Commun. Comput. Phys.
doi: 10.4208/cicp.OA-2018-0235

Vol. **27**, No. 2, pp. 513-545
February 2020

A GPU-Accelerated Hybridizable Discontinuous Galerkin Method for Linear Elasticity

Maurice S. Fabien*

Department of Computational and Applied Mathematics, Rice University, 6100 Main MS-134, Houston, TX 77005, USA.

Received 13 September 2018; Accepted (in revised version) 6 January 2019

Abstract. We design and analyze an efficient GPU-accelerated hybridizable discontinuous Galerkin method for linear elasticity. Performance analysis of the method is done using the state-of-the-art Time-Accuracy-Size (TAS) spectrum. TAS is a new performance measure which takes into account the accuracy of the solution. Standard performance measures, like floating point operations or run-time, are not completely appropriate for gauging the performance of approximations of continuum mechanics problems, as they neglect the solutions accuracy. A standard roofline model demonstrates that our method is utilizing computational resources efficiently, and as such, significant speed ups over a serial implementation are obtained. By combining traditional performance measures and the novel time-accuracy measures [7] into our performance model, we are able to draw more complete conclusions about which discretizations are best suited for an application. Several numerical experiments validate and verify our numerical scheme.

AMS subject classifications: 65Y05, 74S05, 65N30, 65N55

Key words: GPU-acceleration, discontinuous Galerkin, hybridization, multigrid, performance analysis.

1 Introduction

Computer architecture is becoming more sophisticated at the node level, where individual core clock rates are reduced, but more cores are packed into a chipset; to the point that floating point performance is greatly eclipsing memory operations. Efficiently utilizing this type of computational hardware has already been shown to require different programming models and parallel computing paradigms. This trend has consequences for the planning and designing of next-generation high-performance computing software [19, 45]. With this in mind, we design and analyze a GPU-accelerated hybridizable discontinuous Galerkin (HDG) method for linear elasticity.

*Corresponding author. *Email address:* fabien@rice.edu (M. S. Fabien)

HDG methods have several attractive properties, especially for problems where the solution of linear systems is required. As the HDG method is a discontinuous finite element method, it retains many of the features of discontinuous Galerkin (DG) methods that are celebrated, like local conservation, arbitrary order approximations, rigorous mathematical foundation, *hp*-adaptivity, admits unstructured meshes, and so on [12]. One of the important properties of the HDG method that differentiates it from classical DG methods is its ability to use static condensation. This well-known technique introduces additional unknowns on the mesh skeleton, and due to a judicious choice of numerical trace and flux, the original unknowns can be eliminated in an element-by-element fashion [13]. The total number of unknowns for the HDG method is thus reduced by a large amount for higher orders when compared to standard DG methods. As a result, drastic savings for memory storage and computational time are feasible [33].

For the linear elasticity equations, several HDG methods have been proposed, e.g., [15, 17, 17, 25, 53, 57, 58]. These discretizations have variations in how the primal and mixed variables are treated. We follow the discretization outlined in [43], as it is locking-free (for any $k \geq 0$), and is easily extendible to other equations, like Stokes and nonlinear elasticity. Related discretizations, like face centred finite volumes [56], the weak Galerkin method [9, 62], and the hybrid high order method [14, 20] have also been studied. Relevant applications can be found in [16, 34, 35].

The HDG method for linear elasticity presented in [43] has all approximate variables (displacement, gradient of displacement, and hydrostatic pressure) converge at the optimal rate of $k+1$ in the L^2-norm for polynomials of degree $k \geq 0$. There are a few interesting consequences of the gradient of displacement converging at the optimal rate. Quantities of engineering interest, like vorticity, stress, and strain also all converge at the rate of $k+1$ in the L^2-norm [43]. In addition, there exist local postprocessing schemes for the displacement variable which result in a new displacement approximation that superconverges at the rate of $k+2$ in the L^2-norm [44]. As the postprocessing is performed in an element-by-element manner, it is much cheaper than solving the full system at one polynomial order higher.

GPU-accelerated numerical methods for partial differential equations (PDEs) have received a great deal of attention. In particular, lattice Boltzmann and discontinuous Galerkin finite element methods have been demonstrated to perform well for linear wave problems and hyperbolic conservation laws [10, 66, 67]. However, few many-core or even GPU-accelerated DG methods are considered for partial differential equations that are inherently implicit in their nature, like PDEs that have elliptic or parabolic characteristics. The dominant difficulty for this class of PDEs is that they require the solution of large sparse linear systems. Efficient sparse linear solvers for high order DG methods are in general bandwidth bound, and more sophisticated [24, 29]. Preconditioners that map well to many-core architectures (e.g. [1, 3]) may not provide the best performance due to their poor convergence rates [22].

A GPU-accelerated HDG method for the 2D Poisson problem is proposed in [37], which analyzes in detail an efficient HDG assembly based on batch processing. The lin-

ear solver they used is an algebraic multigrid preconditioned conjugate gradient method [4, 5]. The linear solver is not the focus of their paper. On the other hand, they show that for high orders the global solve (ignoring assembly costs) is the bottleneck in their single-GPU study. In [54] a GPU-accelerated specialized block sparse matrix format is introduced, which comes with a fast tensor contraction for the HDG method, and this allows for an efficient sparse matrix-vector product. No linear solver is considered. In [22] a HDG multigrid linear solver is consider for the 2D Poisson problem in a many-core environment (Xeon Phi).

Performance modeling is crucial for determining the efficiently and scalability of computational frameworks [36]. Moreover, it allows for systematic improvements through optimizations and tuning. Standard performance measures, like floating point operations or run-time, are not completely appropriate for gauging the parallel performance of approximations of continuum mechanics problems, as they neglect solution accuracy. In other words, optimal device utilization does not always translate to meaningful and accurate computational science. By incorporating novel time-accuracy measures [7] into our performance model, we are able to draw more complete conclusions about which discretizations are best suited for an application.

The main contributions of this paper is now outlined. Section 2 states the model problem and the HDG discretization. In Section 3 we detail our strategy of implementing the HDG method on the GPU. We accelerate the static condensation process for the HDG method, extending the work in [37] to vector-valued problems. Since our problem is in 3D, and is a vector-valued system, there are some novel enhancements we must make in order to extend the size of problems we can consider. Most notably, the local solver matrices on an element are much larger than for scalar problems. A straightforward implementation severely restricts the size of problem one can consider, due to the limited GPU memory. To address this, we take advantage of the rich sparsity structure that the local solver matrices have. The local solvers are completely data parallel, and element local, so they benefit from parallelism on the GPU. We leverage state-of-the-art optimized GPU-accelerated CUDA libraries [46, 47] to achieve significant speed ups over a serial CPU implementation.

This is covered in Section 3, which also has a traditional parallel performance analysis inspecting device utilization, and roofline model [64]. Exposing fine grain parallelism for the global solver is more challenging. To do this, we detail and assess the algorithmic and computational performance of a GPU-accelerated multigrid preconditioner. This preconditioner is an extension of the work done in [22], applied to the 3D linear elasticity equations, which is the focus of Section 3.3.

In Section 4 several numerical examples are considered to verify and validate our GPU-accelerated linear elasticity solver. Section 5 revisits the computational analysis, where time-accuracy measures [7, 8] are applied to better gauge the performance of the GPU-accelerated HDG approximations. Conclusions follow.

2 Model problem

The small deformation of the elastic isotropic body $\Omega \subset \mathbb{R}^3$ is governed by the following equation

$$-\nabla\cdot(\mu\nabla\boldsymbol{u}+(\mu+\lambda)(\nabla\cdot\boldsymbol{u})\boldsymbol{I})=\boldsymbol{f}, \tag{2.1}$$

where $\boldsymbol{u}$ represents the displacement components, λ is the Lamé parameter, the shear modulus is given by μ, and the body force is denoted by $\boldsymbol{f}$. We introduce the displacement gradient tensor, $\boldsymbol{H}=\nabla\boldsymbol{u}$, and the hydrostatic pressure $p=-(\mu+\lambda)(\nabla\cdot\boldsymbol{u})$. Eq. (2.1) written in first order form is then

$$\boldsymbol{H}-\nabla\boldsymbol{u}=0 \tag{2.2}$$
$$-\nabla\cdot(\mu\boldsymbol{H}-p\boldsymbol{I})=\boldsymbol{f}, \tag{2.3}$$
$$\epsilon p+\nabla\cdot\boldsymbol{u}=0, \tag{2.4}$$

where $\epsilon=(\mu+\lambda)^{-1}$. We work with Eqs. (2.2)-(2.4), as they are extendible to other models in continuum mechanics, e.g., Stokes, acoustic and elastic waves, and nonlinear elasticity. The boundary of Ω, denoted $\partial\Omega$ is decomposed as $\partial\Omega=\partial\Omega_D\cup\partial\Omega_N$, where $\partial\Omega_D\cap\partial\Omega_N=\emptyset$. On the boundaries the following conditions are prescribed:

$$\boldsymbol{u}=\boldsymbol{g}_D, \quad \text{on } \partial\Omega_D, \tag{2.5}$$
$$\boldsymbol{B}\boldsymbol{n}=\boldsymbol{g}_N, \quad \text{on } \partial\Omega_N, \tag{2.6}$$

where $\boldsymbol{n}$ is the outward normal vector to $\partial\Omega_N$, $\boldsymbol{g}_D$ is prescribed Dirichlet data, $\boldsymbol{g}_N$ is prescribed Nuemann data, and $\boldsymbol{B}$ is a linear boundary operator. The natural linear operator $\boldsymbol{B}$ is a gradient-pressure condition: $\boldsymbol{B}=-\mu\boldsymbol{H}+p\boldsymbol{I}$. Other conditions can be considered, for example, for a stress type condition one has: $\boldsymbol{B}=-\mu(\boldsymbol{H}+\boldsymbol{H}^T)+\epsilon\lambda p\boldsymbol{I}$.

2.1 Hybridizable discontinuous Galerkin discretization

The elastic body Ω is partitioned into a collection of tetrahedra, given by $\mathcal{T}_h$. Each tetrahedron is denoted by K, with a maximum diameter h. The mesh skeleton is denoted by Γ_h, which is the union of all the faces in $\mathcal{T}_h$. Put $\partial\mathcal{T}_h$ as the collection of all element boundaries, which contains duplicate faces; whereas Γ_h consists of only unique faces. We assume that $\mathcal{T}_h$ is shape regular [11], that is, there exists a scalar $\gamma>0$ for each $K\in\mathcal{T}_h$ so that

$$\gamma h_K\leq r_K,$$

where h_K is the diameter of K, and r_K is the largest ball inscribed in K.

We define $\mathcal{P}_k(K)$ to be the space of all polynomials of at most degree $k\geq 0$ on a domain K, and $L^2(K)$ to be the space of all square integrable functions on K. Given an integer $k\geq 0$,

the underlying approximation spaces for the HDG method are as follows:

$$\begin{aligned}
\boldsymbol{G}_h &= \{\boldsymbol{G} \in (L^2(\mathcal{T}_h))^{3\times 3} : \boldsymbol{G}|_K \in (\mathcal{P}_k(K))^{3\times 3},\ \forall K \in \mathcal{T}_h\},\\
\boldsymbol{V}_h &= \{\boldsymbol{v} \in (L^2(\mathcal{T}_h))^{3} : \boldsymbol{v}|_K \in (\mathcal{P}_k(K))^{3},\ \forall K \in \mathcal{T}_h\},\\
P_h &= \{p \in L^2(\mathcal{T}_h) : p|_K \in \mathcal{P}_k(K),\ \forall K \in \mathcal{T}_h\},\\
\boldsymbol{M}_h^k &= \{\boldsymbol{\mu} \in (L^2(\Gamma_h))^{3} : \boldsymbol{\mu}|_e \in \mathcal{P}_k(e),\ \forall e \in \Gamma_h\}.
\end{aligned}$$

These are the discontinuous finite element approximation spaces for the gradient of displacement ($\boldsymbol{G}_h$), displacement ($\boldsymbol{V}_h$), hydrostatic pressure (P_h), and trace of displacement ($\boldsymbol{M}_h^k$). The objects of integration over these spaces can be scalars, vectors, or tensors. In order to distinguish between these cases we use the following notation:

$$(p,q)_{\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}\int_K pq, \quad (\boldsymbol{u},\boldsymbol{v})_{\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}\int_K \boldsymbol{u}\cdot\boldsymbol{v}, \quad (\boldsymbol{E},\boldsymbol{H})_{\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}\int_K \mathrm{tr}(\boldsymbol{E}^T\boldsymbol{H}),$$

for $p,q\in\mathcal{P}_h$, $\boldsymbol{u},\boldsymbol{v}\in\boldsymbol{V}_h$ and $\boldsymbol{E},\boldsymbol{H}\in\boldsymbol{G}_h$. Similarly, for the integration over element boundaries, we set

$$\langle p,q\rangle_{\partial\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}\int_{\partial K} pq, \quad \langle \boldsymbol{u},\boldsymbol{v}\rangle_{\partial\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}\int_{\partial K} \boldsymbol{u}\cdot\boldsymbol{v}, \quad \langle \boldsymbol{\eta},\boldsymbol{\mu}\rangle_{\partial\mathcal{T}_h} = \sum_{K\in\mathcal{T}_h}\int_{\partial K} \boldsymbol{\eta}\cdot\boldsymbol{\mu},$$

for $p,q\in\mathcal{P}_h$, $\boldsymbol{u},\boldsymbol{v}\in\boldsymbol{V}_h$ and $\boldsymbol{\eta},\boldsymbol{\mu}\in\boldsymbol{M}_h^k$.

