \leq \xi_1, \xi_2 \leq 1\}.
\end{aligned}
$$

Similarly $\boldsymbol{\Sigma}(\Omega^e) = [\mathcal{T}_P(\Omega^e)]^2$ or $\boldsymbol{\Sigma}(\Omega^e) = [\mathcal{Q}_P(\Omega^e)]^2$. For curvilinear regions the expansions are only polynomials when mapped to a straight-sided standard region [20,21].

2.2 The HDG Method

The HDG method is defined in the following way. We start by rewriting the original problem (1) in auxiliary or mixed form as two first-order differential equations by introducing an auxiliary flux variable $\boldsymbol{q} = \nabla u$. This gives us:

$$-\nabla \cdot \boldsymbol{q} = f(\boldsymbol{x}) \quad \boldsymbol{x} \in \Omega, \tag{3a}$$

$$\boldsymbol{q} = \nabla u(\boldsymbol{x}) \quad \boldsymbol{x} \in \Omega, \tag{3b}$$

$$u(\boldsymbol{x}) = g_D(\boldsymbol{x}) \quad \boldsymbol{x} \in \partial\Omega_D, \tag{3c}$$

$$\boldsymbol{q} \cdot \boldsymbol{n} = g_N(\boldsymbol{x}) \quad \boldsymbol{x} \in \partial\Omega_N. \tag{3d}$$

The HDG method seeks an approximation to $(u, \boldsymbol{q})$, $(u^{\mathrm{DG}}, \boldsymbol{q}^{\mathrm{DG}})$, in the space $V_h \times \boldsymbol{\Sigma}_h$, and determines it by requiring that

$$\sum_{\Omega^e \in \mathcal{T}(\Omega)} \int_{\Omega^e} (\nabla v \cdot \boldsymbol{q}^{\mathrm{DG}})\, d\boldsymbol{x} - \sum_{\Omega^e \in \mathcal{T}(\Omega)} \int_{\partial\Omega^e} v\, (\boldsymbol{n}^e \cdot \widetilde{\boldsymbol{q}}^{\mathrm{DG}})\, ds = \sum_{\Omega^e \in \mathcal{T}(\Omega)} \int_{\Omega^e} v\, f\, d\boldsymbol{x}, \tag{4a}$$

$$\sum_{\Omega^e \in \mathcal{T}(\Omega)} \int_{\Omega^e} (\boldsymbol{w} \cdot \boldsymbol{q}^{\mathrm{DG}})\, d\boldsymbol{x} = - \sum_{\Omega^e \in \mathcal{T}(\Omega)} \int_{\Omega^e} (\nabla \cdot \boldsymbol{w})\, u^{\mathrm{DG}}\, d\boldsymbol{x} + \sum_{\Omega^e \in \mathcal{T}(\Omega)} \int_{\partial\Omega^e} (\boldsymbol{w} \cdot \boldsymbol{n}^e)\, \widetilde{u}^{\mathrm{DG}}\, ds, \tag{4b}$$

for all $(v, \boldsymbol{w}) \in V_h(\Omega) \times \boldsymbol{\Sigma}_h(\Omega)$, where the numerical traces $\widetilde{u}^{\mathrm{DG}}$ and $\widetilde{\boldsymbol{q}}^{\mathrm{DG}}$ are defined in terms of the approximate solution $(u^{\mathrm{DG}}, \boldsymbol{q}^{\mathrm{DG}})$.

2.3 Local Problems of the HDG Method

We begin by assuming that the function

$$\lambda := \widetilde{u}^{\mathrm{DG}} \in \mathcal{M}_h, \tag{5a}$$

is known, for any element Ω^e, from the global formulation of the HDG method. The restriction of the HDG solution to the element Ω^e, $(u^e, \boldsymbol{q}^e)$ is then the function in $P(\Omega^e) \times \boldsymbol{\Sigma}(\Omega^e)$ and satisfies the following Equations:

$$\int_{\Omega^e} (\nabla v \cdot \boldsymbol{q}^e)\, d\boldsymbol{x} - \int_{\partial\Omega^e} v\, (\boldsymbol{n}^e \cdot \widetilde{\boldsymbol{q}}^e)\, ds = \int_{\Omega^e} v\, f\, d\boldsymbol{x}, \tag{5b}$$

$$\int_{\Omega^e} (\boldsymbol{w} \cdot \boldsymbol{q}^e)\, d\boldsymbol{x} = - \int_{\Omega^e} (\nabla \cdot \boldsymbol{w})\, u^e\, d\boldsymbol{x} + \int_{\partial\Omega^e} (\boldsymbol{w} \cdot \boldsymbol{n}^e)\, \lambda\, ds, \tag{5c}$$

for all $(v, \boldsymbol{w}) \in P(\Omega^e) \times \boldsymbol{\Sigma}(\Omega^e)$. To allow us to solve the above equations locally, the numerical trace of the flux is chosen in such a way that it depends only on λ and on $(u^e, \boldsymbol{q}^e)$:

$$\widetilde{\boldsymbol{q}}^e(\boldsymbol{x}) = \boldsymbol{q}^e(\boldsymbol{x}) - \tau(u^e(\boldsymbol{x}) - \lambda(\boldsymbol{x}))\boldsymbol{n}^e \qquad \text{on } \partial\Omega^e \tag{5d}$$

where τ is a positive function. For the HDG method taking τ to be positive ensures that the method is well defined. The results in [22–24] indicate that the best choice is to take τ to be of order one. Note that τ is a function of the set of borders of the elements of the discretization, and so, it is allowed to be different per element and per edge. Thus, if we are dealing with the element whose *global number* is e, we denote the value of τ on the edge whose *local number* is i by $\tau^{e,i}$.

2.4 The Global Formulation for λ

Here we denote the solution of (5b)–(5c) when $f = 0$ and when $\lambda = 0$ by $(U_\lambda, \boldsymbol{Q}_\lambda)$ and $(U_f, \boldsymbol{Q}_f)$, respectively, and define our approximation to be

$$(u^{\mathrm{HDG}}, \boldsymbol{q}^{\mathrm{HDG}}) = (U_\lambda, \boldsymbol{Q}_\lambda) + (U_f, \boldsymbol{Q}_f).$$

Note that for the HDG decomposition allows us to express U_λ, $\boldsymbol{Q}_\lambda$ in terms of λ when $f = 0$.

It remains to determine λ. To do so, we require that the boundary conditions be weakly satisfied *and* that the normal component of the numerical trace of the flux $\widetilde{\boldsymbol{q}}$ given by (5d) be single valued. This renders this numerical trace *conservative*, a highly valued property for this type of methods; see Arnold et al. [25].

So, we say that λ is the function in $\mathcal{M}_h$ such that

$$\lambda = \mathsf{P}_h(g_D) \qquad \text{on } \partial\Omega_D, \tag{6a}$$

$$\sum_{\Omega_e \in \mathcal{T}_h} \int_{\partial\Omega^e} \mu\, \widetilde{\boldsymbol{q}} \cdot \boldsymbol{n} = \int_{\partial\Omega_N} \mu\, g_N, \tag{6b}$$

for all $\mu \in \mathcal{M}_h^0$ such that $\mu = 0$ on $\partial\Omega_D$. Here P_h denotes the L^2-projection into the space of restrictions to $\partial\Omega_D$ of functions of $\mathcal{M}_h$.

3 HDG Discrete Matrix Formulation and Implementation Considerations

In this section, to get a better appreciation of the implementation of the HDG approach, we consider the matrix representation of the HDG equations. The intention here is to introduce the notation and provide the basis for the discussion in the following sections. More details regarding the matrix formulation can be found in Kirby et al. [26].

We start by taking $u^e(\boldsymbol{x})$, $\boldsymbol{q}^e(\boldsymbol{x}) = [q_1, q_2]^T$, and $\lambda^l(\boldsymbol{x})$ to be finite expansions in terms of the basis $\phi_j^e(\boldsymbol{x})$ for the expansions over elements and the basis $\psi_j^l(\boldsymbol{x})$ over the traces of the form:

$$u^e(\boldsymbol{x}) = \sum_{j=1}^{N_u^e} \phi_j^e(\boldsymbol{x})\, \underline{\hat{u}}^e[j] \qquad q_k^e(\boldsymbol{x}) = \sum_{j=1}^{N_q^e} \phi_j^e(\boldsymbol{x})\, \underline{\hat{q}}_k^e[j] \qquad \lambda^l(\boldsymbol{x}) = \sum_{j=1}^{N_\lambda^l} \psi_j^l(\boldsymbol{x})\, \underline{\hat{\lambda}}^l[j],$$

where $u^e(\boldsymbol{x}) : \Omega^e \to \mathbb{R}$, $\mathbf{q}^\mathbf{e}(\boldsymbol{x}) : \Omega^e \to \mathbb{R}^2$ and $\lambda^l(\boldsymbol{x}) : \Gamma^l \to \mathbb{R}$.

In our numerical implementation, we have applied a spectral/*hp* element type discretization which is described in detail in Karniadakis and Sherwin [20]. In this work we use the modified Jacobi polynomial expansions on a triangle in the form of generalized tensor products. This expansion was originally proposed by Dubiner [27] and is also detailed in Karniadakis and Sherwin [20], Sherwin and Karniadakis [21]. We have selected this basis due to computational considerations: tensorial nature of the basis coupled with the decomposition into an *interior* and *boundary* modes [20,21] benefits the HDG implementation. In particular, when computing a boundary integral of an elemental basis function, edge basis function together with edge-to-element mapping can be used. This fact will be further commented upon in the following sections.

