49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition
4 - 7 January 2011, Orlando, Florida

AIAA 2011-687

GPU-accelerated sparse matrix-vector product for a hybridizable discontinuous Galerkin method

X. Roca*
Universitat Politècnica de Catalunya, Barcelona, 08034, Spain
N. C. Nguyen† and J. Peraire‡
Massachusetts Institute of Technology, Cambridge, MA 02139, USA

The iterative solution of the large systems of equations that result from discontinuous Galerkin (DG) discretizations require the ability to carry out fast matrix-vector products. DG matrices have a sparse block structure with a constant number of non-zero equal-sized non-overlapping blocks per row. General-purpose sparse matrix-vector product algorithms are not designed to exploit the specific structure of the DG matrices and, as a consequence, result in sub-optimal performance. To address this issue, we propose a sparse matrix-vector product for DG discretizations based on a dense tensor contraction. A GPU implementation of the proposed algorithm for a hybridizable discontinuous Galerkin (HDG) method is tested on the NVIDIA GEFORCE GTX 285. The results show that the tensor contraction performs at about 20 to 25 GFLOP/s in double precision with a sustained efficiency of more than 40% (60 GBytes/s) of the peak memory bandwidth (160 GBytes/s). Moreover, for HDG matrices in double precision, the proposed method is 2 times faster than the general sparse matrix-vector products provided by the GPU library CUSPARSE and about 30 times faster than MATLAB running on a CPU.

I. Introduction

The hybridizable discontinuous Galerkin (HDG) method was recently introduced in Ref. [1] and further developed in Ref. [2–11] for a wide range of problems. The HDG method is a discontinuous Galerkin discretization with a particular choice of the numerical fluxes. Indeed, the main (only) difference between many existing DG methods and the HDG method is the definition of the numerical trace of the field variable. However, like many mixed finite element methods, the HDG method can be hybridized by considering the numerical trace of the field variable as an independent unknown and introducing an additional equation which explicitly enforces the conservation property. In this way, both the approximate field and gradient variables can be eliminated locally to produce a reduced system which only involves the trace of the field variable. Since the numerical trace is single-valued and defined over the element faces, the HDG method has significantly less globally coupled degrees of freedom than other DG methods. Moreover, the HDG method is compact in the sense that it only connects the degrees of freedom of the neighboring faces. More specifically, for triangular elements, the degrees of freedom on an interior (respectively, boundary) edge are connected to the degrees of freedom on four (respectively, two) neighboring edges; and, for tetrahedral elements, the degrees of freedom on an interior (respectively, boundary) face are connected to the degrees of freedom on six (respectively, three) neighboring faces. This sparse block-structure consisting of dense, non-overlapping, equally sized blocks of the HDG matrices is similar to that of other DG methods such as the method of Bassi and Rebai[12] or the compact DG (CDG) method.[13] In these DG methods however, the degrees of freedom are associated to the elements rather than the faces and the connections are to neighboring elements rather than faces.

*Research Scientist, Laboratori de Càlcul Numèric (LaCàN), Departament de Matemàtica Aplicada III, Universitat Politècnica de Catalunya, Jordi Girona 1-3, 08034 Barcelona, Spain.
†Research Scientist, Department of Aeronautics and Astronautics, M.I.T., 77 Massachusetts Avenue, AIAA Member.
‡Professor, Department of Aeronautics and Astronautics, M.I.T., 77 Massachusetts Avenue, AIAA Associate Fellow.

The HDG discretization of a nonlinear system of conservation laws such as the steady-state compressible Navier-Stokes equations gives rise to a nonlinear system of algebraic equations that can be solved using Newton's method. For large problems, the resulting linear system at each Newton iteration needs to be solved using iterative methods. Efficient iterative solution algorithms rely on the ability to perform fast matrix-vector products. Developing these fast matrix-vector product algorithms implies that the following two issues be addressed. First, the sparse matrix-vector product is a memory-bound operation limited by the peak memory bandwidth of the selected computing platform. Second, general-purpose sparse matrix-vector products are not designed to exploit the specific structure of the HDG matrices.

In this paper, we describe the development of a high-performance matrix-vector product for HDG discretizations that addresses the above two issues. To alleviate the memory bandwidth constraint, we choose the graphic processor unit (GPU) as a computing platform. In recent years, using GPUs as a general-purpose computing platform has received considerable attention in the scientific community. The GPUs have better floating-point performance and memory bandwidth than conventional CPUs. Specifically, the peak memory bandwidth of a GPU is ten times greater than the one of the computer main memory. This increased performance offers significant potential benefits for memory-bound operations such as sparse matrix-vector products. In order to exploit the structure of the HDG matrices, we develop a specific-purpose sparse matrix-vector product which reduces the sparse matrix-vector product to a dense tensor contraction on the GPU. The result is a high-performance matrix-vector product that obtains a high effective memory bandwidth on the GPU by exploiting the specific structure of the HDG matrices.