The HDG discretization for Eq. (2.2) seeks approximations $(\boldsymbol{H}_h,\boldsymbol{u}_h,p_h,\widehat{\boldsymbol{u}}_h)\in(\boldsymbol{G}_h,\boldsymbol{V}_h,P_h,\boldsymbol{M}_h^k)$ of $(\boldsymbol{H},\boldsymbol{u},p,\widehat{\boldsymbol{u}})$ such that

$$(\boldsymbol{H}_h,\boldsymbol{E})_{\mathcal{T}_h} + (\boldsymbol{u}_h,\nabla\cdot\boldsymbol{E})_{\mathcal{T}_h} - \langle\widehat{\boldsymbol{u}}_h,\boldsymbol{E}\boldsymbol{n}\rangle_{\partial\mathcal{T}_h} = 0 \tag{2.7}$$

$$(\mu\boldsymbol{H}_h - p_h\boldsymbol{I},\nabla\boldsymbol{w})_{\mathcal{T}_h} + \langle\widehat{\boldsymbol{h}}_h,\boldsymbol{w}\rangle_{\partial\mathcal{T}_h} = (\boldsymbol{f},\boldsymbol{w})_{\mathcal{T}_h} \tag{2.8}$$

$$(\epsilon p_h,q)_{\mathcal{T}_h} - (\boldsymbol{u}_h,\nabla q)_{\mathcal{T}_h} + \langle\widehat{\boldsymbol{u}}_h\cdot\boldsymbol{n},q\rangle_{\partial\mathcal{T}_h} = 0 \tag{2.9}$$

$$\langle\widehat{\boldsymbol{h}}_h,\boldsymbol{\mu}\rangle_{\partial\mathcal{T}_h\setminus\partial\Omega} + \langle\widehat{\boldsymbol{u}}_h - \boldsymbol{g}_D,\boldsymbol{\mu}\rangle_{\partial\Omega_D} + \langle\widehat{\boldsymbol{b}}_h - \boldsymbol{g}_N,\boldsymbol{\mu}\rangle_{\partial\Omega_N} = 0, \tag{2.10}$$

for all $(\boldsymbol{E},\boldsymbol{w},q,\boldsymbol{\mu})\in(\boldsymbol{G}_h,\boldsymbol{V}_h,P_h,\boldsymbol{M}_h^k)$, and the numerical flux and boundary flux are given by

$$\widehat{\boldsymbol{h}}_h = (-\mu\boldsymbol{H}_h + p_h\boldsymbol{I})\cdot\boldsymbol{n} + \boldsymbol{S}(\boldsymbol{u}_h - \widehat{\boldsymbol{u}}_h), \quad \widehat{\boldsymbol{b}}_h = (-\mu\boldsymbol{H}_h + \epsilon\lambda p_h\boldsymbol{I})\cdot\boldsymbol{n} + \boldsymbol{S}(\boldsymbol{u}_h - \widehat{\boldsymbol{u}}_h). \tag{2.11}$$

Eq. (2.10) enforces continuity of the normal component of the numerical flux. We note that for $\epsilon = 0$, Eqs. (2.7)-(2.10) correspond to a HDG method for Stokes flow (see [44]).

The HDG method boasts convergence rates of $k+1$ for all approximate unknowns in the L^2-norm [43]. Thus, the displacement converges at the rate of $k+1$ in the energy norm. A consequence of the gradient of displacement converging at the rate of $k+1$ is that a postprocessing exists which allows us to generate a superconvergent displacement that converges at the rate of $k+2$ in the L^2 norm [43]. Another result of the optimal gradient of displacement convergence, is other physical quantities of engineering interest also converge at the rate of $k+1$, for instance: stress, strain, and vorticity [43].

2.2 Basis information

On the approximation spaces we utilize the Proriol–Koornwinder–Dubiner–Owens (PKDO) modal basis; a hierarchical orthogonal basis (see [21, 39, 50, 51]). We select this basis for a number of reasons:

- Orthogonality simplifies mass matrices, as they become diagonal.
- Excellent conditioning for interpolation and quadrature which facilitates high order approximations.
- The hierarchical property of the basis renders the prolongation and restriction operators for spectral multigrid binary. Hierarchy can also be exploited if we want to seamlessly switch approximation spaces.

To simplify computation of the HDG method, we form a mapping between the so-called reference elements onto the elements of the mesh. This allows us to do computations on the reference element, instead of defining different basis functions for each physical element [40]. The reference triangle is defined to be the equilateral triangle with points $(-1,1)$, $(-1,-1)$, and $(1,-1)$. The reference tetrahedra is defined by the points $(-1,-1,-1)$, $(1,-1,-1)$, $(-1,1,-1)$, and $(-1,-1,1)$.

In 2D, on a reference triangle, the PKDO basis takes the form

$$\begin{aligned}&\Psi_{m_1}(r,s)=\sqrt{2}P_i(\eta_1)^{(0,0)}P_j(\eta_2)^{(2i+1,0)}(1-\eta_2)^i,\quad \eta_1=\frac{2(r+1)}{(1-s)},\quad \eta_2=s,\\&0\le i,j,\quad i+j\le k,\quad m_1=j+(k+1)i+1-\frac{i(i-1)}{2},\end{aligned}\tag{2.12}$$

with $P_i^{(\alpha,\beta)}(x)$ as the ith order Jacobi polynomial. On the reference tetrahedron the PKDO basis is given by

$$\begin{aligned}&\Phi_{m_2}(r,s,t)=\sqrt{8}P_i(\eta_1)^{(0,0)}P_j(\eta_2)^{(2i+1,0)}P_l(\eta_3)^{(2i+2j+2,0)}(1-\eta_3)^{i+j},\\&\eta_1=-\frac{2(r+1)}{(s+t)}-1,\quad \eta_2=\frac{2(1+s)}{(1-t)}-1,\quad \eta_3=t,\\&0\le i,j,l,\quad i+j+l\le k,\quad m_2=-ij-\frac{(2+k)i^2}{2}-\frac{j^2}{2}+\frac{i^3}{6}.\end{aligned}\tag{2.13}$$

The following notation is used throughout this manuscript. Denote the degrees of freedom (DoFs) on a face by $d_2=\binom{k+2}{2}$, and the degrees of freedom on a tetrahedron by $d_3=\binom{k+3}{3}$. High order polynomials need quadrature rules of sufficient precision. We use the quadrature weights and abscissae on the reference triangle/tetrahedron suggested by the following references: [26, 61].

2.3 Superconvergent postprocessing

The postprocessing scheme we use is described in [43]. Given $K\in\mathcal{T}_h$, an approximate displacement $\boldsymbol{u}_h$, and displacement gradient $\boldsymbol{H}_h$, the postprocessed displacement $\boldsymbol{u}^*$ satisfies

$$(\nabla\boldsymbol{u}^*,\nabla\boldsymbol{w})_K=(\boldsymbol{H}_h,\nabla\boldsymbol{w})_K,\quad \forall\boldsymbol{w}\in(\mathcal{P}^{k+1}(K))^3, \tag{2.14}$$

$$(\boldsymbol{u}^*,1)_K=(\boldsymbol{u}_h,1)_K, \tag{2.15}$$

where $\boldsymbol{u}^*|_K\in(\mathcal{P}^{k+1}(K))^3$. Numerical results (see Section 4, Tables 1 and 3) show that this postprocessing superconverges at the rate of $k+2$ in the L^2–norm for $k>0$. In the case of nearly incompressible materials, the postprocessing still superconverges. This postprocessing is attractive as it can be computed in a completely data parallel manner, and is less expensive than solving the globally coupled system.

2.4 HDG discretization

Here we outline the static condensation feature of the HDG method. In other words, we explain how the interior degrees of freedom corresponding to the gradient of displacement, displacement, and hydrostatic pressure can be eliminated locally; leaving a globally coupled problem defined on the mesh skeleton in terms of the hybrid unknown only. Eqs. (2.7)-(2.10) can be rewritten in matrix form:

$$\begin{bmatrix}\boldsymbol{A}_{HH} & \boldsymbol{A}_{Hu} & \boldsymbol{A}_{Hp} & \boldsymbol{A}_{H\widehat{u}}\\ \boldsymbol{A}_{uH} & \boldsymbol{A}_{uu} & \boldsymbol{A}_{up} & \boldsymbol{A}_{u\widehat{u}}\\ \boldsymbol{A}_{pH} & \boldsymbol{A}_{pu} & \boldsymbol{A}_{pp} & \boldsymbol{A}_{p\widehat{u}}\\ \boldsymbol{A}_{\widehat{u}H} & \boldsymbol{A}_{\widehat{u}u} & \boldsymbol{A}_{\widehat{u}p} & \boldsymbol{A}_{\widehat{u}\widehat{u}}\end{bmatrix}\begin{bmatrix}\boldsymbol{H}\\ \boldsymbol{U}\\ \boldsymbol{P}\\ \widehat{\boldsymbol{U}}\end{bmatrix}=\begin{bmatrix}\boldsymbol{F}_H\\ \boldsymbol{F}_u\\ \boldsymbol{F}_p\\ \boldsymbol{F}_{\widehat{u}}\end{bmatrix}. \tag{2.16}$$

The system in Eq. (2.16) is valid at the element level. We define

$$\mathcal{A}_1=\begin{bmatrix}\boldsymbol{A}_{HH} & \boldsymbol{A}_{Hu} & \boldsymbol{A}_{Hp}\\ \boldsymbol{A}_{uH} & \boldsymbol{A}_{uu} & \boldsymbol{A}_{up}\\ \boldsymbol{A}_{pH} & \boldsymbol{A}_{pu} & \boldsymbol{A}_{pp}\end{bmatrix},\quad \mathcal{A}_2=[\boldsymbol{A}_{H\widehat{u}},\boldsymbol{A}_{u\widehat{u}},\boldsymbol{A}_{p\widehat{u}}]^T,\quad \mathcal{A}_3=[\boldsymbol{A}_{\widehat{u}H},\boldsymbol{A}_{\widehat{u}u},\boldsymbol{A}_{\widehat{u}p}], \tag{2.17}$$

and $\mathcal{F}=[\boldsymbol{F}_H,\boldsymbol{F}_u,\boldsymbol{F}_p]^T$. We can isolate the volume space unknowns $(\boldsymbol{H},\boldsymbol{U},\boldsymbol{P})$ element by element,

$$\begin{bmatrix}\boldsymbol{H}\\ \boldsymbol{U}\\ \boldsymbol{P}\end{bmatrix}=\mathcal{A}_1^{-1}(\mathcal{F}-\mathcal{A}_2\widehat{U}). \tag{2.18}$$

To enforce continuity of the normal component of the numerical flux one has

$$\boldsymbol{A}_{\widehat{u}H}\boldsymbol{H}+\boldsymbol{A}_{\widehat{u}u}\boldsymbol{U}+\boldsymbol{A}_{\widehat{u}p}\boldsymbol{P}+\boldsymbol{A}_{\widehat{u}\widehat{u}}\widehat{\boldsymbol{U}}=\boldsymbol{F}_{\widehat{u}}. \tag{2.19}$$

We can statically condense out the interior unknowns, leaving the only globally coupled unknown as the displacement $\widehat{U}$. Eqs. (2.18) and (2.19) result in the following discrete system

$$\mathbb{H}\widehat{\boldsymbol{U}}=\mathbb{F}, \tag{2.20}$$

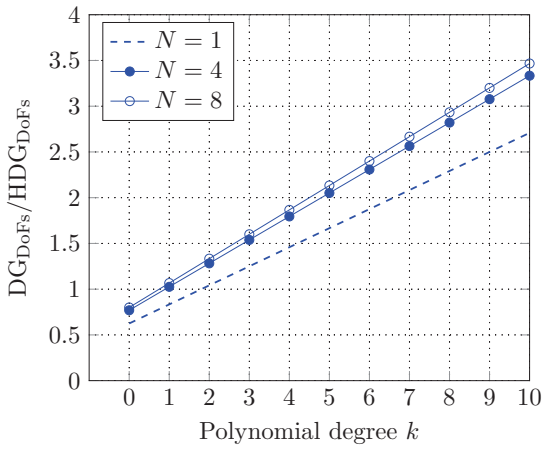


Figure 1: Comparison of DG to HDG DoFs.

where

$$\mathbb{H}=\boldsymbol{A}_{\widehat{u}\widehat{u}}-\mathcal{A}_3\mathcal{A}_1^{-1}\mathcal{A}_2,\quad \mathbb{F}=F_{\widehat{u}}-\mathcal{A}_3\mathcal{A}_1^{-1}\mathcal{F}. \tag{2.21}$$

Note that the equations in (2.21) are valid element-wise. We define $\mathbb{H}_K$ and $\mathbb{F}_K$ as the restriction of $\mathbb{H}$ and $\mathbb{F}$ to the element $K\in\mathcal{T}_h$, respectively. After Eq. (2.20) is solved, one can statically evaporate $\widehat{\boldsymbol{U}}$ to obtain $(\boldsymbol{H},\boldsymbol{U},\boldsymbol{P})$ through Eq. (2.18).