3.1 Matrix Form of the Equations of the HDG Local Solvers

We can now define the matrix form of the local solvers. Following a standard Galerkin formulation, we set the scalar test functions v^e to be represented by $\phi_i^e(\boldsymbol{x})$ where $i = 1, \ldots, N_u^e$, and let our vector test function $\boldsymbol{w}^e$ be represented by $\boldsymbol{e}_k\phi_i$ where $\boldsymbol{e}_1 = [1, 0]^T$ and $\boldsymbol{e}_2 = [0, 1]^T$. We next define the following matrices:

$$\mathbb{D}_k^e[i,j] = \left(\phi_i^e, \frac{\partial \phi_j^e}{\partial x_k}\right)_{\Omega^e} \quad \mathbb{M}^e[i,j] = \left(\phi_i^e, \phi_j^e\right)_{\Omega^e}$$
$$\mathbb{E}_l^e[i,j] = \left\langle \phi_i^e, \phi_j^e \right\rangle_{\partial\Omega_l^e} \quad \widetilde{\mathbb{E}}_{kl}^e[i,j] = \left\langle \phi_i^e, \phi_j^e n_k^e \right\rangle_{\partial\Omega_l^e}$$
$$\mathbb{F}_l^e[i,j] = \left\langle \phi_i^e, \psi_j^{\sigma(e,l)} \right\rangle_{\partial\Omega_l^e} \quad \widetilde{\mathbb{F}}_{kl}^e[i,j] = \left\langle \phi_i^e, \psi_j^{\sigma(e,l)} n_k^e \right\rangle_{\partial\Omega_l^e}.$$

Note that we choose the trace expansion to match the expansions used along the edge of the elemental expansion and the local coordinates are aligned, that is $\psi_i^{\sigma(e,l)}(s) = \phi_{k(i)}(s)$ (which is typical of a modified expansion basis defined earlier). With this choice, $\mathbb{E}_l^e$ contains the same entries as $\mathbb{F}_l^e$ and similarly $\widetilde{\mathbb{E}}_{kl}^e$ contains the same entries as $\widetilde{\mathbb{F}}_{kl}^e$.

After inserting the finite expansion of the trial functions into Eqs. (5b) and (5c), and using the definition of the flux given in Eq. (5d), the equations for the local solvers can be written in matrix form as:

$$\mathbb{A}^e \underline{v}^e + \mathbb{C}^e \underline{\hat{\lambda}}^e = \underline{w}^e. \tag{7}$$

where $\underline{f}^e[i] = (\phi_i, f)_{\Omega^e}$, $\underline{w}^e = (\underline{f}^e, 0, 0)^T$ and $\underline{v}^e = (\underline{\hat{u}}^e, \underline{\hat{q}}_1^e, \underline{\hat{q}}_2^e)^T$ is the concatenation of all the unknowns into one vector.

In case of a triangular element, $\underline{\hat{\lambda}}^e = \left(\underline{\hat{\lambda}}^{\sigma(e,1)}, \underline{\hat{\lambda}}^{\sigma(e,2)}, \underline{\hat{\lambda}}^{\sigma(e,3)}\right)^T$ and matrices $\mathbb{A}^e$ and $\mathbb{C}^e$ are defined as follows:

$$\mathbb{A}^e = \begin{pmatrix} \sum_{l=1}^{N_b^e} \tau^{(e,l)} \mathbb{E}_l^e & -\mathbb{D}_1^e & -\mathbb{D}_2^e \\ (\mathbb{D}_1^e)^T & \mathbb{M}^e & 0 \\ (\mathbb{D}_2^e)^T & 0 & \mathbb{M}^e \end{pmatrix}. \tag{8}$$

$$\mathbb{C}^e = \begin{pmatrix} -\tau^{e,1}\mathbb{F}_1^e & -\tau^{e,2}\mathbb{F}_2^e & -\tau^{e,3}\mathbb{F}_3^e \\ -\widetilde{\mathbb{F}}_{11}^e & -\widetilde{\mathbb{F}}_{12}^e & -\widetilde{\mathbb{F}}_{13}^e \\ -\widetilde{\mathbb{F}}_{21}^e & -\widetilde{\mathbb{F}}_{22}^e & -\widetilde{\mathbb{F}}_{23}^e \end{pmatrix} \tag{9}$$

We note that each block matrix $\mathbb{A}^e$ is invertible since every local solver involves the DG discretization of an elemental domain with weakly enforced Dirichlet boundary conditions $\underline{\hat{\lambda}}^e$. Therefore each local elemental problem is well-posed and invertible.

In the following sections, in order to solve local problems (7) (express $\underline{v}^e$ in terms of $\underline{\lambda}^e$), we will require the application of the inverse of $\mathbb{A}^e$. Instead of inverting the full size matrix $\mathbb{A}^e$ we have chosen to form $(\mathbb{A}^e)^{-1}$ in a block-wise fashion, which would involve the inversion of much smaller elemental matrices:

$$(\mathbb{A}^e)^{-1} = \begin{pmatrix} \mathbb{Z}^e & \mathbb{Z}^e\,\mathbb{D}_1^e\,(\mathbb{M}^e)^{-1} & \mathbb{Z}^e\,\mathbb{D}_2^e\,(\mathbb{M}^e)^{-1} \\ -(\mathbb{M}^e)^{-1}(\mathbb{D}_1^e)^T\mathbb{Z}^e & (\mathbb{M}^e)^{-1}[\mathbb{I} - (\mathbb{D}_1^e)^T\mathbb{Z}^e\mathbb{D}_1^e(\mathbb{M}^e)^{-1}] & -(\mathbb{M}^e)^{-1}\,(\mathbb{D}_1^e)^T\,\mathbb{Z}^e\,\mathbb{D}_2^e\,(\mathbb{M}^e)^{-1} \\ -(\mathbb{M}^e)^{-1}(\mathbb{D}_2^e)^T\mathbb{Z}^e & -(\mathbb{M}^e)^{-1}\,(\mathbb{D}_2^e)^T\,\mathbb{Z}^e\,\mathbb{D}_1^e\,(\mathbb{M}^e)^{-1} & (\mathbb{M}^e)^{-1}[\mathbb{I} - (\mathbb{D}_2^e)^T\mathbb{Z}^e\mathbb{D}_2^e(\mathbb{M}^e)^{-1}] \end{pmatrix} \tag{10}$$

where

$$\mathbb{Z}^e = \left(\sum_{l=1}^{N_b^e} \tau^{(e,l)} \mathbb{E}_l^e + \mathbb{D}_1^e (\mathbb{M}^e)^{-1} (\mathbb{D}_1^e)^T + \mathbb{D}_2^e (\mathbb{M}^e)^{-1} (\mathbb{D}_2^e)^T \right)^{-1} \tag{11}$$

and we have explicitly used the fact that $\mathbb{M}^e = (\mathbb{M}^e)^T$ and $\mathbb{Z}^e = (\mathbb{Z}^e)^T$.

3.2 Matrix Form of the Global Equation for λ

Using the matrices from the previous section we can write the transmission condition (6b) in a similar matrix form. First we introduce the matrices:

$$\bar{\mathbb{F}}^{l,e}[i,j] = \left\langle \psi_i^l, \phi_j^e \right\rangle_{\Gamma^l} \quad \tilde{\bar{\mathbb{F}}}_k^{l,e}[i,j] = \left\langle \psi_i^l, \phi_j^e n_k^e \right\rangle_{\Gamma^l} \qquad \bar{\mathbb{G}}^l[i,j] = \left\langle \psi_i^l, \psi_j^l \right\rangle_{\Gamma^l}.$$

After defining $\underline{g_N}^l[i] = \langle g_N, \psi_i^l \rangle_{\Gamma^l \cap \partial\Omega_N}$, the transmission condition (6b) for a single edge can be written as:

$$\mathbb{B}^e \underline{v}^e + \mathbb{G}^e \underline{\hat{\lambda}}^e + \mathbb{B}^f \underline{v}^f + \mathbb{G}^f \underline{\hat{\lambda}}^f = \underline{g}_N^l, \tag{12}$$

where matrices $\mathbb{B}^e$ and $\mathbb{G}^e$ are defined as follows:

$$\mathbb{B}^e = \begin{pmatrix} -\tau^{e,1} (\mathbb{F}_1^e)^T & (\tilde{\mathbb{F}}_{11}^e)^T & (\tilde{\mathbb{F}}_{21}^e)^T \\ -\tau^{e,2} (\mathbb{F}_2^e)^T & (\tilde{\mathbb{F}}_{12}^e)^T & (\tilde{\mathbb{F}}_{22}^e)^T \\ -\tau^{e,3} (\mathbb{F}_3^e)^T & (\tilde{\mathbb{F}}_{13}^e)^T & (\tilde{\mathbb{F}}_{23}^e)^T \end{pmatrix} \tag{13}$$

$$\mathbb{G}^e = \begin{pmatrix} \tau^{e,1} \bar{\mathbb{G}}^{\sigma(e,1)} & 0 & 0 \\ 0 & \tau^{e,2} \bar{\mathbb{G}}^{\sigma(e,2)} & 0 \\ 0 & 0 & \tau^{e,3} \bar{\mathbb{G}}^{\sigma(e,3)} \end{pmatrix}. \tag{14}$$

Here we are assuming that $l = \sigma(e,i) = \sigma(f,j)$, that is, that the elements e and f have the common internal edge Γ^l. While forming matrix $\mathbb{B}^e$ we use the following two identities which relate previously defined matrices:

$$\mathbb{F}_l^e = \left(\bar{\mathbb{F}}^{\sigma(e,l),e} \right)^T \quad \tilde{\mathbb{F}}_{kl}^e = \left(\tilde{\bar{\mathbb{F}}}_k^{\sigma(e,l),e} \right)^T$$

We see that the transmission condition can be constructed from elemental contributions. In the next section, we show how to use our elemental local solvers given by Eqs. (7) and (12) to obtain a matrix equation for λ only.