Recently, the GPU implementation of DG methods with explicit time integration has been proposed in Ref. 14. They show that the GPU can speed up the performance of the DG codes by more than one order of magnitude when compared to the CPU implementation. However, explicit methods do not require the iterative solution of linear systems and therefore no sparse matrix-vector product is required. Iterative solvers have been developed for the CDG method[15] as well as other implicit DG methods.[12,16,17] To date, these methods have only been implemented on the CPU.

The interest for generic linear algebra operations on the GPU was initially focused in dense matrices and vectors.[18–20] Nevertheless, several iterative solvers on the GPU[21–24] have been proposed in the last years. Thus, general-purpose sparse matrix-vector products for the GPU have already been developed for non-block[21,25–27] and block[22,28,29] storage formats. The proposed GPU implementations exploit the high-peak memory bandwidth of the latest GPUs and their performance is superior to that of the existent highly optimized products for the CPU.[30] However, these general-purpose GPU products are not specific for HDG matrices. Here, we present the first sparse-matrix vector product for HDG matrices on the GPU.

The rest of the paper is organized as follows. First, we give in section II an overview of the HDG method for the compressible Navier-Stokes equations. Second, we describe the sparsity pattern and storage format for the HDG matrices are described in section III. Third, the implementation on the GPU of the sparse matrix-vector product is given in section IV. Finally, we present in section V some numerical results to demonstrate the performance and speedup of the proposed matrix-vector product.

II. Overview of the HDG Method

II.A. Governing Equations

We consider the steady-state compressible Navier-Stokes equations written in conservative form as

$$\begin{aligned} \boldsymbol{q} - \nabla \boldsymbol{u} &= 0, \qquad \text{in } \Omega, \\ \nabla \cdot (\boldsymbol{F}(\boldsymbol{u}) + \boldsymbol{G}(\boldsymbol{u}, \boldsymbol{q})) &= 0, \qquad \text{in } \Omega, \end{aligned} \tag{1}$$

where $\boldsymbol{u}$ is the m-dimensional vector of conserved quantities (namely, density, momentum and energy), and $\boldsymbol{F}(\boldsymbol{u})$ and $\boldsymbol{G}(\boldsymbol{u}, \boldsymbol{q})$, are the inviscid and viscous fluxes of dimension $m \times d$. The nondimensional form of the Navier-Stokes equations as well as the definition of the inviscid and viscous fluxes can be found in Ref. [31]. Of course, the Navier-Stokes equations (1) should be supplemented with appropriate boundary conditions at the inflow, outflow and solid wall boundaries, as well as a source term which are omitted here for simplicity of exposition.

To describe the HDG method for solving the Navier-Stokes equations, we introduce first some notation. We denote by $\mathcal{T}_h$ a collection of disjoint regular elements K that partition Ω and set $\partial\mathcal{T}_h := \{\partial K : K \in \mathcal{T}_h\}$. For an element K of the collection $\mathcal{T}_h$, $F = \partial K \cap \partial\Omega$ is the boundary face if the $d-1$ measure of F is

nonzero. For two elements K^+ and K^- of the collection $\mathcal{T}_h$, $F = \partial K^+ \cap \partial K^-$ is the interior face between K^+ and K^- if the $d-1$ measure of F is nonzero. We denote by $\mathcal{E}_h^o$ and $\mathcal{E}_h^\partial$ the set of interior and boundary faces, respectively. We set $\mathcal{E}_h = \mathcal{E}_h^o \cup \mathcal{E}_h^\partial$.

Let $\mathcal{P}^k(D)$ denote the space of polynomials of degree at most k on a domain D and let $L^2(D)$ be the space of square integrable functions on D. We introduce the following discontinuous finite element approximation spaces

$$\begin{aligned}\boldsymbol{W}_h^k &= \{\boldsymbol{w} \in (L^2(\mathcal{T}_h))^m \ : \ \boldsymbol{w}|_K \in (\mathcal{P}^k(K))^m, \ \forall \ K \in \mathcal{T}_h\},\\ \boldsymbol{V}_h^k &= \{\boldsymbol{v} \in (L^2(\mathcal{T}_h))^{m\times m} \ : \ \boldsymbol{v}|_K \in (\mathcal{P}^k(K))^{m\times m}, \ \forall \ K \in \mathcal{T}_h\}.\end{aligned}$$

In addition, we introduce a finite element approximation space for the approximate trace of the solution

$$\boldsymbol{M}_h^k = \{\boldsymbol{\mu} \in (L^2(\mathcal{E}_h)^m \ : \ \boldsymbol{\mu}|_F \in (\mathcal{P}^k(F))^m, \ \forall \ F \in \mathcal{E}_h\}.$$

Note that $\boldsymbol{M}_h$ consists of functions which are continuous inside the faces (or edges) $F \in \mathcal{E}_h$ and discontinuous at their borders.