The statically condensed linear system gives rise to fewer DoFs than classical DG methods, especially for higher polynomial degrees. Below we give a brief complexity comparison. More thorough analyses including different dimensions and element types are carried out in [33]. For simplicity, we assume that the underlying domain is a cube, and it is partitioned into $N\times N\times N$ hexahedral elements. Each hexahedron is then divided into 5 tetrahedra. The number of faces in the mesh is $2N^2(3N+1)$ and the number of elements is $5N^3$. The ratio of DG to HDG DoFs becomes

$$\mathrm{HDG_{DoFs}}=3(2d_2N^2(3N+1)),\quad \mathrm{DG_{DoFs}}=3(5d_3N^3),\quad \frac{\mathrm{DG_{DoFs}}}{\mathrm{HDG_{DoFs}}}=\frac{5d_3N^3}{2d_2N^2(3N+1)}.$$

For $N>3$ and $k>0$ HDG has fewer DoFs than DG. If $N<4$ and $k>1$, then HDG has fewer DoFs than DG. Fig. 1 plots this ratio for $N\in\{1,4,8\}$. From this complexity comparison, we can conclude that the HDG method benefits from higher orders. Also, when the mesh is sufficiently large (e.g. $N>3$), the lower order discretizations match or have fewer DoFs than DG. It should also be stated that the total number of nonzero entries in the discretization matrix is smaller than that of the DG method [33].

3 GPU strategy

In this section we describe our GPU implementation. The two main steps are the local and global solvers. The local solvers are defined on the element level by the equations

in (2.21). We make extensive use of optimized linear algebra libraries for dense matrices. In particular, Basic Linear Algebra Subprograms (BLAS) [41] and Linear Algebra Package (LAPACK) [2]. The BLAS library has optimized routines for fundamental linear algebra operations like matrix-matrix multiplication, dot products, vector addition, etc. In LAPACK, optimized routines for matrix factorizations, solving linear systems, and matrix decompositions are provided.

Since the local solvers are valid element-wise, we can use a sequence of GPU-accelerated batched BLAS matrix-matrix multiplication and LAPACK operations in order to form them. The BLAS matrix-matrix multiplication routines and LAPACK operations have a favorable flop/byte ratio as well as data reuse. The local solvers are completely parallel, and batched routines allow us to process all local solvers simultaneously. Similar approaches were considered in [22,37]. Once the local solvers are generated, we may solve the globally coupled problem described by (2.20). The globally coupled system is solved iteratively with a preconditioned conjugate gradient method. A multigrid preconditioner is developed for this problem.

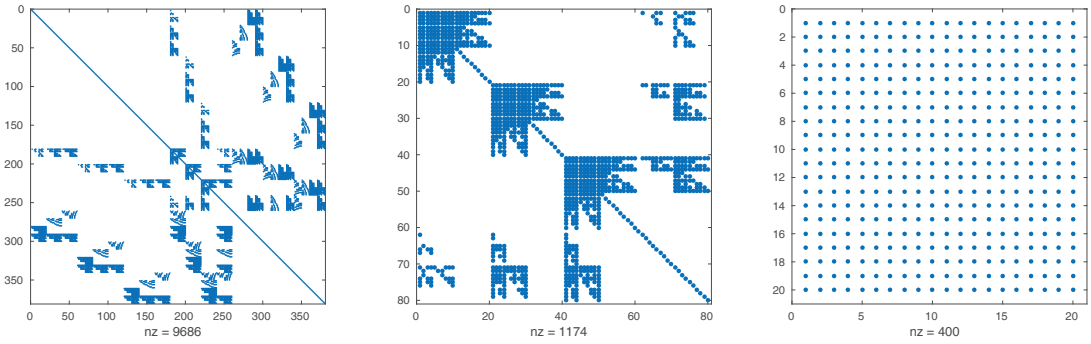
The workstation used for the computational results has the following components: NVIDIA TITAN X (2016) GPU (maximum memory bandwidth 480 GB/s), 64GB of DDR4 memory, Intel i7-6700K CPU at 4.00GHz (maximum memory bandwidth 34.1 GB/s), running Ubuntu 16.04 and CUDA version 8.0. All computations are done in double precision arithmetic, and several performance metrics are extracted through the `nvprof/nvvp` profiling tools [47].

3.1 Local solvers

The works in [22,37] consider many-core implementations of HDG methods for the Poisson problem. In our paper we extend the batched processing strategies described therein for the vector valued elasticity problem. One complication that arises is that the local solver matrices (2.17) are much larger for vector valued problems. As such, new modifications must be made to accommodate these larger matrices. Batched BLAS/LAPACK operations are intended for small to moderately sized matrices, for instance, the routines provided by cuBLAS [46]. Moreover, these routines work with dense matrices, even if they exhibit a great deal of sparsity. A naive implementation is possible, but will severely limit the problem size one can consider due to the limited GPU memory.

From Eq. (2.21), it is clear that the matrix $\mathcal{A}_1$ must be factorized or inverted. However, $\mathcal{A}_1$ is of size $(13d_3 \times 13d_3)$. GPU batched operations like `gemm` (matrix-matrix multiplication), `getrf` (LU factorization), `getrs` (backward/forward solve using LU factorization), and `getri` (compute inverse using LU factorization) are intended for matrices of small size. The batched `getrf` and `getrs` operations typically have one thread block per matrix, and one thread is responsible for one entry of the matrix. As such, for larger matrices, the resources required for these operations becomes exceedingly prohibitive.

For example, the NVIDIA TITAN X (Pascal) GPU has 28 streaming multiprocessors, each of which is capable of launching a maximum of 2048 threads. Hence, a maximum



(a) $\mathcal{A}_1$ (b) $\mathbf{Z}^{-1}$ (c) $\mathbf{Z}_m^{-1}$

Figure 2: Sparsity pattern for the local solver matrices ($k=3$).

of $28*2048=57344$ resident threads is possible on this GPU. If we set $k=3$, then $\mathcal{A}_1$ has 67600 entries, and one thread per matrix entry easily exhausts the maximum number of resident threads.

To avoid this limitation, we make explicit use of the sparsity of $\mathcal{A}_1$. The sparsity pattern for $\mathcal{A}_1$ is displayed in Fig. 2(a). Note that the upper left block of $\mathcal{A}_1$ is diagonal, which is a consequence of our choice of the PKDO orthonormal hierarchical basis, see Eq. (2.13). Instead of factoring $\mathcal{A}_1$, we consider its block 2×2 representation, and form its Schur complement. That is, we set

$$\mathcal{A}_1^{-1}=\begin{bmatrix}\mathbf{A} & \mathbf{B}\\ \mathbf{C} & \mathbf{D}\end{bmatrix}^{-1}=\begin{bmatrix}\mathbf{A}^{-1}+\mathbf{A}^{-1}\mathbf{B}\mathbf{Z}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\mathbf{Z}\\ -\mathbf{Z}\mathbf{C}\mathbf{A}^{-1} & \mathbf{Z}\end{bmatrix},\quad \mathbf{Z}=(\mathbf{D}-\mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1},$$

with

$$\mathbf{A}=\boldsymbol{A}_{HH},\quad \mathbf{B}=[\boldsymbol{A}_{Hu},\boldsymbol{A}_{Hp}],\quad \mathbf{C}=[\boldsymbol{A}_{uH},\boldsymbol{A}_{pH}]^T,\quad \mathbf{D}=\begin{bmatrix}\boldsymbol{A}_{uu} & \boldsymbol{A}_{up}\\ \boldsymbol{A}_{pu} & \boldsymbol{A}_{pp}\end{bmatrix}.$$

The Schur complement $\mathbf{Z}$ has dimension $4d_3$. We observe that the matrix $\mathbf{Z}^{-1}$ is also sparse with a rich structure, as can be seen in Fig. 2(b). Thus, we put $\mathbf{Z}^{-1}$ in a block 2×2 representation, and form its Schur complement as well:

$$\mathbf{Z}^{-1}=\begin{bmatrix}\mathbf{A}_m & \mathbf{B}_m\\ \mathbf{C}_m & \mathbf{D}_m\end{bmatrix}^{-1}=\begin{bmatrix}\mathbf{A}_m^{-1}+\mathbf{A}_m^{-1}\mathbf{B}_m\mathbf{Z}_m\mathbf{C}_m\mathbf{A}_m^{-1} & -\mathbf{A}_m^{-1}\mathbf{B}_m\mathbf{Z}_m\\ -\mathbf{Z}_m\mathbf{C}\mathbf{A}_m^{-1} & \mathbf{Z}_m\end{bmatrix},\quad \mathbf{Z}_m=(\mathbf{D}_m-\mathbf{C}\mathbf{A}_m^{-1}\mathbf{B}_m)^{-1}.$$

The matrix $\mathbf{Z}_m$ is dense (Fig. 2(c)), but has dimension d_3, which renders the batched BLAS/LAPACK operations suitable for a much wider range of polynomial orders (around $k=8$ or $k=9$ for our targeted GPUs). The difference in matrix sizes and nonzero entries can be inferred from Fig. 2. As such, we need only to generate $\mathbf{Z}_m$, and then use it to build $\mathbf{Z}$. The matrix $\mathcal{A}_1^{-1}$ does not need to be directly constructed as we only need its action against other matrices, and this is performed through operator evaluation. We

now summarize the general steps to invert $\mathcal{A}_1$, where it is understood that this procedure is done for all elements $K \in \mathcal{T}_h$ simultaneously:

1. Form $\boldsymbol{Z}_m^{-1}$ using the nonzero blocks of $\boldsymbol{Z}^{-1}$ (accomplished by a sequence of `gemm` routines).
2. Invert $\boldsymbol{Z}_m^{-1}$ to obtain $\boldsymbol{Z}_m$ (accomplished by `getri` or `getrf` and `getrs`).
3. Form $\boldsymbol{Z}$ using $\boldsymbol{Z}_m$, and the nonzero blocks of $\mathbf{A}_m$, $\mathbf{B}_m$, $\mathbf{C}_m$, and $\mathbf{D}_m$ (accomplished by a sequence of `gemm` routines).

Once $\mathbf{Z}$ is formed, $\mathbb{H}$ and $\mathbb{F}$ from Eq. (2.21) can be generated element by element simultaneously by a sequence of batched matrix-matrix multiplications. The terms $\mathbb{H}$ and $\mathbb{F}$ from Eq. (2.21) require the matrices $\mathcal{A}_1^{-1}$, $\mathcal{A}_2$, and $\mathcal{A}_3$. We do not explicitly form these matrices, rather they are constructed by operation evaluation, which results in a sequence of small dense matrix-matrix multiplications. These small matrices come from elemental contributions, e.g., volume-space mass matrices $(d_3 \times d_3)$, lift matrices $(4d_2 \times d_3)$, and flux matrices $(d_3 \times 4d_2)$. This allows us to utilize batched `gemm` operations from the cuBLAS library, which simultaneously processes the contributions for all elements. We refer the reader to [37] for further details of this approach.

We make note of a few observations regarding the cuBLAS batched `gemm` implementation. There are currently no batched matrix-vector operations in cuBLAS (e.g. matrix-vector multiplication). Further, using cuBLAS batched `gemm` for `gemv` operations (needed to build $\mathbb{F}$) was found to be very inefficient, compared to a straightforward custom batched `gemv` kernel (one thread block per matrix, one thread per row of said matrix). As the basic work unit in CUDA is a warp (group of 32 threads), padding the elemental matrices so that their dimensions round up to the nearest warp or half warp on average gives a $2\times$ speed up. In addition, we found that a straightforward custom batched `gemm` kernel outperforms cuBLAS batched `gemm` for polynomial orders $0 \leq k \leq 3$. The cuBLAS batched `gemm` operation was designed to target many different matrix sizes, but the elemental matrices for $0 \leq k \leq 3$ fall into the "very small" matrix category, and cuBLAS is not necessarily optimized for these cases. Some researchers have explored this case [18, 28, 32, 42, 65], however, this is beyond the scope of our work.

Fig. 3 has a comparison of these three kernels for a batch size of 25000, where each matrix is of size $(d_3 \times d_3)$, and $0 \leq k \leq 4$. We see that the custom kernel for $k=0$ and $k=1$ has a run time below 1 ms, and for $k \leq 2$, significant speed ups are obtained compared to the best cuBLAS implementation. A $1.4\times$ speed up is achieved for $k=3$, and cuBLAS is more efficient for $k>3$. Comparing the padded and unpadded cuBLAS implementation, we see that the padded version maintains a nearly constant run time for $0 \leq k \leq 4$. This is expected as the matrix dimensions are rounded up the nearest multiple of the warp size. So, for $k<4$, the matrices are always of size (32×32), and for $k=4$ they are size (64×64). To get a sense for how the local solver scales with polynomial degree, Fig. 4 plots the total normalized run time (the total run time for polynomial degree k divided by the total run time for polynomial degree k=7), on a mesh with 5400 unstructured tetrahedral elements.

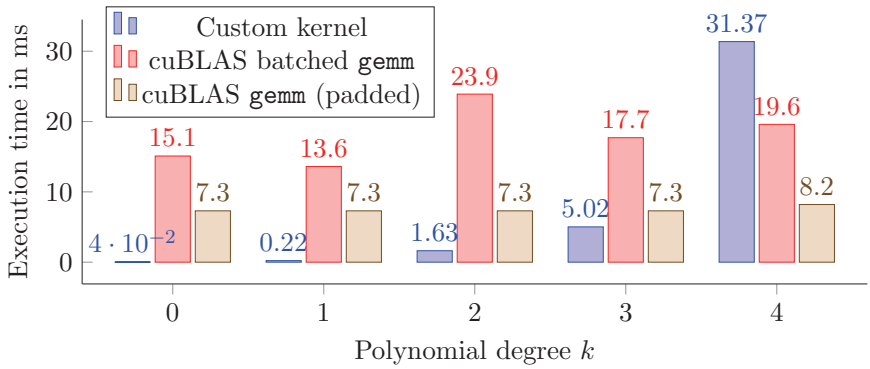


Figure 3: Comparison of run times for various GPU batched gemm operations. Time is measured in milliseconds. Batch size is 25000 matrices.