3.3 Assembling the Transmission Condition from Elemental Contributions

The last component we require to form the global trace system is the elemental trace spreading operator $\mathcal{A}_{HDG}^e$ that will copy the global trace space information into the local (elemental) storage denoted by $\underline{\hat{\lambda}}^e$ in Sects. 3.1 and 3.2. Let $\underline{\Lambda}^l$ denote the vector of degrees of freedom on the edge Γ^l and let $\underline{\Lambda}$ be the concatenation of these vectors for all the edges of the triangulation. The size of $\underline{\Lambda}$ is therefore

$$N_\lambda = \sum_{l \in \Gamma} N_\lambda^l,$$

where N_λ^l is the number degrees of freedom of λ on the interior edge Γ^l.

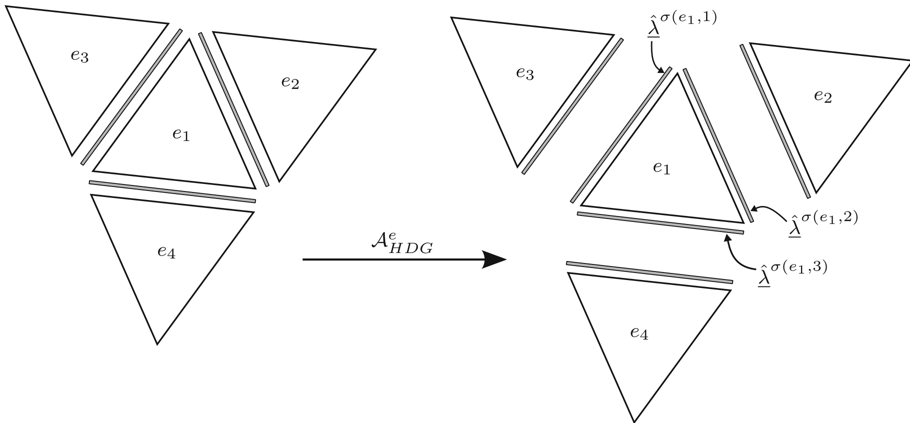


Fig. 1 Diagram showing the results of the spreading operation $\mathcal{A}_{HDG}$. Unique degrees of freedom of λ on an edge are copied to their locally-indexed counterparts

We define the elemental trace space spreading operator $\mathcal{A}^e_{HDG}$ as a matrix of size $(\sum_{l\in\partial\Omega^e} N^l_\lambda) \times N_\lambda$ which "spreads" or scatters the unique trace space values to their local edge vectors. For each element e, which consists of N^e_b edges, let $\hat{\underline{\lambda}}^{e,l}$ denote the local copy of the trace-space information as portrayed in Fig. 1.

With this notation in place we can replace $\hat{\underline{\lambda}}^e$ by $\mathcal{A}^e_{HDG}\underline{\Lambda}$ in local solver Eqs. (7):

$$\mathbb{A}^e\underline{v}^e + \mathbb{C}^e\mathcal{A}^e_{HDG}\underline{\Lambda} = \underline{w}^e \tag{15}$$

We can similarly write the transmission conditions (12) between interfaces as:

$$\sum_{e=1}^{|\mathcal{T}(\Omega)|} (\mathcal{A}^e_{HDG})^T\left[\mathbb{B}^e\underline{v}^e + \mathbb{G}^e\mathcal{A}^e_{HDG}\underline{\Lambda}\right] = \underline{g}_N \tag{16}$$

where the sum over elements along with the left application of the transpose of the spreading operator acts to "assemble" (sum up) the elemental contributions corresponding to each trace space edge and where $\underline{g}_N$ denotes the concatenation of the individual edge Neumann conditions $\underline{g}^l_N$.

Manipulating Eq. (15) to solve for $\underline{v}^e$ and inserting it into Eq. (16) yields:

$$\sum_{e=1}^{|\mathcal{T}(\Omega)|} (\mathcal{A}^e_{HDG})^T\left[\mathbb{B}^e(\mathbb{A}^e)^{-1}\left(\underline{w} - \mathbb{C}^e\mathcal{A}^e_{HDG}\underline{\Lambda}\right) + \mathbb{G}^e\mathcal{A}^e_{HDG}\underline{\Lambda}\right] = \underline{g}_N$$

which can be reorganized to arrive at matrix equation for λ:

$$\mathbf{K}\underline{\Lambda} = \underline{F}, \tag{17}$$

where

$$\mathbf{K} = \sum_{e=1}^{|\mathcal{T}(\Omega)|} (\mathcal{A}^e_{HDG})^T\mathbb{K}^e\mathcal{A}^e_{HDG} = \sum_{e=1}^{|\mathcal{T}(\Omega)|} (\mathcal{A}^e_{HDG})^T\left[\mathbb{G}^e - \mathbb{B}^e(\mathbb{A}^e)^{-1}\mathbb{C}^e\right]\mathcal{A}^e_{HDG}$$

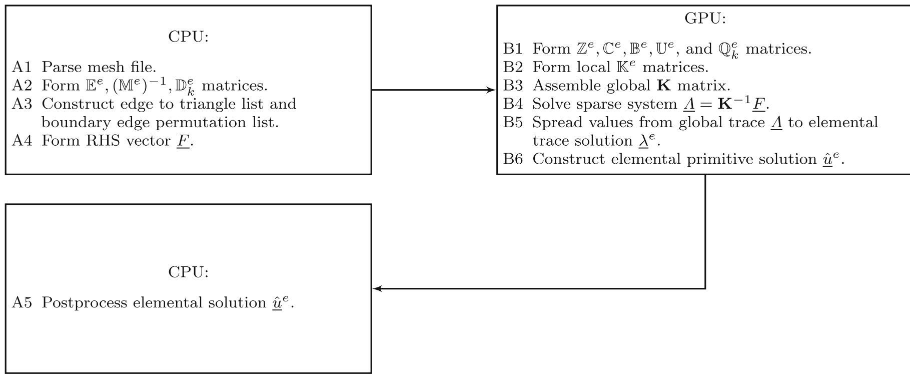


Fig. 2 HDG pipeline

and

$$\underline{F} = \underline{g}_N - \sum_{e=1}^{|\mathcal{T}(\Omega)|} (\mathcal{A}_{HDG}^e)^T \mathbb{B}^e (\mathbb{A}^e)^{-1} \underline{w}. \tag{18}$$

We observe that $\mathbf{K}$ is constructed elementally through the sub-matrices $\mathbb{K}^e$ which can also be considered as the Schur complement of a larger matrix system which consists of combining Eqs. (15) and (16). We would like to remark that the "assembly" in this section is used in the sense of an operator: system matrix $\mathbf{K}$ does not necessarily need to be formed explicitly but can also be stored as a collection of elemental matrices and corresponding mappings.

4 Implementation Pipeline

We formulated our approach as a pipeline which illustrates the division of tasks between CPU (host) and GPU (Fig. 2). Initial setup steps are handled by the CPU after which the majority of the work is performed on the GPU and finally the resulting elemental solution is passed back to the CPU. Initially, the host parses the mesh file to determine the number of elements, forcing function, and mesh configuration. From this information the CPU can generate the data set that is required by the GPU to compute the finite element solution. This is followed by the generation of the $\mathbb{E}^e$, $(\mathbb{M}^e)^{-1}$, $\mathbb{D}_k^e$ elemental matrices, edge to element mappings, global edge permutation lists and the right hand side vector $\underline{F}$. This data is then transferred to the GPU.

The GPU handles the bulk of the operations in our HDG implementation. The first step is the construction of the local elemental matrices through batch processing. The local elemental matrices $\mathbb{Z}^e$, $\mathbb{C}^e$, $\mathbb{B}^e$, $\mathbb{U}^e$, and $\mathbb{Q}_k^e$ are formed from the mass and derivative matrices passed over by the host.