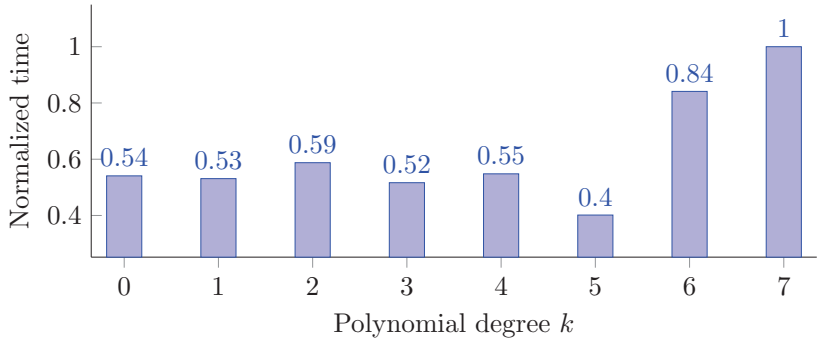


Figure 4: Normalized total run time for local solver stage of the HDG pipeline.

For $0 \leq k \leq 5$, we observe a weak scaling pattern emerge. A spike in performance occurs at $k=5$. For $k=5$, no padding is needed as the cuBLAS batched heuristics work well with the native elemental matrix dimensions. Fig. 5 has a breakdown of the dominant costs in terms of percentage of run time. It is clear that $\mathbb{H}$ on average accounts for roughly 75% of the run time for the local solvers. This is the case as $\mathbb{H}$ is block (3×3), for a total of 9 blocks of dimension $4d_2$. The combined time for both $\mathbf{Z}$ and $\mathbb{F}$ is considerably smaller, as there is much less work required.

3.2 Global solver

The core routine in our global solver is a globally coupled sparse matrix-vector multiplication. For tetrahedral meshes, and moderate polynomial orders, the authors in [6] recommend a matrix-free operation evaluation approach. That is, we do not directly assemble the global sparse matrix, but instead we use the local matrices and mesh connectivity information to perform global matrix-vector multiplications (which is needed

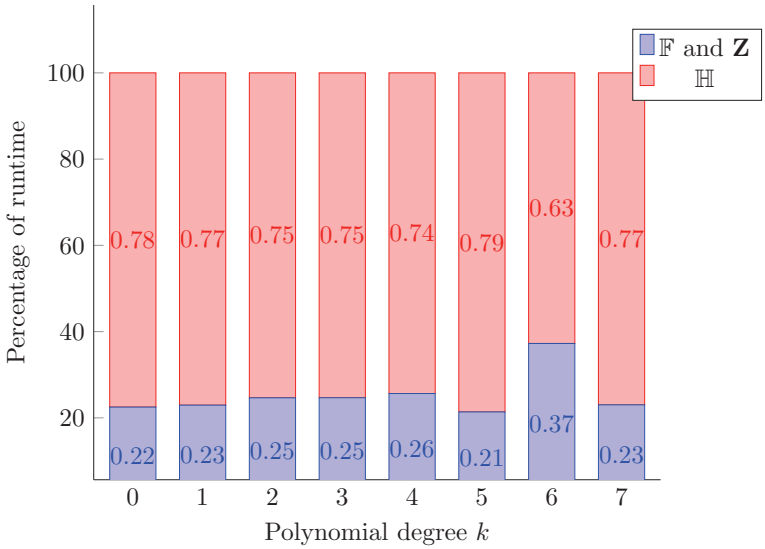


Figure 5: Percentage of run time for the local solvers.

for iterative solvers). That is, for a input $\vec{x}$, we have from (2.20)

$$\mathcal{A}\vec{x} = \sum_{K\in\mathcal{T}_h} (\mathcal{A}^K_{HDG})^T \mathbb{H}_K \mathcal{A}^K_{HDG}\vec{x},$$

where $\mathcal{A}^K_{HDG}$ is gather operator for all trace degrees of freedom on element K and $(\mathcal{A}^K_{HDG})^T$ is a scatter operator for all trace degrees of freedom on element K. As the statically condensed HDG method has its only unknowns defined on the mesh skeleton, we use a face-only connectivity. The benefit for this is that no atomics or mutex locks are needed to avoid race conditions. For interior faces, there are two elements that will contribute to the resultant vector. This algorithm results in dense local matrix-vector products that are data parallel. We assign one thread block per face to perform the dense matrix-vector products, so that one thread is responsible for the contributions of one face degree of freedom (for all displacement components). We are mainly interested in higher orders, but for lower orders, performance will drop. This is because as each thread loads data from the L1 cache, they will not make full use of the 128-byte cache line that is fetched. Thread transposition could be used to boost performance in this case [38].

A GPU-accelerated tensor contraction for the HDG method was studied in [54], which stores the HDG matrix in a block dense format, and is shown to be more efficient than a standard compressed row storage format. The approach we adopt is similar, except that our global matrix-vector is matrix-free as defined in [6].

Algorithm 1 describes our sparse matrix-vector multiply (SPMV) routine. It is a tensor contraction, as there are three displacement components, and to promote data reuse, the local input vector is put into shared memory. Moreover, our data is organized such that memory is coalesced, which is important for bandwidth performance. An additional

optimization that we make is to limit the maximum registers allocated per thread, which was determined to be 30-31 from the NVIDIA Visual Profiler (`nvvp`) for our target GPU.

Algorithm 1 Global SPMV

Input: $\mathbb{H}$ of size $(12d_2, 12d_2, |\mathcal{T}_h|)$, and x of size $(3d_2, |\Gamma_h|)$,
Output: y of size $(3d_2, |\Gamma_h|)$,
for $e \in \{1,2,\cdots,|\Gamma_h|\}$ **do**
  $K_1 \leftarrow$ element number adjacent to face e
  $K_2 \leftarrow$ element number adjacent to face e
  if $e \in \partial\Omega$ **then**
    $K_1 = K_2$
  end if
  for $K \in \{K_1, K_2\}$ **do**
    for $j \in \{1,\cdots,12d_2\}$ **do**
      $\hat{x}_j \leftarrow$ all face DoFs of x on K
    end for
    `Barrier()`
    for $j \in \{1,\cdots,12d_2\}$ **do**
      $y_{i,e} \leftarrow y_{i,e} + \mathbb{H}_{i,j,K}\hat{x}_j$
    end for
  end for
end for

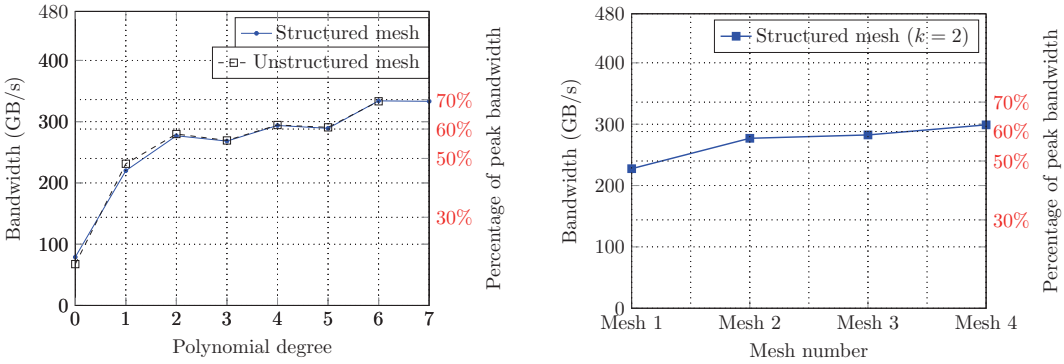
Many algorithms related to finite element problems are bandwidth bound, and SPMV falls into this category. The key components of the multigrid method (see Subsection 3.3) we use also fall into this category. As such, the most significant tuning benefits will come from memory transfer optimizations. To measure memory transfer performance, we utilize the following nvprof tool metrics:

`dram_read_throughput`, `dram_write_throughput`, `gld_efficiency`, `gst_efficiency`.

Memory bandwidth is measured as

$$\text{Bandwidth}(\text{GB/s}) = \texttt{dram\_read\_throughput} + \texttt{dram\_write\_throughput}.$$

The metrics `gld_efficiency` (global load efficiency) and `gst_efficiency` (global store efficiency) give the ratio of requested global load (store) throughput to required global load (store) throughput. We observe that `gld_efficiency` and `gst_efficiency` are both greater than or equal to 90% efficiency. This indicates that our tensor contraction kernels load (store) operations use the device's memory bandwidth effectively. Fig. 6 examines the performance of the global sparse matrix-vector multiplication. For a structured mesh of 7986 tetrahedral elements, and a unstructured mesh of 8051 tetrahedral elements, Fig. 6(a) shows the achieved bandwidth as a function of the polynomial order.



(a) Bandwidth under k refinement (b) Bandwidth under h refinement

Figure 6: Bandwidth achieved by the tensor contraction SPMV. For sufficiently large problem sizes, the tensor contraction kernel is able to achieve over 60% of the peak bandwidth.

For $2 \leq k \leq 5$, we sustain a bandwidth that is approximately 60% of the peak 480 GB/s (GTX TITAN X). For $k > 5$, the achieved bandwidth reaches close to 70% of the peak. In our tensor contraction framework, lower order polynomial degrees ($k < 2$) suffer from a lack of thread saturation at the warp level; as there are significantly fewer degrees of freedom. An optimization that could be utilized is studied in [38], where they introduce a thread transposition technique that has threads/thread blocks process contributions for multiple cells.

Fig 6(b) examines the SPMV bandwidth for different structured meshes, with the polynomial degree fixed at $k = 2$. The meshes used are uniform refinements of the unit cube, with 320, 2560, 20480, and 163840 elements. For coarse meshes, the bandwidth will be lower because the number of threads per thread block is proportional to the DoFs on a face. At a few hundred elements, the bandwidth is near 50% of the peak. As the number of elements increases, a higher bandwidth is observed, and it moves past 60% of the peak.

To get a better sense of the device utilization of the tensor contraction kernel, we study the roofline model, which allows us to identify optimization opportunities as well as model performance [64]. In Fig. 7, a roofline model associated with Fig. 6(a) is given. As expected, the tensor contraction is bandwidth bound, since it consists of small dense matrix-vector products. In addition, as the polynomial degree is increased, the throughput of the tensor contraction moves closer to the performance ceiling. Despite this, our tensor contraction is only able to achieve roughly 18% to 30% of the peak double precision throughput (317 GFLOPS/s).

The performance analysis done in this section ensures that our algorithm is efficiently using the GPU's computational resources. However, it completely ignores the accuracy of our approximations; which is a very important measure. Building off our previous work in [7], we incorporate time-accuracy measures for a more complete performance assessment in Section 5.

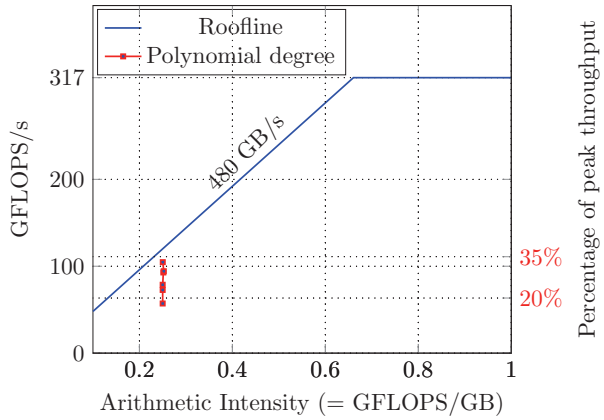


Figure 7: Roofline analysis of the tensor contraction SPMV associated with Fig. 6(a). As k is increased the throughput performance improves.

3.3 p-multigrid preconditioner

Our global solver is a Krylov-accelerated p-multigrid method (see [22] for a nested HDG multigrid method). For memory considerations on the GPU, we use the conjugate gradient method [55].

The p-multigrid preconditioner starts with a full-depth p-multigrid (conducted entirely on the GPU), then switches to an AMG method after the polynomial order has been reduced to $k=0$ (conducted on the host), and after the AMG hierarchy a direct solver is invoked. Fig. 8 has a diagram illustrating the multigrid algorithm.

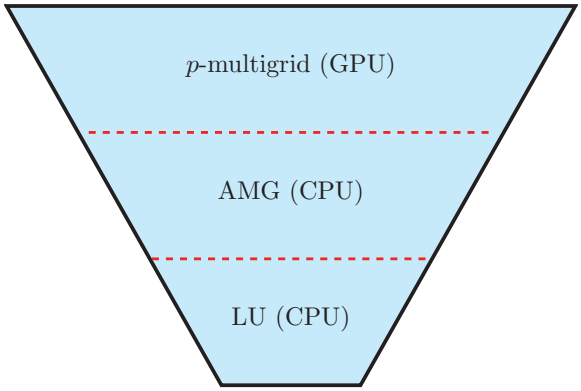


Figure 8: Diagram of the multigrid algorithm.

3.3.1 p–multigrid

The p-multigrid hierarchy is defined by the discretization of Eqs. (2.7)-(2.10) for different polynomial degrees. For simplicity, we use a rapid coarsening ratio, so that the p-

multigrid method consists of two levels, $k=k_{\max}>0$ on the fine grid and $k=0$ for the coarse grid.

The associated intergrid transfer operators are the canonical elliptic projection operators, which are applied in a matrix–free context. These operators are binary, as our basis is hierarchical. We pay careful attention to ensure that the sum of the order of these operators is strictly greater than the order of the underlying differential equation [30]. As such, the intergrid transfer operations do not reintroduce unwanted frequencies that have been eliminated by the previous multigrid operations.

3.3.2 Intergrid transfer operators

Finite element methods lend themselves well to multilevel methods, due to the flexibility in choice of underlying spaces. Consider the nested discontinuous finite element spaces $\Omega^H\subset\Omega^h$, where Ω^H is a coarse space, and Ω^h is a fine space. In our application, $\Omega^H=\boldsymbol{M}_h^0$ (e.g, coarse level corresponds to the $k=0$ approximation on a fixed mesh), and $\Omega^h=\boldsymbol{M}_h^{k_{\max}}$ (e.g, fine level corresponds to the $k=k_{\max}$ approximation on the same mesh). We define $(\Omega^H)'$ and $(\Omega^h)'$ to be the dual spaces of Ω^H and Ω^h, respectively [48].

The canonical prolongation operator uses the fact that $\Omega^H\subset\Omega^h$. That is, any coarse grid function can be expanded as a linear combination of fine grid functions. Thus, we take the prolongation operator to be the natural embedding from Ω^H into Ω^h. As such, $\boldsymbol{P}:\Omega^H\to\Omega^h$, where $\boldsymbol{P}(u^H)=u^H$ for any $u^H\in\Omega^H$. Hence, given a coarse grid function, it can be expressed as a linear combination of fine grid functions. For example, for a coarse grid basis function $\Psi_i^H\in\Omega^H$, and a fine grid basis function $\Psi_j^h\in\Omega^h$, one has

$$\Psi_i^H=\sum_j c_{ij}\Psi_j^h,$$

where c_{ij} are a unique coefficients, and the basis functions Ψ are defined in Eq. (2.12). Due to our choice of hierarchical basis, the matrix representation for the prolongation c^T is binary.

A canonical restriction is defined by a L^2 projection. The restriction operator is given by $\boldsymbol{R}:(\Omega^h)'\to(\Omega^H)'$,

$$(\boldsymbol{R}v^h,u^H)=(v^h,u^H),\quad\forall v^h\in(\Omega^h)',\quad\forall u^H\in\Omega^H.$$

Intergrid transfer operators that are effective in the multigrid context are often designed with a certain criterion. This criterion states that: the orders of sum of the intergrid transfer operators should at the very least equal the order of the underlying PDE that is to be solved (see [31, 59, 63]). The canonical intergrid transfer operators satisfy this criterion, since piecewise polynomial reconstruction is used on every element. For $k=0$, state prolongation is taken to be higher order to ensure this criterion is satisfied. Canonical intergrid transfer operators are commonly used because they are natural in the finite element framework. Other ideas regarding interpolation have been investigated, especially operator dependent interpolation (see [59, 60]).

Although the intergrid transfer operators do not dominate the cost of the multigrid method, they are cheaper for HDG method because the unknowns are on the skeleton of the mesh. Instead of using intergrid transfer operators for dimension d, the intergrid transfer operators associated with dimension $d-1$ are invoked. In other words, for the intergrid transfer operators, we use the 2D basis functions defined by Eq. (2.12), and not the 3D basis functions defined in Eq. (2.13). These operators are binary for hierarchical bases [23], and we apply them in a matrix-free manner on the GPU, as they are essentially stream operations.

3.3.3 Coarse grid solver

The computations for the $k=0$ system are handled on the host, as there is a lack of available GPU-accelerated direct solvers on GPUs. It is assembled into a sparse matrix, as matrix-free has little benefit for lower orders. An unsmoothed algebraic multigrid (AMG) method is used to solve the coarse grid system. The $k=0$ system is similar to a low order finite volume discretization, and numerically we observe a constant number of AMG iterations regardless of the mesh size.

Without additional modifications to the $k=0$ coarse problem, setting the coarse problem to be associated with $k=1$ results in a more effective p-multigrid method, but the linear system is larger, and lower order discretizations benefit more from utilizing globally assembled sparse linear systems (e.g, compressed row storage) [6,38]. Also, by using $k=0$ as a coarse problem, we can inspect a fully GPU-accelerated lower order locking-free HDG discretization in our framework.

3.3.4 Relaxation

Relaxation plays a critical role in the multigrid process. Its goal is to remove high frequencies from the error, leaving stubborn low frequencies for coarser grids to address. We use block smoothers, which are known to work well for high order discontinuous Galerkin methods [22]. Let the discretization operator $\mathbb{H}$ restricted to a face e be denoted by $\mathbb{H}_e$. The block smoother is of Jacobi type, and it is posed on faces, that is, given $e\in\Gamma_h$, from Eq. (2.20),

$$\mathbf{M}=\sum_{e=1}^{|\Gamma_h|}\mathbf{R}_e{}^T\mathbb{H}_e{}^{-1}\mathbf{R}_e,$$

where $\mathbf{R}_e$ is a binary restriction operator that transfers global DoFs to local DoFs. The relaxation is applied as follows:

$$\vec{x}^{n+1}=\vec{x}^n+\mathbf{M}(\mathbb{F}-\mathbb{H}\vec{x}^n),$$

where $\vec{x}^n$ is the nth iterate. This relaxation takes place on the GPU, where we leverage several optimized cuBLAS routines. A batched preprocessing step forms $\mathbb{H}_e^{-1}$, for all $e\in\Gamma_h$, simultaneously. These block matrices are small, of dimension d_2 for a displacement component. To apply the block smoother in the cuBLAS framework, one can batch invert the blocks (`getri`) then apply a batched dense matrix-vector multiplication (`gemm`) or

compute a batched LU factorization (`getrf`) then use batched forward/backward solves (`getrs`). The former was found to be more efficient in our implementation, as `getrf` has limited parallelism, and the statically condensed stiffness matrix has smaller blocks than classical DG methods.

For the HDG method, static condensation results in a linear system with its only unknowns belonging to the mesh skeleton. Furthermore, static condensation introduces a significant dimensionality reduction. This manifests itself in all portions of the multigrid method, including the relaxation. For standard discontinuous finite elements in d dimensions using basis functions of degree k, typical block preconditioners have blocks of the cell size, $\mathcal{O}(k^d)$. This means that the cost of inverting this block is approximately $\mathcal{O}(k^{3d})$ with $\mathcal{O}(k^{2d})$ storage, as inversion is known to cost $\mathcal{O}(N^3)$ work for a of matrix dimension N. However, the statically condensed HDG system has its blocks of size $\mathcal{O}(k^{d-1})$. As such, the cost of inverting this block is approximately $\mathcal{O}(k^{3(d-1)})$. In 3D the cost for standard DG is thus $\mathcal{O}(k^9)$ with $\mathcal{O}(k^6)$ storage, and for HDG its is $\mathcal{O}(k^6)$ with $\mathcal{O}(k^4)$ storage. This is an important complexity and storage cost reduction, especially for higher orders.

Algorithm 2 has a description of the p-multigrid method we use. The operators $\boldsymbol{R}$ and $\boldsymbol{P}$ are restriction and prolongation operators as defined in Subsection 3.3.2. The fine grid operator $\boldsymbol{A}_h$ corresponds to the HDG discretization for $k=k_{\max}$. Likewise, the coarse grid operator $\boldsymbol{A}_H$ corresponds to the HDG discretization for $k=0$.

Algorithm 2 p-multigrid preconditioner

1: **Pre-Relax**: $\vec{y}_h = \text{Smooth}(\vec{x}_h, \boldsymbol{A}_h, \vec{b}_h)$,
2: **Residual**: $\vec{r}_h = \vec{b}_h - \boldsymbol{A}_h \vec{y}_h$,
3: **Restriction**: $\vec{r}_H = \boldsymbol{R}\vec{r}_h$,
4: **Coarse grid solve**: $\vec{e}_H = (\boldsymbol{A}_H)^{-1} r_H$,
5: **Coarse correction**: $\vec{y}_h = \vec{x}_h + \boldsymbol{P}\vec{e}_H$,
6: **Post-Relax**: $\vec{x}_h = \text{Smooth}(\vec{y}_h, \boldsymbol{A}_h, \vec{b}_h)$,

4 Numerical experiments

In this section we consider numerical examples that verify and validate the HDG discretization, as well as to test the robustness of our multigrid preconditioner. The preconditioned conjugate gradient iterations are terminated after the relative error of the residual in the preconditioned norm is less than or equal to 10^{-7}.

4.1 Manufactured solution

In this section we verify convergence rates using a manufactured solution. The solution takes the following form:

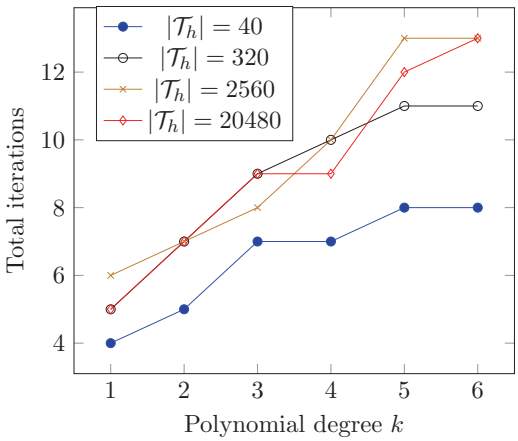


Figure 9: Total iterations from the multigrid preconditioned conjugate gradient method for the example in Subsection 4.1. Iteration counts are fairly insensitive to the polynomial order or mesh spacing.

$$\boldsymbol{u}(x,y,z) = \begin{bmatrix} (1/100)(x\sin(2\pi y)+y\cos(2\pi z)) \\ (1/100)(y\sin(2\pi z)+z\cos(2\pi x)) \\ (1/100)(z\sin(2\pi x)+x\cos(2\pi y)) \end{bmatrix},$$

where the load force $\boldsymbol{f}$ is determined from $\boldsymbol{u}$. The domain is taken as $\Omega = [-1,1]^3$, with a Neumann condition on the face corresponding to $x=1$, and the remaining boundaries have Dirichlet boundary conditions. A related problem can be found in [57]. For simplicity we fix $\mu = \lambda \equiv 1$. We consider a sequence of uniform tetrahedral meshes.

Numerically we expect that all approximate variables except the postprocessing have errors that behave like $\mathcal{O}(h^{k+1})$ in the L^2–norm [43]. The error of the postprocessed variable (with $k>0$) behaves like $\mathcal{O}(h^{k+2})$ in the L^2–norm. The computed L^2–norm errors of the approximate variables for $0 \leq k \leq 6$ can be found in Table 1. All variables except the postprocessing converge at the rate of $k+1$ in the L^2–norm. For $k>0$, the postprocessed variable superconverges at the rate of $k+2$ in the L^2–norm. We note that numerically, the trace displacement converges at a rate close to $k+3/2$, but its error is of the same magnitude as the volume displacement. The postprocessed displacement error is roughly an order of magnitude smaller than the trace (volume) displacement.

Fig. 9 examines the multigrid preconditioned conjugate gradient performance. Both the mesh size and polynomial order are varied. We see that for a given polynomial degree, the number of iterations is substantially insensitive to the mesh spacing. Similarly, for a given mesh, the iteration counts are essentially independent of increasing k.

4.2 Nearly incompressible materials

This section explores the locking–free behavior of the HDG method. The example solution we consider is taken from [52]:

Table 1: Convergence rates for a manufactured solution ($\mu=\lambda\equiv 1$).

k	$\lvert\mathcal{T}_h\rvert$	$\lVert \boldsymbol{H}-\boldsymbol{H}_h\rVert_{L^2(\Omega)}$	$\lVert \boldsymbol{u}-\boldsymbol{u}_h\rVert_{L^2(\Omega)}$	$\lVert p-p_h\rVert_{L^2(\Omega)}$	$\lVert \hat{\boldsymbol{u}}-\hat{\boldsymbol{u}}_h\rVert_{L^2(\Omega)}$	$\lVert \boldsymbol{u}-\boldsymbol{u}^*\rVert_{L^2(\Omega)}$
0	40	5.1033e-01	6.9164e-02	2.3024e-02	1.2474e-01	6.8904e-02
	320	3.7850e-01	5.8414e-02	1.5926e-02	5.8279e-02	5.4165e-02
	2560	2.3796e-01	3.4968e-02	1.2691e-02	2.0828e-02	3.1797e-02
1	40	3.9400e-01	5.4954e-02	2.3692e-02	9.7604e-02	2.6094e-02
	320	1.6024e-01	2.5917e-02	1.1606e-02	2.2667e-02	5.4598e-03
	2560	4.4056e-02	7.2306e-03	2.3228e-03	4.0801e-03	7.4251e-04
2	40	2.2559e-01	3.3473e-02	1.2935e-02	5.6346e-02	1.3493e-02
	320	3.7014e-02	6.4845e-03	1.4037e-03	6.1141e-03	1.0967e-03
	2560	5.7105e-03	9.6493e-04	2.9360e-04	5.6032e-04	7.7537e-05
3	40	8.9438e-02	1.4638e-02	4.2630e-03	2.4721e-02	4.4644e-03
	320	8.7764e-03	1.4913e-03	5.4249e-04	1.3085e-03	1.8969e-04
	2560	5.7246e-04	9.9067e-05	2.8198e-05	5.9077e-05	6.1243e-06
4	40	3.3893e-02	5.3837e-03	1.8748e-03	8.9508e-03	1.2864e-03
	320	1.2737e-03	2.3045e-04	4.0346e-05	2.2320e-04	2.4324e-05
	2560	4.6489e-05	8.2314e-06	2.1932e-06	5.0304e-06	4.1156e-07
5	40	8.8506e-03	1.5380e-03	3.2667e-04	2.6577e-03	3.0108e-04
	320	1.9758e-04	3.5136e-05	1.1150e-05	3.2213e-05	2.9995e-06
	2560	3.1712e-06	5.7368e-07	1.4295e-07	3.5829e-07	2.4190e-08
6	40	2.3523e-03	3.9781e-04	1.1927e-04	6.8938e-04	6.6713e-05
	320	2.1255e-05	3.9637e-06	6.0651e-07	3.9668e-06	3.0352e-07
	2560	1.8652e-07	3.4382e-08	8.0346e-09	2.1888e-08	1.2501e-09

$$\boldsymbol{u}=\begin{bmatrix} 200\mu(x-x^2)^2(2y.^3-3y^2+y)(2z^3-3z^2+z) \\ -100\mu(y-y^2)^2(2x^3-3x^2+x)(2z^3-3z^2+z) \\ -100\mu(z-z^2)^2(2y^3-3^2+y)(2x^3-3x^2+x) \end{bmatrix},$$

where Ω is the unit cube, all boundaries are of Dirichlet type, and $\boldsymbol{f}$ is determined from $\boldsymbol{u}$. In terms of locking, we give special attention to the lower order discretizations $k<2$. This is because higher order continuous Galerkin discretizations may not exhibit locking behavior [27]. The shear modulus is fixed as $\mu\equiv 1$. Tables 2 and 3 display the results of the $k=0$ and $k=1$ HDG discretizations for various mesh sizes and Lamé parameters. We

Table 2: Nearly incompressible material experiments for the $k=0$ discretization. Notice that the method is locking–free for a wide range of λ.