To solve the global trace system we require the assembly of the global matrix $\mathbf{K}$ from the elemental matrices $\mathbb{K}^e$ using the assembly process discussed in Sect. 3.3. We formulate the construction of the elemental $\mathbb{K}^e$ matrices as follows:

$$\mathbb{K}^e = \mathbb{G}^e - \mathbb{B}^e \begin{bmatrix} \mathbb{U}^e \\ \mathbb{Q}_0^e \\ \mathbb{Q}_1^e \end{bmatrix}.$$

where $\mathbb{U}^e$ and $\mathbb{Q}_k^e$ are formulated as:

$$\mathbb{U}^e = -[\mathbb{I} \quad 0 \quad 0](\mathbb{A}^e)^{-1}\mathbb{C}^e = -\mathbb{Z}^e[\mathbb{I} \quad \mathbb{D}_1^e(\mathbb{M}^e)^{-1} \quad \mathbb{D}_2^e(\mathbb{M}^e)^{-1}]\mathbb{C}^e$$
$$\mathbb{Q}_0^e = -[0 \quad \mathbb{I} \quad 0](\mathbb{A}^e)^{-1}\mathbb{C}^e, \qquad \mathbb{Q}_1^e = -[0 \quad 0 \quad \mathbb{I}](\mathbb{A}^e)^{-1}\mathbb{C}^e$$

Note that that the action of $(\mathbb{A}^e)^{-1}$ can be evaluated using definition (10) and so does not need to be directly constructed. The matrices in the first block-row of $(\mathbb{A}^e)^{-1}$ can be reused in the formulation of the second and third block-rows, thereby reducing the computational cost of constructing the matrix.

We next determine the trace space solution $\underline{\Lambda} = \mathbf{K}^{-1}\underline{F}$ where, as was demonstrated in Kirby et al. [26], $\underline{F}$ can be evaluated using $\mathbb{U}^e$ as

$$\underline{F} = \underline{g}_N + \sum_{e=1}^{|\mathcal{T}(\Omega)|} (\mathcal{A}_{HDG}^e)^T (\mathbb{U}^e)^T \underline{f}^e$$

Finally we recover the elemental trace solution $\underline{\lambda}^e = \mathcal{A}_{HDG}^e \underline{\Lambda}$ and obtain the elemental primitive solution $\underline{\hat{u}}^e$ from Eq. (7) as

$$\underline{\hat{u}}^e = \mathbb{Z}^e \underline{f}^e + \mathbb{U}^e \underline{\lambda}^e.$$

4.1 Building the Local Problems on the GPU

The local matrices are created using a batch processing scheme. The generation of the local matrices can be conducted in a matrix-free manner, but we choose to construct the matrices to take advantage of BLAS Level 3 batched matrix functions. We have found this to be a more computationally efficient approach on the GPU. Each step of the local matrix generation process is executed as a batch operating on all elements in the mesh. The batched matrix operations assign a thread block to each elemental matrix. In most cases a thread is assigned to operate on each element of a matrix, which are processed concurrently by the GPU in the various assembly and matrix operations.

Before we proceed to discuss the details of the local matrix generation we would like to make note of a certain implementation detail: the use of the edge to element map. As was previously mentioned in Sect. 3.1, we choose the trace expansion to match the elemental expansion along the element's edge. This choice allows us to use edge expansions together with the edge to element map to generate some of the matrices in a more efficient manner. For example, in Eq. (11) we use the edge to element map to form a sparse matrix $\mathbb{E}_l^e[i,j] = \left\langle \phi_i^e, \phi_j^e \right\rangle_{\partial\Omega_l^e}$ from the entries of a dense matrix $\hat{\mathbb{E}}_l^e[m,n] = \langle \psi_m^e, \psi_n^e \rangle_{\partial\Omega_l^e}$. This approach is also used in the formation of the $\widetilde{\mathbb{E}}_{kl}^e$, $\mathbb{F}_l^e$ and $\widetilde{\mathbb{F}}_{kl}^e$ matrices.

The goal of the local matrix generation process (steps B1 and B2) is to form matrices $\mathbb{K}^e$ for every element in the mesh. In order to facilitate this, the following matrices must be generated: $\mathbb{Z}^e$, block entries of $(\mathbb{A}^e)^{-1}$, $\mathbb{C}^e$, $\mathbb{B}^e$ and $\mathbb{G}^e$. The $\mathbb{Z}^e$ and $\mathbb{U}^e$ matrices will be saved for later computations while the rest of the matrices are discarded after use to reduce memory constraints.

The construction process first requires the $\mathbb{Z}^e$ matrices to be formed from the values of the elemental mass and derivative matrices. The matrices $\mathbb{M}^e$, $\mathbb{D}_k^e$ and $\mathbb{E}_l^e$ are utilized in the formation of the $(\mathbb{Z}^e)^{-1}$ matrices (Eq. 11), which is then inverted in a batch matrix inversion process using Gaussian elimination. Pivoting is not necessary due to the symmetry of the matrices. Next, the block entries of the $(\mathbb{A}^e)^{-1}$ matrices are formed from combinations of the $\mathbb{Z}^e$, $\mathbb{D}_k^e$ and $(\mathbb{M}^e)^{-1}$ matrices (definition 10). The entries from the first block-row of

$(A^e)^{-1}$ are used in the formulation of the second and third block-rows and do not need to be explicitly recomputed. The $\mathbb{U}^e$ and $\mathbb{Q}_k^e$ elemental matrices are created through the multiplication of the block rows of $(\mathbb{A}^e)^{-1}$ and matrix $\mathbb{C}^e$. Note that matrix $\mathbb{B}^e = (\mathbb{C}^e)^T\tilde{\mathbb{I}}$, where

$$\tilde{\mathbb{I}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix},$$

which simplifies the formation process of the $\mathbb{C}^e$ and $\mathbb{B}^e$ matrices.

The final step of the local matrix generation involves constructing the local $\mathbb{K}^e$ matrices which are formed from the explicit matrix-matrix multiplication of the $\mathbb{B}^e$ matrices with the concatenated $\mathbb{U}^e$ and $\mathbb{Q}_k^e$ matrices. This is subtracted from the diagonal $\mathbb{G}^e$ matrix, which is not formed explicitly, to form $\mathbb{K}^e$. Note that matrices $\mathbb{M}^e$, $\mathbb{Z}^e$, and $\mathbb{K}^e$ matrices will be symmetric which halves the required storage space. The elemental operations at each step are independent of each other so the batches can be broken up into smaller tiles to conform to memory constraints or to be distributed across multiple processing units. This process results in the local $\mathbb{K}^e$ matrices being generated for each element which are then used to assemble the global $\mathbf{K}$ matrix.

4.2 Assembling the Local Problems on the GPU

In this section we describe the assembly of the global linear system matrix $\mathbf{K}$ from the elemental matrices $\mathbb{K}^e$. A typical CG or DG element-based approach to the assembly process, when parallelized, has to employ atomic operations to avoid race conditions. In this paper we propose an edge based assembly process that eliminates the need of expensive GPU atomic operations and avoids race conditions by using reduction operations. The reduction list is generated with a sorting operation which is relatively efficient on GPUs. This lock-free approach is better suited for the SIMD architecture of the GPU where each thread is acting on a separate edge in the mesh. In this way we avoid any race conditions during the assembly process while still maximizing throughput on the GPU.

Next, we describe the proposed method for triangular meshes. Note that this approach can be straightforwardly extended to quadrilateral meshes. In order to evaluate a single entry of the global matrix $\mathbf{K}$ we need to determine the indices of entries to which local matrices $\mathbb{K}^e$ will be assembled. To do this, we need to know which element(s) a given edge l_i belongs to. Given the input triangle list that stores the global edge indices of each triangle, we can generate the edge neighbor list that stores the neighboring triangle indices for each edge. Having the edge neighbor list, we assign the assembly task of each row of $\mathbf{K}$ to a thread. Each thread uses the edge neighbor list and the triangle list to find the element index e as well as the entry indices of $\mathbb{K}^e$ to fetch the appropriate data and perform the assembly operation on the corresponding row of $\mathbf{K}$.

To give a better illustration of the assembly process, let us consider a simple mesh displayed in Fig. 3. This mesh consists of two triangles e_0 and e_1 and five edges: l_0 through l_4. To further simplify our example, let us assume that we have only one degree of freedom per edge. $\mathbb{K}^e$ is therefore a 3×3 matrix and $\mathbf{K}$ is a 5×5 matrix. Element e_0 consists of edges $l_0 = \sigma(e_0, 0)$, $l_1 = \sigma(e_0, 1)$ and $l_2 = \sigma(e_0, 2)$ and element e_1 consists of edges $l_0 = \sigma(e_1, 0)$, $l_4 = \sigma(e_1, 1)$ and $l_3 = \sigma(e_1, 2)$.

For our example, the triangle list would be {0,1,2,0,4,3}. Using it we can create an edge neighbor list {0,0,0,1,1,1} that stores the index of a triangle to which each edge from the first list belongs. Next we sort the triangle list by edge index and permute the edge neighbor list

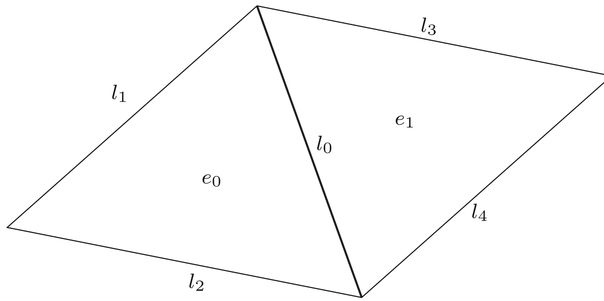


Fig. 3 Mesh with two elements: e_0 and e_1

Fig. 4 Assembly of the 0th row of K

$$K^{e_0} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \qquad K^{e_1} = \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

$$K = \begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} & k_{04} \\ k_{10} & k_{11} & k_{12} & k_{13} & k_{14} \\ k_{20} & k_{21} & k_{22} & k_{23} & k_{24} \\ k_{30} & k_{31} & k_{32} & k_{33} & k_{34} \\ k_{40} & k_{41} & k_{42} & k_{43} & k_{44} \end{bmatrix}$$

according to the sorting. Now the triangle list and edge neighbor list are {0,0,1,2,3,4} and {0,1,0,0,1,1} respectively. These new lists indicate that edge l_0 neighbors triangles e_0 and e_1, and that edge l_1 has neighbors only one triangle e_0, etc. Figure 4 demonstrates the assembly process of the 0th row (corresponding to the l_0 edge) of the $\mathbf{K}$ matrix from the entries of elemental matrices $\mathbb{K}^{e_0}$ and $\mathbb{K}^{e_1}$.