λ	$\lvert\mathcal{T}_h\rvert$	$\lVert\boldsymbol{H}-\boldsymbol{H}_h\rVert_{L^2(\Omega)}$	$\lVert\boldsymbol{u}-\boldsymbol{u}_h\rVert_{L^2(\Omega)}$	$\lVert p-p_h\rVert_{L^2(\Omega)}$	$\lVert\hat{\boldsymbol{u}}-\hat{\boldsymbol{u}}_h\rVert_{L^2(\Omega)}$	$\lVert\boldsymbol{u}-\boldsymbol{u}^*\rVert_{L^2(\Omega)}$
10^0	40	9.96e-01	2.16e-01	1.42e-02	7.80e-02	2.15e-01
	320	6.87e-01	1.27e-01	1.45e-02	4.05e-02	1.24e-01
	2560	3.87e-01	6.65e-02	8.98e-03	1.60e-02	6.42e-02
	20480	2.01e-01	3.35e-02	4.73e-03	5.80e-03	3.22e-02
10^3	40	9.96e-01	2.16e-01	4.66e-05	7.77e-02	2.15e-01
	320	6.86e-01	1.27e-01	4.39e-05	4.05e-02	1.24e-01
	2560	3.87e-01	6.65e-02	2.58e-05	1.60e-02	6.41e-02
	20480	2.01e-01	3.35e-02	1.30e-05	5.80e-03	3.21e-02
10^6	40	9.96e-01	2.16e-01	4.67e-08	7.77e-02	2.15e-01
	320	6.86e-01	1.27e-01	4.40e-08	4.05e-02	1.24e-01
	2560	3.87e-01	6.65e-02	2.58e-08	1.60e-02	6.41e-02
	20480	2.01e-01	3.35e-02	1.30e-08	5.80e-03	3.21e-02

Table 3: Nearly incompressible material experiments for the $k=1$ discretization. The method is locking–free and $\boldsymbol{u}^*$ superconverges for a wide range of λ.

λ	$\lvert\mathcal{T}_h\rvert$	$\lVert\boldsymbol{H}-\boldsymbol{H}_h\rVert_{L^2(\Omega)}$	$\lVert\boldsymbol{u}-\boldsymbol{u}_h\rVert_{L^2(\Omega)}$	$\lVert p-p_h\rVert_{L^2(\Omega)}$	$\lVert\hat{\boldsymbol{u}}-\hat{\boldsymbol{u}}_h\rVert_{L^2(\Omega)}$	$\lVert\boldsymbol{u}-\boldsymbol{u}^*\rVert_{L^2(\Omega)}$
10^0	40	3.46e-01	8.23e-02	1.80e-02	5.96e-02	1.30e-02
	320	1.26e-01	2.55e-02	6.15e-03	1.50e-02	2.54e-03
	2560	3.57e-02	7.05e-03	1.65e-03	2.95e-03	3.91e-04
	20480	9.34e-03	1.83e-03	4.18e-04	5.37e-04	5.29e-05
10^3	40	3.44e-01	8.16e-02	6.03e-05	5.93e-02	1.38e-02
	320	1.26e-01	2.55e-02	1.76e-05	1.50e-02	2.64e-03
	2560	3.56e-02	7.04e-03	4.49e-06	2.95e-03	4.00e-04
	20480	9.33e-03	1.83e-03	1.11e-06	5.37e-04	5.38e-05
10^6	40	3.44e-01	8.16e-02	6.05e-08	5.93e-02	1.38e-02
	320	1.26e-01	2.55e-02	1.76e-08	1.50e-02	2.64e-03
	2560	3.56e-02	7.04e-03	4.50e-09	2.95e-03	4.00e-04
	20480	9.33e-03	1.83e-03	1.25e-09	5.37e-04	5.38e-05

take $\lambda\in\{1,10^3,10^6\}$, which approximately corresponds to the following Young's moduli and Poisson ratios:

$$E\in\{2.5\text{e}{+}00,2.99900099900100\text{e}{+}00,2.99999900000100\text{e}{+}00\},$$
$$\nu\in\{2.5\text{e}{-}01,4.99500499500499\text{e}{-}01,4.99999500000500\text{e}{-}01\}.$$

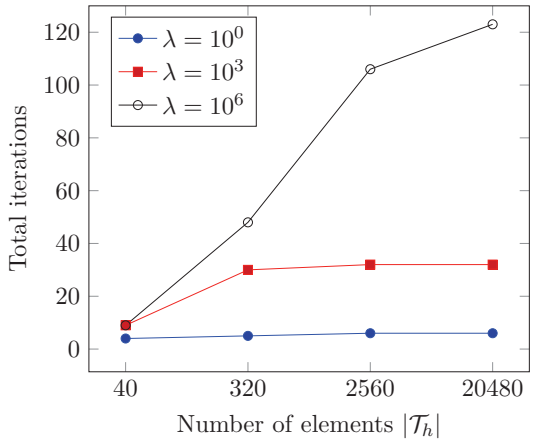


Figure 10: Total iterations from the multigrid preconditioned conjugate gradient method for the nearly incompressible example in Subsection 4.2 ($k=1$ discretization). Different Lamé parameters and mesh sizes are considered.

All variables converge at the rate of $k+1$ for $k \geq 0$ in the L^2–norm. It is apparent no volumetric locking occurs. The postprocessed variable $\boldsymbol{u}^*$ superconverges for $k > 0$, even when the Poisson ratio approaches 0.5. Since the gradient of displacement tensor converges at the rate of $k+1$ for polynomials of degree $k \geq 0$, other important quantities also converge at the rate of $k+1$. For instance, strain, stress, and vorticity.

We next examine the multigrid preconditioned conjugate gradient performance for the $k=1$ discretization. Fig. 10 has the results. For moderate Lamé parameters, the method performs very well as the mesh size increases. Numerically we observe that the total number of iterations has a dependence on the Lamé parameter.

4.3 Gravity-induced deflection of a clamped beam

This validation test case models a clamped beam deformed under its own weight. The beam is given by the following parallelepiped $\Omega = \{(x,y,z) \in \mathbb{R}^3 : \ 0 \leq x \leq 1, 0 \leq y,z \leq 0.2\}$. At the plane $x=0$, the beam is clamped (zero Dirichlet boundary condition), and the remaining boundary conditions are traction-free. A forcing function is prescribed $\boldsymbol{f} = (0,0,-\rho g)^T$, with $\rho = \mu \equiv 1$, $g = 1.6 \cdot 10^{-2}$, and $\lambda = 1.25$.

A mesh with 20480 elements and piecewise cubic polynomials are used. Fig. 11 shows the beam in its original position, which is outlined in white. The displacement in the deformed geometry is also displayed, which acts as expected; the bar is fixed near the plane $x=0$ and experiences a vertical deformation elsewhere.

4.4 Tripod under vertical load

Another validation test case for the HDG method is presented here. The example examines a tripod under vertical load. A similar example is studied in [49]. In Fig. 12 a

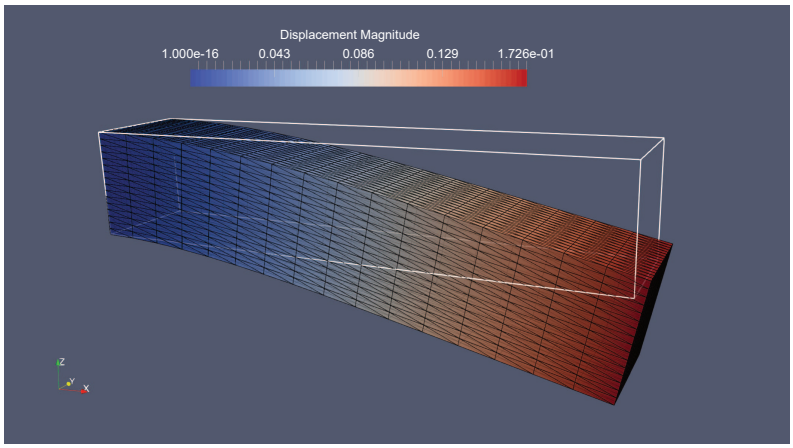
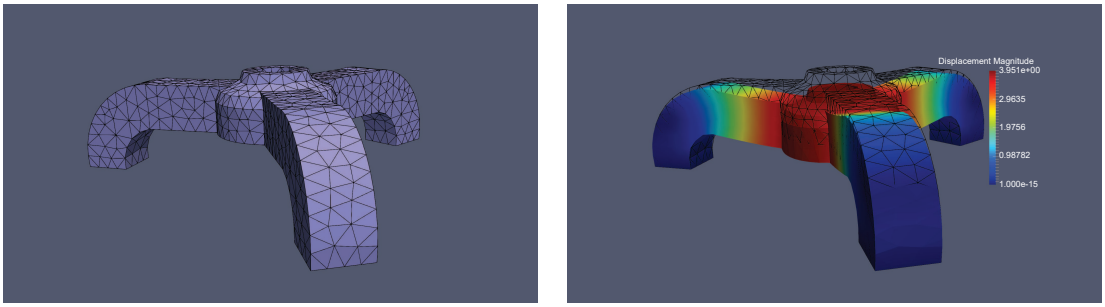


Figure 11: Beam in its original position (outline in white), and the resulting gravity-induced deflection.



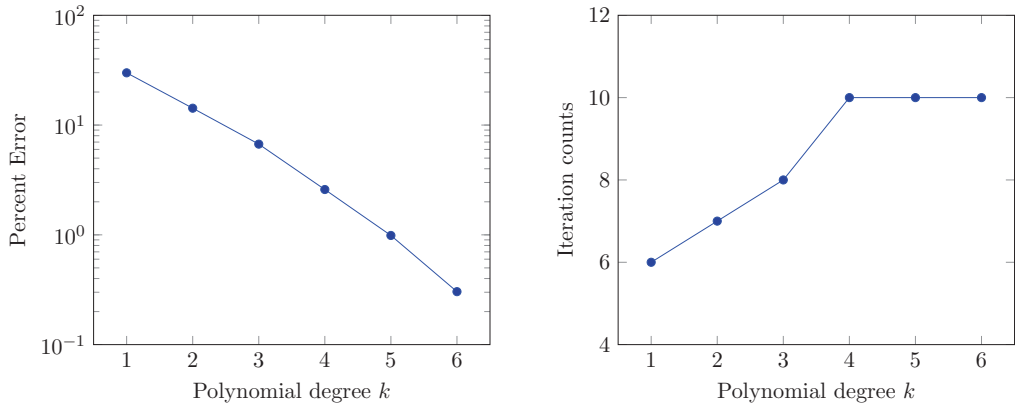
(a) Tripod mesh (b) Displacement

Figure 12: Tripod mesh with 2979 elements, and the deformed mesh with a wireframe of the reference state.

visualization of the tripod and its unstructured mesh is given.

We take the Young's moduli and Poisson ratio as $E = 10^3$ and $\nu = 0.3$, respectively. The bottom of the tripod is fixed to the ground with zero Dirichlet boundary conditions. All remaining boundary faces are traction-free. A vertical load is applied to the top of the tripod. The deformed tripod overlaid with its displacement magnitudes is depicted in Fig. 12(b); and the reference tripod skeleton is outlined in black. An intuitive displacement is exhibited.

A numerical convergence study is conducted in Fig. 13. No exact solution is known, so we compute a reference maximal displacement magnitude in the vertical direction on a refined mesh. The reference value is computed on the original tripod mesh (Fig. 12(a)) that is refined uniformly three times. We compute the maximal displacement magnitude in the vertical direction for polynomial degrees one through six on the original tripod mesh with 2979 elements, and compare it to the reference value. As the polynomial order is increased, on the mesh given in Fig. 12(a), we observe convergence to the reference



(a) Percent error (b) Iteration counts

Figure 13: Percentage error for a reference maximal displacement magnitude in the vertical direction (Fig. 13(a)). Iteration counts from the global solver (Fig. 13(b)).

displacement. The percentage error is below 1% for $k=5$ and $k=6$, which can be seen in Fig. 13(a). Fig. 13(b) shows that the iteration counts from our global solver remain robust with respect to the polynomial degree. Even though the mesh is unstructured, the iteration counts are similar to those found in Subsections 4.2 and 4.1.

5 Time-accuracy inclusive performance analysis

The analysis of the performance of our global solver is continued in this section. We utilized a roofline analysis in Section 3 to assess the performance of our tensor contraction SPMV, since it is a fundamental routine in the global solver. However, the classical roofline model does not include the accuracy of HDG approximations, which leads to an incomplete analysis. FLOPS and even run-time are not holistic performance measures. For instance, lower order methods are typically cheaper and may be faster for some applications as there is less work to do. This does not take into account that low order methods are less accurate, and may need meshes that are prohibitively large in order to match the accuracy of higher order methods. Conversely, higher order methods may be able to use coarser meshes and better utilize computational hardware, but for some problems obtainable accuracy is limited and overall performance is hindered. Other complications like solver tolerances, material parameters also need to be taken into account.

Digits of efficacy (DoE) is defined in [7] as

$$\text{DoE} = -\log_{10}(E \cdot T),$$

where $E = \|\widehat{u} - \widehat{u}_h\|_{L^2(\Omega)}$, and T is time (measured in seconds). The DoE measure allows us to gauge numerical accuracy and computational cost. Another measure we use is degrees

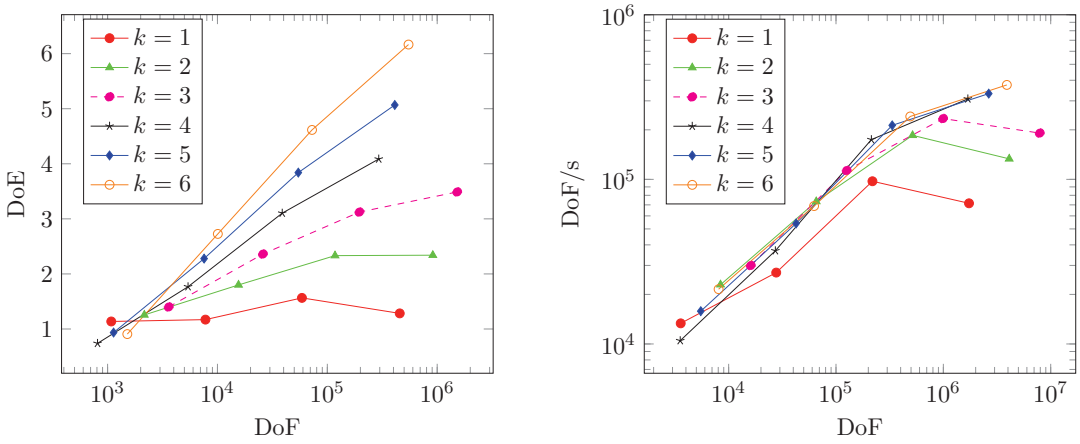
of freedom per second (DoF/s) which allows us to determine how fast a discretization processes degrees of freedom.

5.1 Manufactured solution problem revisited

We examine the manufactured solution problem given in Subsection 4.1. Fig. 14 plots the DoE vs DoF for our solver. For each curve we fix a polynomial order and vary the mesh spacing. From Figs. 14(a) and 14(b), it is evident that the higher order discretizations are more efficient, as they have a higher DoE, and processes DoFs faster than the lower order methods.

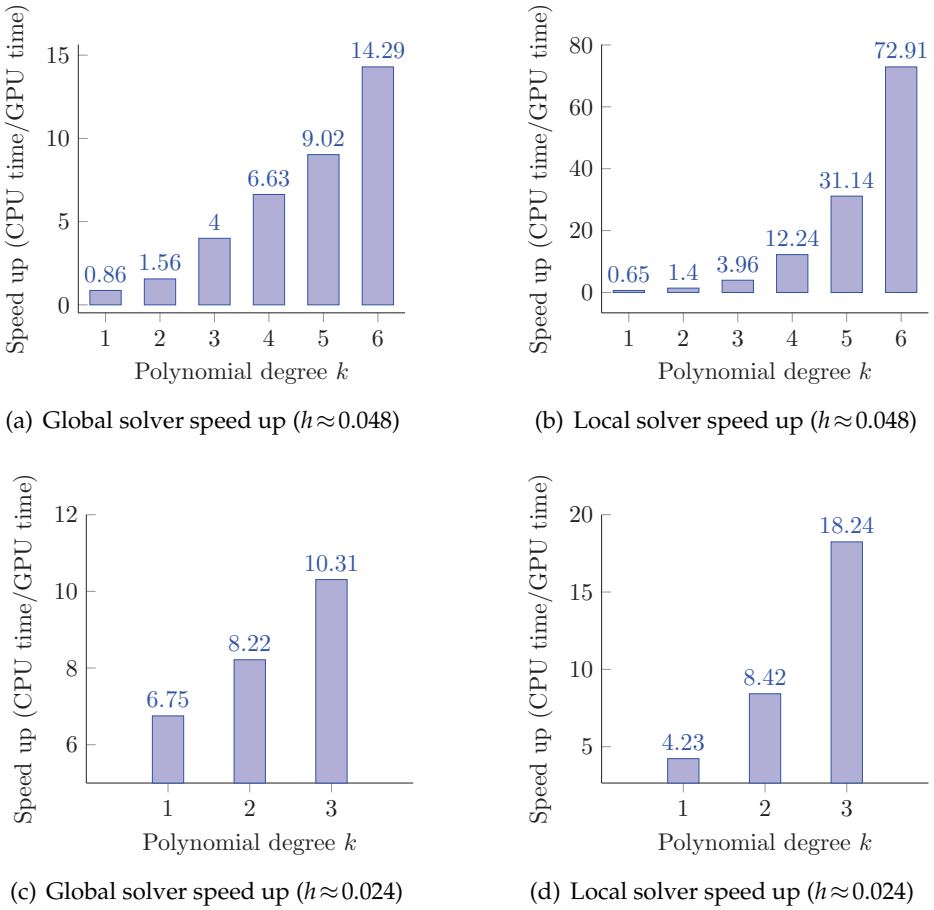
The speed ups over a serial CPU implementation is given in Fig. 15. Figs. 15(a) and 15(b) have the same mesh spacing and polynomial orders. For $k>2$, we obtain significant speed ups for both the global and local solvers. From Fig. 15(a), we see that the global solver speed ups range from $4\times$ to $14\times$, depending on the polynomial degree $k>2$. The local solvers also receive great benefit from GPU-acceleration. Fig. 15(b) displays speed ups ranging from $3.96\times$ to $72.91\times$, depending on the polynomial degree $k>2$. The local solvers give rise to larger speed ups due to the fact that they are completely data parallel.

According to Figs. 15(a) and 15(b), for $k<3$ there is limited or no speed up. We refine the mesh in this case to see if our implementation performs better on a larger workload. Indeed, Figs. 15(c) and 15(d) show that lower polynomial orders with smaller mesh spacing can obtain large speed ups. Here the global solver experiences speed ups of $6.75\times$ to $10.31\times$, depending on the polynomial degree. Similarly, the local solver experiences speed ups of $4.23\times$ to $18.24\times$, depending on the polynomial degree.



(a) DoE vs DoF (b) DoF/s vs DoF

Figure 14: Time-accuracy performance analysis for the compressible manufactured solution.



(a) Global solver speed up ($h \approx 0.048$)

(b) Local solver speed up ($h \approx 0.048$)

(c) Global solver speed up ($h \approx 0.024$)

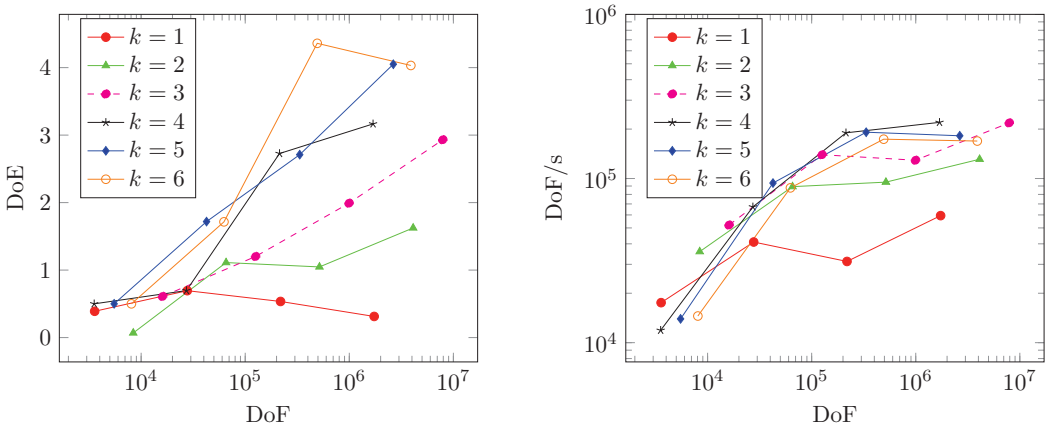
(d) Local solver speed up ($h \approx 0.024$)

Figure 15: Speed ups for the compressible manufactured solution with respect to different polynomial orders k and mesh spacings h.

5.2 Nearly incompressible materials problem revisited

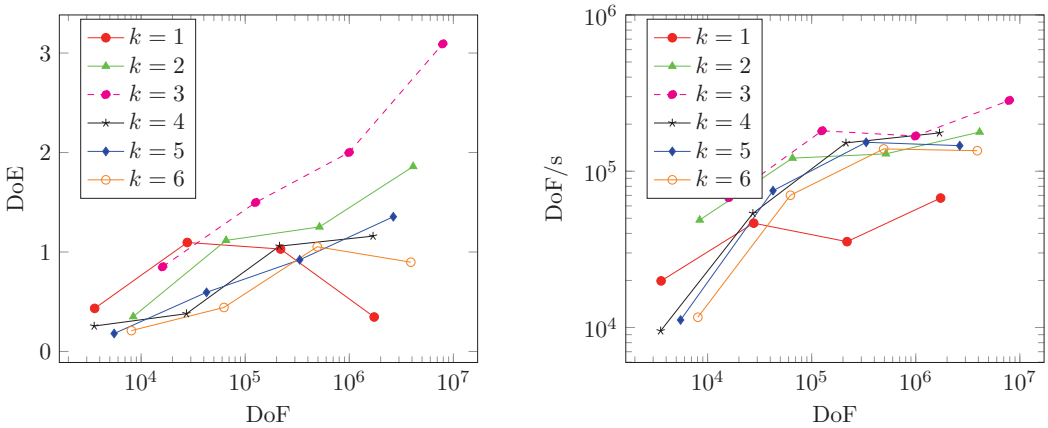
The example in Subsection 4.1 assumes compressible materials. In this situation high order methods are more efficient, according to our TAS spectrum analysis given in Subsection 5.1. On the other hand, the problem from Subsection 4.2 assumes nearly incompressible materials. It is not immediately clear if high order discretizations will be advantageous due to the worsening condition number from large k and λ.

The same problem from Subsection 4.2 is taken, with $\lambda = 10^3$, where the mesh spacing and polynomial degree are altered. From Fig. 16, we observe that the situation is different from Subsection 5.1, particularly for $k > 3$. The higher order discretizations have better DoE (Fig. 16(a)), but the discretizations for which $k > 2$ eventually process DoFs at a faster rate than $k = 6$ (Fig. 16(b)). Moreover, for $k > 3$, the DoE measures are much more competitive with one another. This is in contrast to the results in Subsection 5.1.



(a) DoE vs DoF (b) DoF/s vs DoF

Figure 16: Time-accuracy performance analysis for the nearly incompressible problem ($\lambda=10^3$).



(a) DoE vs DoF (b) DoF/s vs DoF

Figure 17: Time-accuracy performance analysis for the nearly incompressible problem ($\lambda=10^6$).

In the next experiment we increase the Lamé parameter to $\lambda=10^6$. Fig. 17 houses the results. Due to round off-errors and deterioration of the multigrid preconditioner, HDG discretizations for $k>3$ are no longer more efficient. Fig. 17(a) shows that $k=2$ and $k=3$ have the best computational efficiency, and are more reliable than the other discretizations. The processing of DoFs is competitive, as can be seen in Fig. 17(a). High order is still beneficial, but now we deduce that $k=3$ is the most effective discretization, since it has larger DoE and DoF/s measures.

We note that the classical performance analysis conducted in Section 3 does not provide the same insights as the TAS spectrum. The TAS measures take into account the

accuracy of our application, which results in a holistic performance model. Although better device utilization is observed for large k (e.g., Fig. 6(a)), this does not allow us to infer that larger k yields the best performance for all applications.

6 Conclusions

We presented a GPU-accelerated locking-free HDG method for linear elasticity. Several numerical experiments verify and validate the method. By conducting a traditional performance analysis, we demonstrated that our implementation efficiently uses the GPU's computational hardware. Significant speed ups over a optimized serial CPU code were obtained for both the global and local solvers, which dominate the run-time of our implementation. In order to accomplish this, a novel treatment of the local solvers taking advantage of the sparsity pattern was examined. By leveraging the optimized GPU-accelerated cuBLAS library, we showed how the resulting local solver algorithm can be efficiently implemented. The performance critical areas of our global solver was analyzed through a roofline performance model. In addition, we developed a multigrid preconditioner, and assessed its algorithmic performance.

Traditional performance analysis ignores solution accuracy, and a repercussion of this is that they may yield misleading performance assessments. We deployed the novel time-accuracy-size spectrum, so that we could arrive at meaningful conclusions about the best HDG discretization to select for a given application. TAS performance measures enhance our understanding of the parallel performance of computational software.

Acknowledgments

The author would like to thank Professor R. A. Tapia for directing the XSEDE Scholars Program (2011-2017), as well as A. Weeden and Dr. R. Panoff for fruitful discussions that inspired this manuscript. This work used the Extreme Science and Engineering Discovery Environment (XSEDE, 2016), which is supported by National Science Foundation grant number ACI-1053575.