In practice, the global matrix $\mathbf{K}$ is $N_\lambda^l N_\Gamma \times N_\lambda^l N_\Gamma$ and usually sparse. For triangular meshes, each row of $\mathbf{K}$ has at most $5N_\lambda^l$ non-zero values and all the interior edges (edges that do not fall on Dirichlet boundary) have exactly $5N_\lambda^l$ non-zero values. The fact that the number of non-zero entries per row of $\mathbf{K}$ is constant (apart from the rows corresponding to the Dirichlet boundary edges) determines our choice of the Ellpack (ELL) sparse matrix data structure [28] to store $\mathbf{K}$. The ELL data structure contains two arrays. One consists of column indices and the other of matrix values, both of which are of the size $N_\lambda^l N_\Gamma \times 5N_\lambda^l$. The former array stores the column indices of the non-zero values in the matrix, and the latter array stores the non-zero values. For rows that have less than $5N_\lambda^l$ non-zero values, sentinel values (-1 usually) are stored in the column-indices array. Each thread, which is in charge of assembling one row of $\mathbf{K}$, locates its neighboring triangle indices from the edge neighbor list and then obtains the edge indices of these neighboring triangles from the triangle list. The edge indices are then written into the column-indices array of the ELL matrix. Lastly, the local matrix values of the neighboring triangles are assembled into $\mathbf{K}$.

The global assembly process can be summarized as follows:

```
Data: Triangle List TL
\\Generate edge neighbor list EL;
for i ← 0 to NumTriangles−1 do
    EL[i*3] ← i;
    EL[i*3+1] ← i;
    EL[i*3+2] ← i;
end
//Sort triangle list by edge index
TL ← Sort(TL);
//Permute edge neighbor list according to sorted order of triangle list
EL ← Permute(TL);
//Compute the Edge Count List (ECL) through reduction by key, which is the number
  of neighboring triangles on each edge
ECL ← ReduceByKey(TL);
//Calculate a prefix sum on the reduced list (RL) to find the offsets in the sorted triangle
  list
RL ← Scan(ECL);
Local-to-Global Mapping(TL, RL, EL);
```

Algorithm 1: Global Assembly

```
Data: TL, RL, EL
foreach edge e do
    //Locate the neighboring triangles of edge e from the permuted edge neighbor list
      and offset list
    Tris ← Neighbors(EL, RL, e);
    TEdges ← Edges(Tris);
    //Obtain the global indices (GI) of the edges of these triangles from the triangle list
    GI ← TL[Tris*3 + 0,1,2];
    //Store the global indices in the column-indices array (CI)
    CI ← GI;
    //Compute the local indices (LI) of each edge in the neighboring triangles
    LI ← Local Indices(Tris);
    //Locate the entries in the local matrices of the corresponding neighboring triangles
      according to the local indices, and add those entries to the corresponding locations
      in the the global K matrix
    foreach index i in LI do
        K(Map(i)) ← K(Map(i)) + 𝕂^e[i];
    end
end
```

Algorithm 2: Local-to-Global Mapping

the the from the triangles from array. corresponding and add those global **K**

Remark 1 We would like to stress the importance of the edge-only inter-element connectivity provided by the HDG method. This property ensures that the sparsity (number of nonzero entries per row) of the global linear system matrix depends only on the element types used and not on the mesh structure (e.g. vertex degree). The other benefit provided by the HDG method is the ability to assemble the system matrix by-edges as opposed to by-elements,

which removes the need for costly atomic assembly operations. Now, if we look at the CG method, elements are connected through both edge degrees of freedom and vertex degrees of freedom. This through-the-vertex element connectivity makes it both unfeasible to use the compact ELL system matrix representation for a general mesh and makes it hard to avoid atomic operations in the assembly process.

Remark 2 We note that there are multiple ways to address the issue of evaluating the discrete system. A full global system need not be assembled in some cases. One can use a local matrix approach or a global matrix approach. In the local matrix approach, a local operator matrix is applied to each elemental matrix. This allows for on the fly assembly without the need to construct a global matrix system. The global matrix approach assembles a global matrix system from the local elemental contributions. Vos et al. [29] describe these approaches in detail for the continuous Galerkin (FEM) method. In either case, information from multiple elements must be used to compute any given portion of the final system. This requires the use of some synchronized ordering within the mapping process. There are several methods for handling this ordering. One such method is to use atomic operations to ensure that each element in the final system is updated without race conditions. Another method is to use asynchronous ordering and pass the updates to a communication interface which handles the updates in a synchronized fashion. This is demonstrated in the work by Goddekke et al. [30,31], in which they use MPI to handle the many-to-one mapping through asynchronous ordering. In either case a many-to-one mapping exists and a synchronized ordering must be used to prevent race conditions. We chose to use the global approach to compare our results to the previous work by Kirby et al. [26], in which the authors also used the global approach.

4.3 Trace Space Solve and Local Problem Spreading on the GPU

The final steps of the process construct the elemental primitive solution $\hat{u}^e$ (B5 and B6 of the GPU pipeline). This requires retrieving the elemental solution from the trace solution. We form the element-wise vector of local $\underline{\lambda}^e$ coefficients by scattering the coefficients of the global trace solution $\underline{\Lambda}$ produced by the sparse solve. The values are scattered back out to the local vectors using the edge to triangle list. Each interior edge will be scattered to two elements and each boundary edge will be scattered to one element. This is equivalent to the operation performed by the trace space spreading operator $\mathcal{A}^e_{HDG}$ which we conduct in a matrix free manner.

Obtaining the elemental solution involves two batched matrix–vector multiplications across all elements followed by a vector–vector sum:

$$\underline{\hat{u}}^e = \mathbb{Z}^e \underline{f}^e + \mathbb{U}^e \underline{\lambda}^e.$$

After the local element modes are computed they are transferred back to the CPU as a vector grouped by element.

5 Numerical Results

In this section we discuss the performance of the GPU implementation of the HDG method using the Helmholtz equation as a test case. In the end of the section we also provide a short discussion of the CG method GPU implementation based on the preliminary data collected. For verification and runtime comparison we use a CPU implementation of the Helmholtz solver existing within the Nektar++ framework v3.2 [32]. Nektar++ is a freely-available

Table 1 Numerical errors from the GPU implementation of Helmholtz solver on a 40×40 triangular mesh

Order	GPU L^{∞} error	Order of convergence	GPU L^2 error	Order of convergence
1	1.59334e−02	–	3.95318e−03	–
2	4.95546e−04	5.01	8.04917e−05	5.62
3	1.10739e−05	5.48	1.3446e−06	5.90
4	1.93802e−07	5.84	1.88309e−08	6.16
5	5.71909e−09	5.08	1.07007e−09	4.14
6	1.40495e−08	−1.30	4.63559e−09	−2.12
7	2.46212e−08	−0.81	5.77189e−09	−0.32
8	5.19398e−08	−1.08	1.44714e−08	−1.33
9	1.17087e−07	−1.17	2.92382e−08	−1.01

highly-optimized finite element framework. The code is robust and efficient, and it allows for ease of reproducibility of our CPU test results. Our implementation also takes advantage of the GPU parallel primitives in the CUDA Cusp and Thrust libraries [33,34]. All the tests referenced in this section were performed on a machine with a Nvidia Tesla M2090 GPU, 128 GB of memory, and an Intel Xeon E5630 CPU running at 2.53 GHz. The system was using openSUSE 12.1 with CUDA runtime version 4.2.

The numerical simulation considers the Helmholtz equation

$$\begin{aligned} \nabla^2 u(\boldsymbol{x}) - \lambda u(\boldsymbol{x}) &= f(\boldsymbol{x}) \quad \boldsymbol{x} \in \Omega, \\ u(\boldsymbol{x}) &= g_D(\boldsymbol{x}) \quad \boldsymbol{x} \in \partial\Omega_D, \end{aligned}$$

where $\lambda = 1$, $\Omega = [0, 1]^2$ and $f(\boldsymbol{x})$ and $g_D(\boldsymbol{x})$ are selected to give an exact solution of the form:

$$u(x, y) = sin(2\pi x)sin(2\pi y).$$

Tests were performed on a series of regular triangular meshes, produced by taking a uniform quadrilateral mesh and splitting each quadrilateral element diagonally into two triangles. We define the level of mesh refinement by the number of equispaced segments along each side of the domain. The notation $n \times n$ further used in this section corresponds to a mesh comprised of $n \times n$ quads, each split into 2 triangles. We consider meshes of size $20 \times 20 = 800$ elements, $40 \times 40 = 3{,}200$ elements, and $80 \times 80 = 12{,}800$ elements. Although we tested this method on structured meshes, the algorithm does not depend upon the mesh structure and can easily operate over unstructured meshes. In order to help ensure that the sensitivity of the timing routines does not influence the results, we averaged the data over 3 separate runs.

To verify the correctness of our implementations we compared our solution for the Helmholtz equation with the corresponding analytic solution using the L^2 and L^{∞} error norms. The parameter τ for the HDG solver (see Eq. (5d)) was set to 1 for both CPU and GPU implementations. We observe that our implementations produce solutions that match that of the analytic solution to within machine precision. Numerical errors produced by the GPU implementation are presented in Table 1.

Next we consider the total run-time comparison between GPU and CPU implementations of the HDG method. Table 2 presents timing results of both implementations across the entire range of test meshes as well as the relative speedup factors. Columns 2, 5 and 8 indicate the time required by the GPU implementation to complete steps A2–B6 of the HDG pipeline, including time to transfer the data to the GPU but excluding the transfer time of the solution vector back to the CPU. Columns 3, 6 and 9 indicate the time taken by the CPU

Table 2 Total run time data for CPU and GPU implementation of Helmholtz problem (time is measured in ms)

Order	20 × 20 mesh			40 × 40 mesh			80 × 80 mesh		
	GPU	CPU	Speedup	GPU	CPU	Speedup	GPU	CPU	Speedup
1	117	268	2.29	231	1,427	6.19	559	9,889	17.69
2	170	483	2.84	323	2,843	8.8	858	24,459	28.5
3	264	828	3.14	480	5,145	10.71	1,508	54,728	36.28
4	383	1,414	3.69	853	8,896	10.43	2,777	105,896	38.13
5	526	2,268	4.31	1,387	15,165	10.94	4,894	180,373	36.85
6	769	3,484	4.53	2,295	24,873	10.84	8,165	289,319	35.44
7	1,136	5,251	4.62	3,550	36,869	10.39	12,879	436,217	33.87
8	1,613	7,683	4.76	5,393	54,474	10.1	20,072	630,613	31.42
9	2,214	11,451	5.17	7,489	79,604	10.63	28,481	883,340	31.02

implementation to complete the equivalent steps with no induced transfer time. It can be observed that GPU implementation scales well with the increase in mesh size, and the GPU implementation gains performance improvement on the order of 30× over a well optimized serial CPU implementation. Note, that the performance of the GPU implementation can be increased even further by moving additional code for local matrix generation from CPU to GPU (step A2 of the pipeline). The results indicate the method demonstrates strong scaling with respect to mesh size.

In order to provide the reader with a better intuition on the scaling of different stages of the GPU solver with respect to mesh size and polynomial order, we broke the GPU implementation into four stages which were individually timed. The local matrix generation stage corresponds to steps B1 and B2 of the GPU process plus the transfer of required data from CPU to GPU. The transfer time and processing times in this stage are additive, and there is no concurrent processing while transferring data from the host to the GPU. This represents a worst-case scenario for timing results as the performance would only increase with concurrent processing while transferring data. The global assembly stage represents step B3 of the GPU process. The global solve stage is step B4, and the local solve stage corresponds to steps B5 and B6 (not including the time to transfer the solution back to the host). We note that the GPU implementation requires the most allocated memory in the global assembly process, during which the $\mathbb{Z}^e$, $\mathbb{U}^e$, $\mathbb{K}^e$, and $\mathbf{K}$ matrices must be allocated. This point is a memory bottleneck in the system. Table 3 illustrates the memory constraints for each mesh size across the range of polynomial orders. The GPU is generally more memory constrained than the CPU and it will eventually reach a limit based on mesh size and polynomial order. The $\mathbb{Z}^e$ and $\mathbb{U}^e$ matrices could be deallocated and recalculated again in step B6 to lower memory constraints.

Tables 4, 5 and 6 provide the timing results of the individual stages for the 20 × 20, 40 × 40, and 80 × 80 meshes respectively. As can be seen from Tables 4, 5 and 6, for smaller problem sizes (in terms of both polynomial order and element count) global solve is the dominating factor; however, as the problems size increases, the balance shifts in favor of local matrix generation stage. Figure 5 demonstrates the trend in the distribution of total run-time between different stages for a moderately sized problem: run-time taken by the local matrix generation grows quickly as polynomial order increases, reaching approximately 50 % of the total run-time for polynomial order $P = 9$ on an 80 × 80 mesh.

Table 3 GPU memory requirements (in kB) for each mesh and polynomial order

Polynomial order	20 × 20 mesh	40 × 40 mesh	80 × 80 mesh
1	685	2,727	14,869
2	1,887	7,517	41,818
3	3,968	15,821	89,211
4	7,160	28,560	162,624
5	11,693	46,656	267,633
6	17,797	71,031	409,813
7	25,703	102,605	594,740
8	35,640	142,301	827,989
9	47,840	191,040	1,115,136

Table 4 Timing data for the four major stages of GPU implementation on 20 × 20 mesh (time is measured in ms)

Polynomial order	Local matrix generation—HDG	Global assembly	Global solve	Local solve
1	7	18	75	2
2	9	51	106	2
3	11	47	113	2
4	14	56	161	2
5	21	42	162	2
6	40	95	215	2
7	60	113	203	2
8	107	121	253	2
9	155	132	246	3

Table 5 Timing data for the four major stages of GPU implementation on 40 × 40 mesh (time is measured in ms)

Polynomial order	Local matrix generation—HDG	Global assembly	Global solve	Local solve
1	11	29	124	3
2	14	47	128	3
3	19	61	133	4
4	28	59	191	4
5	55	57	195	5
6	94	192	257	6
7	140	139	266	7
8	249	92	346	7
9	422	137	361	8

We use batched matrix-matrix multiplication operations as the baseline comparison for our method. The FLOPS demonstrated by homogeneous BLAS3 operations serve as an upper bound on the the performance of the batched operations carried out in the HDG process. The batched operations in the HDG pipeline are a combination of BLAS1, BLAS2, BLAS3, and matrix inversion operations. BLAS3 operations demonstrate the best performance, in terms

Table 6 Timing data for the four major stages of GPU implementation on 80×80 mesh (time is measured in ms)

Polynomial order	Local matrix generation—HDG	Global assembly	Global solve	Local solve
1	18	53	210	6
2	32	88	213	7
3	44	135	239	8
4	82	159	303	9
5	194	146	355	10
6	347	236	469	10
7	537	291	551	12
8	868	322	722	13
9	1,413	405	769	17

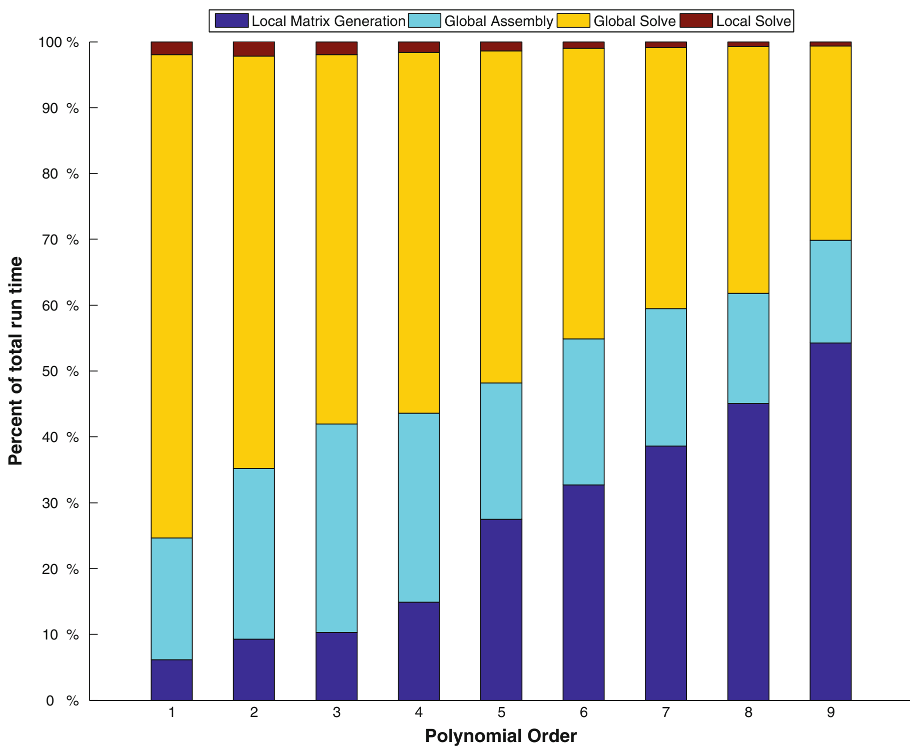


Fig. 5 Ratios of different stages of GPU implementation with respect to the total run time. 80×80 mesh is used

of FLOPS, due to to higher computational density over the other operations. Our method demonstrates peak performance of 60 GFLOPS, which is ∼75 % of the peak FLOPS seen by batched matrix-matrix multiplication operations using cuBLAS [35], on a GPU with 665 peak GFLOPS for double precision. The addition of matrix inversion operations, BLAS1 and BLAS2 operations lower the computational performance from that of pure BLAS3 operations.