References

[1] M. ADAMS, M. BREZINA, J. HU, AND R. TUMINARO, *Parallel multigrid smoothing: polynomial versus Gauss–Seidel*, Journal of Computational Physics, 188 (2003), pp. 593–610.

[2] E. ANDERSON, Z. BAI, C. BISCHOF, L. S. BLACKFORD, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, ET AL., *LAPACK Users' guide*, SIAM, 1999.

[3] H. ANZT, S. TOMOV, M. GATES, J. DONGARRA, AND V. HEUVELINE, *Block-asynchronous multigrid smoothers for GPU-accelerated systems*, Procedia Computer Science, 9 (2012), pp. 7–16.

[4] N. Bell, S. Dalton, and L. N. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM Journal on Scientific Computing, 34 (2012), pp. C123–C152.
[5] N. Bell and M. Garland, *Cusp: Generic parallel algorithms for sparse matrix and graph computations*, Version 0.3. 0, 35 (2012).
[6] C. Cantwell, S. Sherwin, R. Kirby, and P. Kelly, *From h to p efficiently: Strategy selection for operator evaluation on hexahedral and tetrahedral elements*, Computers & Fluids, 43 (2011), pp. 23–28.
[7] J. Chang, M. S. Fabien, M. G. Knepley, and R. T. Mills, *Comparative study of finite element methods using the Time-Accuracy-Size (TAS) spectrum analysis*, ArXiv e-prints, (2018).
[8] J. Chang, K. B. Nakshatrala, M. G. Knepley, and L. Johnsson, *A performance spectrum for parallel computational frameworks that solve PDEs*, Concurrency and Computation: Practice and Experience, 30 (2018), p. e4401.
[9] G. Chen and X. Xie, *A robust weak Galerkin finite element method for linear elasticity with strong symmetric stresses*, Computational Methods in Applied Mathematics, 16 (2016), pp. 389–408.
[10] J. Cheng, X. Liu, T. Liu, and H. Luo, *A parallel, high-order direct discontinuous Galerkin method for the Navier-Stokes equations on 3D hybrid grids*, Communications in Computational Physics, 21 (2017), pp. 1231–1257.
[11] P. Ciarlet, *The Finite Element Method for Elliptic Problems*, Studies in Mathematics and its Applications, Elsevier Science, 1978.
[12] B. Cockburn, *An introduction to the discontinuous Galerkin method for convection-dominated problems*, in Advanced numerical approximation of nonlinear hyperbolic equations, Springer, 1998, pp. 150–268.
[13] ——, *Static condensation, hybridization, and the devising of the HDG methods*, in Building bridges: connections and challenges in modern approaches to numerical partial differential equations, Springer, 2016, pp. 129–177.
[14] B. Cockburn, D. A. Di Pietro, and A. Ern, *Bridging the hybrid high-order and hybridizable discontinuous Galerkin methods*, ESAIM: Mathematical Modelling and Numerical Analysis, 50 (2016), pp. 635–650.
[15] B. Cockburn and G. Fu, *Devising superconvergent HDG methods with symmetric approximate stresses for linear elasticity by M-decompositions*, IMA Journal of Numerical Analysis, 38 (2017), pp. 566–604.
[16] B. Cockburn, N. Nguyen, and J. Peraire, *HDG methods for hyperbolic problems*, in Handbook of Numerical Analysis, vol. 17, Elsevier, 2016, pp. 173–197.
[17] B. Cockburn and K. Shi, *Superconvergent HDG methods for linear elasticity with weakly symmetric stresses*, IMA Journal of Numerical Analysis, 33 (2012), pp. 747–770.
[18] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Proceedings of the nineteenth annual ACM symposium on Theory of computing, (1987), pp. 1–6.
[19] B. Dally, *Computer architecture in the many-core era*, in Computer Design, 2006. ICCD 2006. International Conference on, IEEE, 2007, pp. 1–1.
[20] D. A. Di Pietro and A. Ern, *A hybrid high-order locking-free method for linear elasticity on general meshes*, Computer Methods in Applied Mechanics and Engineering, 283 (2015), pp. 1–21.
[21] M. Dubiner, *Spectral methods on triangles and other domains*, Journal of Scientific Computing, 6 (1991), pp. 345–390.
[22] M. S. Fabien, M. G. Knepley, and B. Riviere, *Heterogeneous computing for a hybridizable discontinuous Galerkin geometric multigrid method*, arXiv preprint arXiv:1705.09907, (2017).

[23] K. J. FIDKOWSKI, T. A. OLIVER, J. LU, AND D. L. DARMOFAL, *p-multigrid solution of high-order discontinuous Galerkin discretizations of the compressible navier–stokes equations*, Journal of Computational Physics, 207 (2005), pp. 92–113.
[24] D. FORTUNATO, C. H. RYCROFT, AND R. SAYE, *Efficient operator-coarsening multigrid schemes for local discontinuous Galerkin methods*, ArXiv e-prints, (2018).
[25] G. FU, B. COCKBURN, AND H. STOLARSKI, *Analysis of an HDG method for linear elasticity*, International Journal for Numerical Methods in Engineering, 102 (2015), pp. 551–575.
[26] A. GRUNDMANN AND H.-M. MÖLLER, *Invariant integration formulas for the n-simplex by combinatorial methods*, SIAM Journal on Numerical Analysis, 15 (1978), pp. 282–290.
[27] H. HAKULA, Y. LEINO, AND J. PITKÄRANTA, *Scale resolution, locking, and high-order finite element modelling of shells*, Computer methods in applied mechanics and engineering, 133 (1996), pp. 157–182.
[28] A. HEINECKE, H. PABST, AND G. HENRY, *Libxsmm: a high performance library for small matrix multiplications*, Technical Report, (2015).
[29] B. HELENBROOK, D. MAVRIPLIS, AND H. ATKINS, *Analysis of"p"-multigrid for continuous and discontinuous finite element discretizations*, in 16th AIAA Computational Fluid Dynamics Conference, 2003, p. 3989.
[30] P. HEMKER, *On the order of prolongations and restrictions in multigrid procedures*, Journal of Computational and Applied Mathematics, 32 (1990), pp. 423–429.
[31] P. W. HEMKER, *On the order of prolongations and restrictions in multigrid procedures*, Journal of Computational and Applied Mathematics, 32 (1990), pp. 423 – 429.
[32] J. HUANG, T. M. SMITH, G. M. HENRY, AND R. A. VAN DE GEIJN, *Strassen's algorithm reloaded*, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, (2016), p. 59.
[33] A. HUERTA, A. ANGELOSKI, X. ROCA, AND J. PERAIRE, *Efficiency of high-order elements for continuous and discontinuous Galerkin methods*, International Journal for numerical methods in Engineering, 96 (2013), pp. 529–560.
[34] A. HUNGRIA, D. PRADA, AND F.-J. SAYAS, *HDG methods for elastodynamics*, Computers & Mathematics with Applications, 74 (2017), pp. 2671–2690.
[35] H. KABARIA, A. J. LEW, AND B. COCKBURN, *A hybridizable discontinuous Galerkin formulation for non-linear elasticity*, Computer Methods in Applied Mechanics and Engineering, 283 (2015), pp. 303–329.
[36] D. E. KEYES, L. C. MCINNES, C. WOODWARD, W. GROPP, E. MYRA, M. PERNICE, J. BELL, J. BROWN, A. CLO, J. CONNORS, ET AL., *Multiphysics simulations: Challenges and opportunities*, The International Journal of High Performance Computing Applications, 27 (2013), pp. 4–83.
[37] J. KING, S. YAKOVLEV, Z. FU, R. M. KIRBY, AND S. J. SHERWIN, *Exploiting batch processing on streaming architectures to solve 2D elliptic finite element problems: A hybridized discontinuous Galerkin (HDG) case study*, Journal of Scientific Computing, 60 (2014), pp. 457–482.
[38] M. G. KNEPLEY, K. RUPP, AND A. R. TERREL, *Finite element integration with quadrature on the GPU*, arXiv preprint arXiv:1607.04245, (2016).
[39] T. KOORNWINDER, *Two-variable analogues of the classical orthogonal polynomials*, in Theory and application of special functions (Proc. Advanced Sem., Math. Res. Center, Univ. Wisconsin, Madison, Wis., 1975), Academic Press New York, 1975, pp. 435–495.
[40] M. LARSON AND F. BENGZON, *The Finite Element Method: Theory, Implementation, and Applications*, Texts in Computational Science and Engineering, Springer Berlin Heidelberg, 2013.
[41] C. L. LAWSON, R. J. HANSON, D. R. KINCAID, AND F. T. KROGH, *Basic linear algebra sub-*

programs for fortran usage, ACM Transactions on Mathematical Software (TOMS), 5 (1979), pp. 308–323.
[42] I. MASLIAH, A. ABDELFATTAH, A. HAIDAR, S. TOMOV, M. BABOULIN, J. FALCOU, AND J. DONGARRA, *High-performance matrix-matrix multiplications of very small matrices*, European Conference on Parallel Processing, (2016), pp. 659–671.
[43] N. C. NGUYEN AND J. PERAIRE, *Hybridizable discontinuous Galerkin methods for partial differential equations in continuum mechanics*, Journal of Computational Physics, 231 (2012), pp. 5955–5988.
[44] N. C. NGUYEN, J. PERAIRE, AND B. COCKBURN, *A hybridizable discontinuous Galerkin method for Stokes flow*, Computer Methods in Applied Mechanics and Engineering, 199 (2010), pp. 582–597.
[45] J. NICKOLLS AND W. J. DALLY, *The GPU computing era*, IEEE micro, 30 (2010).
[46] NVIDIA, *cuBLAS library*, NVIDIA Corporation, Santa Clara, California, 15 (2008), p. 31.
[47] ———, *CUDA*. `https://developer.nvidia.com/cuda-zone`, 2008. Accessed: 2017-05-30.
[48] M. A. OLSHANSKII AND E. E. TYRTSHNIKOV, *Iterative methods for linear systems: theory and applications*, vol. 138, SIAM, 2014.
[49] L. N. OLSON, J. SCHRODER, AND R. S. TUMINARO, *A new perspective on strength measures in algebraic multigrid*, Numerical Linear Algebra with Applications, 17 (2010), pp. 713–733.
[50] R. G. OWENS, *Spectral approximations on the triangle*, Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, 454 (1998), pp. 857–872.
[51] J. PRORIOL, *Sur une famille de polynomes á deux variables orthogonaux dans un triangle*, Comptes rendus hebdomadaires des seances de l'Academie des sciences, 245 (1957), pp. 2459–2461.
[52] H. QI, L.-H. WANG, AND W.-Y. ZHENG, *On locking-free finite element schemes for three-dimensional elasticity*, Journal of Computational Mathematics, (2005), pp. 101–112.
[53] W. QIU, J. SHEN, AND K. SHI, *An HDG method for linear elasticity with strong symmetric stresses*, arXiv preprint arXiv:1312.1407, (2013).
[54] X. ROCA, N. C. NGUYEN, AND J. PERAIRE, *GPU-accelerated sparse matrix-vector product for a hybridizable discontinuous Galerkin method*, in 49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition, 2011, p. 687.
[55] Y. SAAD, *Iterative methods for sparse linear systems*, vol. 82, siam, 2003.
[56] R. SEVILLA, M. GIACOMINI, AND A. HUERTA, *A locking-free face-centred finite volume (FCFV) method for linear elasticity*, arXiv preprint arXiv:1806.07500, (2018).
[57] R. SEVILLA, M. GIACOMINI, A. KARKOULIAS, AND A. HUERTA, *A super–convergent hybridisable discontinuous Galerkin method for linear elasticity*, arXiv preprint arXiv:1802.02123, (2018).
[58] S.-C. SOON, B. COCKBURN, AND H. K. STOLARSKI, *A hybridizable discontinuous Galerkin method for linear elasticity*, International journal for numerical methods in engineering, 80 (2009), pp. 1058–1092.
[59] C. W. O. U. TROTTENBERG AND A. SCHÜLLER, *Multigrid*, Academic Press, 2001.
[60] W. L. WAN, T. F. CHAN, AND B. SMITH, *An energy-minimizing interpolation for robust multigrid methods*, SIAM Journal on Scientific Computing, 21 (1999), pp. 1632–1649.
[61] S. WANDZURAT AND H. XIAO, *Symmetric quadrature rules on a triangle*, Computers & Mathematics with Applications, 45 (2003), pp. 1829–1840.
[62] R. WANG AND R. ZHANG, *A weak Galerkin finite element method for the linear elasticity problem in mixed form*, Journal of Computational Mathematics, 36 (2018), pp. 469–491.
[63] P. WESSELING, *An introduction to multigrid methods*, Pure and applied mathematics, John Wiley & Sons Australia, Limited, 1992.

[64] S. WILLIAMS, A. WATERMAN, AND D. PATTERSON, *Roofline: an insightful visual performance model for multicore architectures*, Communications of the ACM, 52 (2009), pp. 65–76.
[65] V. V. WILLIAMS, *Multiplying matrices in o(n2.373) time*, Technical Report, (2014).
[66] Q. ZHANG, C. ZHU, AND M. ZHU, *Three-dimensional numerical simulation of droplet evaporation using the lattice Boltzmann method based on GPU-CUDA accelerated algorithm*, CCommunications in Computational Physics, 23 (2018), pp. 1150–1166.
[67] J. ZHAO AND H. TANG, *Runge-Kutta Central Discontinuous Galerkin Methods for the Special Relativistic Hydrodynamics*, Communications in Computational Physics, 22 (2017), pp. 643–682.