Figure 6 illustrates the FLOPS and bandwidth of the local matrix generation process and provides a comparison between the rates on the CPU and GPU (with and without the transfer time). Figure 7 provides an estimate of the FLOPS for the global solve stage. The solver performs the conjugate gradient method on the sparse global matrix. From this we estimated the FLOPS based on the size of **K**, the number of non-zero entries in the global matrix, and the number of iterations required to converge to a solution. Our estimate may be slightly higher than the actual FLOPS demonstrated by the solver, due to implementation specific optimizations. Our FLOPS estimate was derived from the conjugate gradient algorithm which requires approximately $2N_{nz} + 3N_{rows} + N_{iter} * (2N_{nz} + 10N_{rows})$ operations, where N_{nz} is the number of non-zero entries in the sparse global system (which is approximately $N_\lambda^l N_\Gamma \times 5N_\lambda^l$), N_{rows} is the number of rows (which corresponds to $N_\lambda^l N_\Gamma$), and N_{iter} is the number of iterations required to converge to a solution.

The efficiency of the HDG method on the GPU is highlighted by the growth rate of the local matrix generation stage. As polynomial order increases, this step becomes the dominant factor in the run-time. The batch processing technique takes advantage of the independent nature of the local (elemental) operations. The computational density per step increases with mesh size which makes the GPU operations more efficient. At lower mesh sizes the performance is lower due to the increased relative overhead associated and lower computational density.

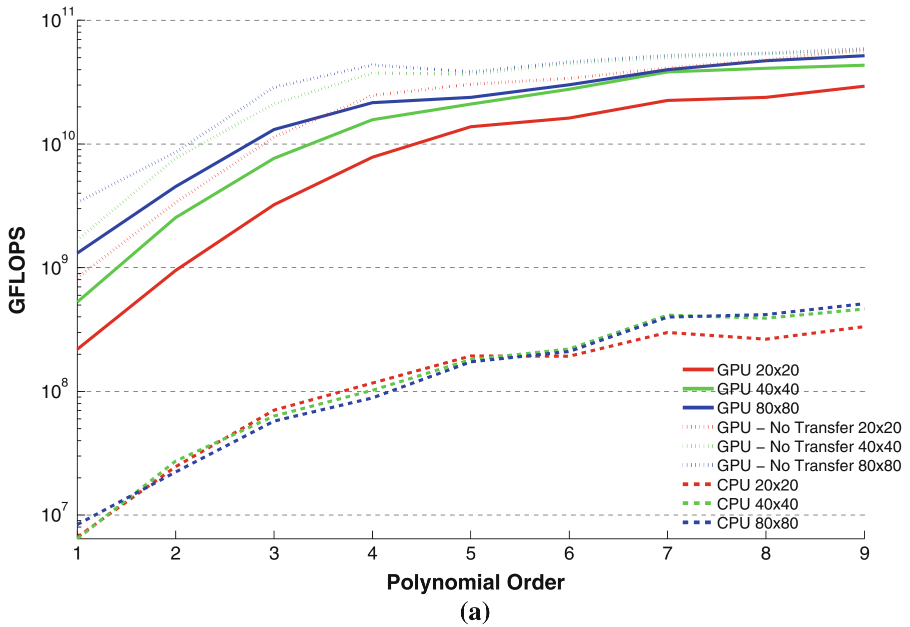
We note that the global solve stage contributes a non-negligible amount of time to the overall method. The choice of iterative solver influences the time taken by this stage. In our CPU implementation we use a banded Cholesky solver, while the GPU implementation uses an iterative conjugate gradient solver from the CUSP library. This CUDA library uses a multigrid preconditioner and is a state-of-the-art GPU solver for sparse linear systems. There are alternatives to this approach, such as the sparse matrix-vector product technique described by Roca et al. [36]. Their method takes advantage of the sparsity pattern of the global matrix to efficiently perform an iterative solve of the system. We chose our approach based on the fact that the global system solve is not the focus of our method, and instead focus on the parallelization of the elemental operations.

We would like to conclude this section with a brief discussion of the efficacy of GPU parallelization when applied to the statically condensed CG method. Static condensation allows the interior modes to be formulated in terms of solutions on the boundary modes through the use of the Schur Complement (see Karniadakis and Sherwin [20] for more details). The statically condensed CG method can therefore be formulated in a similar fashion to the HDG method, which allows it to be implemented within our GPU pipeline. We expect the CG method to take less time during the local matrix generation stage than in the HDG case. This is due to the simpler formulation of the local $\mathbb{K}^e$ matrices, which, as demonstrated in Kirby et al. [26], can be expressed as

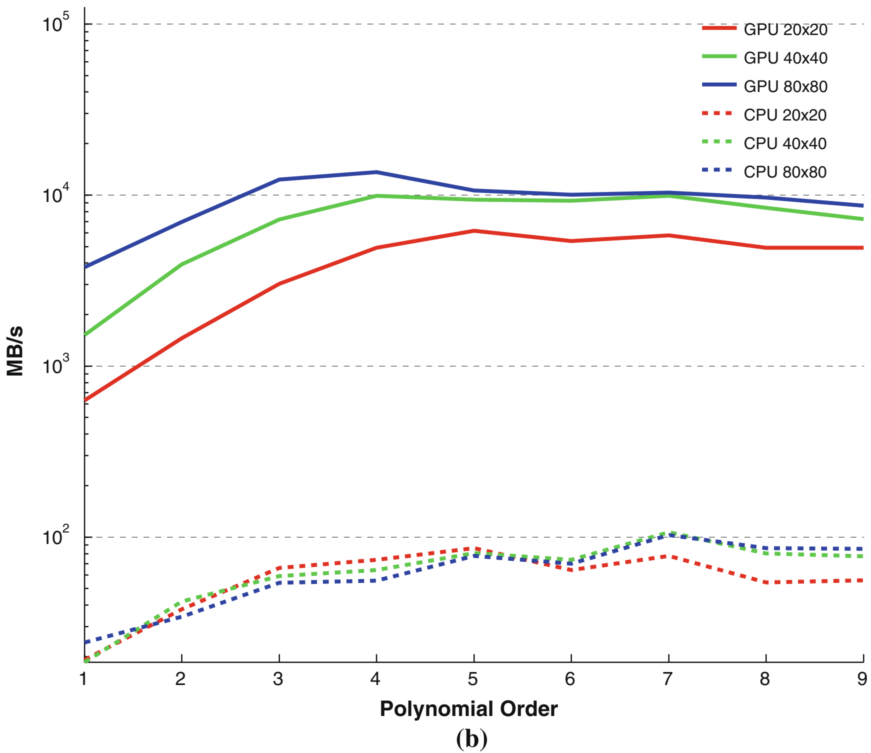
$$\mathbb{K}^e = (\mathbb{D}_1^e)^T (\mathbb{M}^e)^{-1} \mathbb{D}_1^e + (\mathbb{D}_2^e)^T (\mathbb{M}^e)^{-1} \mathbb{D}_2^e - \mathbb{M}^e.$$

This is merely expressing the local elemental matrix in the form of the mass matrix subtracted from the Laplacian matrix which derives from the Helmholtz equation.

To gain further insight into this area, we conducted some preliminary tests. We setup the local (elemental) $\mathbb{K}^e$ matrix generation within our pipeline for the CG case. Table 7 provides the timing results of the local $\mathbb{K}^e$ matrix generation for the HDG and CG methods within our framework across the range of test meshes. For the statically condensed CG method it takes 35–65 % (depending on mesh size and polynomial order) less time to compute the $\mathbb{K}^e$ matrices compared to the HDG method. Our results are only preliminary, as we did not fully implement the statically condensed CG method within our framework. However, our conjecture is that the global assembly step will take longer due to the stronger coupling



(a)



(b)

Fig. 6 GPU local matrix generation metrics. **a** FLOPS of local matrix generation process. **b** Bandwidth of local matrix generation process

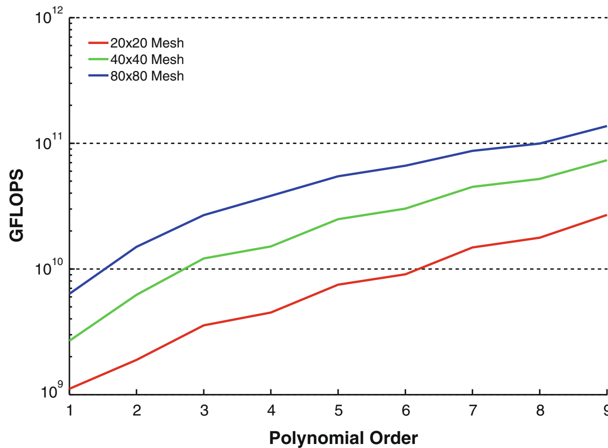


Fig. 7 FLOPS estimate for the global solve stage

Table 7 Local matrix generation time for CG and HDG methods on GPU (time is measured in ms)

Order	20 × 20 mesh		40 × 40 mesh		80 × 80 mesh	
	HDG	CG	HDG	CG	HDG	CG
1	7	4	11	7	18	14
2	9	6	14	12	32	20
3	11	10	19	15	44	32
4	14	12	28	22	82	56
5	21	16	55	36	194	116
6	40	20	94	54	347	209
7	60	30	140	91	537	338
8	107	40	249	131	868	492
9	155	59	422	205	1,413	808

between elements in the CG method. We also suspect that the global solve step may take longer for CG as indicated in Kirby et al. [26], but it may be influenced by differences in architecture (CPU vs. GPU) as well as the choice of solver.

6 Conclusions and Future Work

We have directly compared a CPU and GPU implementation of the HDG method for a two-dimensional elliptic scalar problem using regular triangular meshes with polynomial orders ranging from $1 \leq P \leq 9$. We have discussed how to efficiently implement the HDG method within the context of the GPU architecture, and we provide results which show the relative costs and scaling of the stages that take place in the HDG method as polynomial order and mesh size increase.

Our results indicate the efficacy of applying batched operations to the HDG method. We provide an efficient way to map values from the local matrices to the global matrix during the global assembly step through the use of a lock-free edge mapping technique. This technique avoids atomic operations and is key for implementing an efficient HDG method on the GPU.

The framework we suggest illustrates an effective GPU pipeline which could be adapted to fit methods structurally similar to HDG.

Through our numerical tests we have demonstrated that the HDG method is well suited to large scale streaming SIMD architectures such as the GPU. We consistently see a speed up of $30\times$ or more for meshes of size 80×80 and larger. The method demonstrates strong scaling with respect to mesh size. With each increasing mesh size, for a given polynomial order, the number of elements increases by $4\times$, and we see a corresponding increase in compute time of roughly $\sim 4\times$. As the mesh size increases, the process becomes more efficient due to increased computational density relative to processing overhead. We have also demonstrated that the HDG method is well-suited to batch processing with low inter-element coupling and highly independent operations.

Let us end by indicating possible extensions to the work presented. One possible extension could be a GPU implementation of the statically condensed CG method. The formulation of the statically condensed CG method is similar to that of the HDG method. The structure of the global $\mathbf{K}$ matrix will differ due to increased coupling between elements in the CG case (see Kirby et al. [26] for details). This may present an additional challenge in formulating the global assembly step in an efficient manner on the GPU, because elements are coupled by edges and vertices. We suspect that the performance gains will not be as great as in the HDG case.

Another possible extension could be scaling of the HDG method to multiple GPUs. The local matrix generation and the global assembly step consist of independent operations and would scale well with increased parallelization. The cost of the local matrix generation stage grows at a faster rate than the other stages, and becomes the dominant factor for $P \geq 7$ for moderately sized and larger meshes. The global assembly stage would also see performance gains, since the assembly process is performed on a per-edge basis. Each GPU could be given a unique set of edges to assemble into the global matrix $\mathbf{K}$, with some overlapping edges being passed along to avoid cross communication. The global solve stage may prove to be a bottleneck in a multi-GPU implementation since it cannot be easily divided up amongst multiple processing units. However, as we have shown in our results, the computation time for this step does not grow at the same rate as the local matrix generation step.

Acknowledgments We would like to thank Professor B. Cockburn (U. Minnesota) for the helpful discussions on this topic. This work was supposed by the Department of Energy (DOE NETL DE-EE0004449) and under NSF OCI-1148291.

References

1. Buck, I.: GPU computing: programming a massively parallel processor. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO '07, p. 17. IEEE Computer Society, Washington, DC, USA (2007)
2. Bell, N., Yu, Y., Mucha, P.J.: Particle-based simulation of granular materials. In: Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '05, pp. 77–86. ACM, New York, NY, USA (2005)
3. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE **96**(5), 879–899 (2008)
4. Hesthaven, J.S., Warburton, T.: Nodal Discontinuous Galerkin Methods: Algorithms, Analysis and Applications. Springer, New York (2008)
5. Ali, A., Syed, K.S., Ishaq, M., Hassan, A., Luo, Hong.: A communication-efficient, distributed memory parallel code using discontinuous Galerkin method for compressible flows. In: Emerging Technologies (ICET), 2010 6th International Conference on, pp. 331–336, oct 2010

6. Eskilsson, C., El-Khamra, Y., Rideout, D., Allen, G., Jim Chen Q., Tyagi, M.: A parallel High-Order Discontinuous Galerkin Shallow Water Model. In: Proceedings of the 9th International Conference on Computational Science: Part I, ICCS '09, pp. 63–72. Springer-Verlag, Berlin, Heidelberg (2009)
7. Goedel, N., Schomann, S., Warburton, T., Clemens, M.: GPU accelerated Adams-Bashforth multirate discontinuous Galerkin FEM simulation of high-frequency electromagnetic fields. IEEE Trans. Magn. **46**(8), 2735–2738 (2010)
8. Goedel, N., Warburton, T., Clemens, M.: GPU accelerated Discontinuous Galerkin FEM for electromagnetic radio frequency problems. In: Antennas and Propagation Society International Symposium, 2009. APSURSI '09. IEEE, pp. 1–4, June 2009
9. Klöckner, A., Warburton, T., Hesthaven, J.S.: High-Order Discontinuous Galerkin Methods by GPU Metaprogramming. In: GPU Solutions to Multi-scale Problems in Science and Engineering, pp. 353–374. Springer (2013)
10. Cockburn, B., Karniadakis, G.E., Shu, C.-W. (eds.): The Development of Discontinuous Galerkin Methods. In: Discontinuous Galerkin Methods: Theory, Computation and Applications, pp. 135–146. Springer-Verlag, Berlin (2000)
11. Cockburn, B., Gopalakrishnan, J., Lazarov, R.: Unified hybridization of discontinuous Galerkin mixed and continuous Galerkin methods for second order elliptic problems. SIAM J. Numer. Anal. **47**, 1319–1365 (2009)
12. Klöckner, A., Warburton, T., Bridge, J., Hesthaven, J.S.: Nodal discontinuous Galerkin methods on graphics processors. J. Comput. Phys **228**, 7863–7882 (2009)
13. Lanteri, S., Perrussel, R.: An implicit hybridized discontinuous Galerkin method for time-domain Maxwell's equations. Rapport de recherche RR-7578, INRIA, March (2011)
14. NVIDIA Corporation. CUDA Programming Guide 4.2, April 2012
15. AMD Corporation. AMD Accelerated Parallel Processing Math Libraries, Jan 2011
16. ATI. AMD Accelerated Parallel Processing OpenGL Programming Guide, Jan 2011
17. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, pp. 31:1–31:11. IEEE Press, Piscataway, NJ, USA (2008)
18. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., Tomov, S.: A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In: GPU Computing Gems, Jade Edition **2**, 473–484 (2011)
19. Song, F., Tomov, S., Dongarra, J.: Efficient Support for Matrix Computations on Heterogeneous Multi-core and Multi-GPU Architectures. University of Tennessee, Computer Science Technical, Report UT-CS-11-668 (2011)
20. Karniadakis, G.E., Sherwin, S.J.: Spectral/HP Element Methods for CFD, 2nd edn. Oxford University Press, UK (2005)
21. Sherwin, S.J., Karniadakis, G.E.: A triangular spectral element method. Applications to the incompressible Navier–Stokes equations. Comput. Methods Appl. Mech. Eng. **123**, 189–229 (1995)
22. Cockburn, B., Dong, B., Guzmán, J.: A superconvergent LDG-Hybridizable Galerkin method for second-order elliptic problems. Math. Comput. **77**(264), 1887–1916 (2007)
23. Cockburn, B., Gopalakrishnan, J., Sayas, F.-J.: A projection-based error analysis of HDG methods. Math. Comput. **79**, 1351–1367 (2010)
24. Cockburn, B., Guzmán, J., Wang, H.: Superconvergent discontinuous Galerkin methods for second-order elliptic problems. Math. Comput. **78**, 1–24 (2009)
25. Arnold, D.N., Brezzi, F., Cockburn, B., Marini, D.: Unified analysis of discontinuous Galerkin methods for elliptic problems. SIAM J. Numer. Anal. **39**, 1749–1779 (2002)
26. Kirby, Robert M., Sherwin, Spencer J., Cockburn, Bernardo: To CG or to HDG: a comparative study. J. Sci. Comput. **51**(1), 183–212 (Apr 2012)
27. Dubiner, M.: Spectral methods on triangles and other domains. J. Sci. Comput. **6**, 345–390 (1991)
28. Bell, N., Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec 2008
29. Vos P.E.J.: From h to p efficiently : optimising the implementation of spectral / hp element methods. PhD thesis, University of London, 2011
30. Göddeke, Dominik, Strzodka, Robert, Mohd-Yusof, Jamaludin, McCormick, Patrick S., Wobker, Hilmar, Becker, Christian, Turek, Stefan: Using GPUs to improve multigrid solver performance on a cluster. Int. J. Comput. Sci. Eng. **4**(1), 36–55 (2008)
31. Göddeke, Dominik, Wobker, Hilmar, Strzodka, Robert, Mohd-Yusof, Jamaludin, McCormick, Patrick S., Turek, Stefan: Co-processor acceleration of an unmodified parallel solid mechanics code with FEASTGPU. Int. J. Comput. Sci. Eng. **4**(4), 254–269 (2009)
32. Kirby, R.M., Sherwin, S.J.: Nektar++ finite element library. http://www.nektar.info/

33. Bell, N., Garland, M.: Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations, 2012. Version 0.3.0
34. Hoberock, J., Bell, N.: Thrust: A Parallel Template Library, 2010. Version 1.7.0
35. Ha, L.K., King, J., Fu, Z., Kirby, R.M.: A High-Performance Multi-Element Processing Framework on GPUs. SCI Technical Report UUSCI-2013-005, SCI Institute, University of Utah (2013)
36. Roca, X., Nguyen N.C., Peraire, J.: GPU-accelerated sparse matrix-vector product for a hybridizable discontinuous Galerkin method. Aerospace Sciences Meetings. American Institute of Aeronautics and Astronautics, Jan 2011. doi:10.2514/6.2011-687