Finally, we define various inner products for our finite element spaces. We write $(w,v)_{\mathcal{T}_h} := \sum_{K\in\mathcal{T}_h}(w,v)_K$, where $(w,v)_D$ denotes the integral of $w\,v$ over the domain $D \subset \mathbb{R}^d$. We also write $(\boldsymbol{w},\boldsymbol{v})_{\mathcal{T}_h} := \sum_{i=1}^m (w_i, v_i)_{\mathcal{T}_h}$ for $\boldsymbol{w},\boldsymbol{v} \in \boldsymbol{W}_h^k$ and $(\boldsymbol{w},\boldsymbol{v})_{\mathcal{T}_h} := \sum_{i=1}^m\sum_{j=1}^d (w_{ij}, v_{ij})_{\mathcal{T}_h}$ for $\boldsymbol{w},\boldsymbol{v} \in \boldsymbol{V}_h^k$. We then write $\langle \eta,\zeta\rangle_{\partial\mathcal{T}_h} := \sum_{K\in\mathcal{T}_h}\langle\eta,\zeta\rangle_{\partial K}$ and $\langle\boldsymbol{\eta},\boldsymbol{\zeta}\rangle_{\partial\mathcal{T}_h} := \sum_{i=1}^m\langle\eta_i,\zeta_i\rangle_{\partial\mathcal{T}_h}$, for $\boldsymbol{\eta},\boldsymbol{\zeta} \in \boldsymbol{M}_h^k$, where $\langle\eta,\zeta\rangle_D$ denotes the integral of $\eta\,\zeta$ over the domain $D \subset \mathbb{R}^{d-1}$.

II.B. Formulation of the HDG Method

The HDG method seeks an approximation $(\boldsymbol{q}_h, \boldsymbol{u}_h, \widehat{\boldsymbol{u}}_h) \in \boldsymbol{V}_h^k \times \boldsymbol{W}_h^k \times \boldsymbol{M}_h^k$ to the solution of the system (1) such that

$$\begin{aligned}(\boldsymbol{q}_h,\boldsymbol{v})_{\mathcal{T}_h} + (\boldsymbol{u}_h, \nabla\cdot\boldsymbol{v})_{\mathcal{T}_h} - \langle\widehat{\boldsymbol{u}}_h, \boldsymbol{v}\cdot\boldsymbol{n}\rangle_{\partial\mathcal{T}_h} &= 0, \qquad \forall \boldsymbol{v} \in \boldsymbol{V}_h^k,\\ -(\boldsymbol{F}(\boldsymbol{u}_h) + \boldsymbol{G}(\boldsymbol{u_h},\boldsymbol{q}_h), \nabla\boldsymbol{w})_{\mathcal{T}_h} + \left\langle(\widehat{\boldsymbol{h}}_h\cdot\boldsymbol{n}, \boldsymbol{w}\right\rangle_{\partial\mathcal{T}_h} &= 0, \qquad \forall \boldsymbol{w} \in \boldsymbol{W}_h^k,\\ \left\langle\widehat{\boldsymbol{h}}_h, \boldsymbol{\mu}\right\rangle_{\partial\mathcal{T}_h\setminus\partial\Omega} + \left\langle\widehat{\boldsymbol{b}}_h, \boldsymbol{\mu}\right\rangle_{\partial\Omega} &= 0, \qquad \forall \boldsymbol{\mu} \in \boldsymbol{M}_h^k,\end{aligned} \tag{2}$$

where

$$\widehat{\boldsymbol{h}}_h = (\boldsymbol{F}(\widehat{\boldsymbol{u}}_h) + \boldsymbol{G}(\widehat{\boldsymbol{u}}_h,\boldsymbol{q}_h))\cdot\boldsymbol{n} + \boldsymbol{S}(\widehat{\boldsymbol{u}}_h)(\boldsymbol{u}_h - \widehat{\boldsymbol{u}}_h). \tag{3}$$

Here $\widehat{\boldsymbol{h}}_h$ is the interior numerical flux, while $\widehat{\boldsymbol{b}}_h$ is the boundary numerical flux which depends on the types of boundary conditions applied on the domain boundary. Note also that $\boldsymbol{S}(\widehat{\boldsymbol{u}}_h)$ is a *local stabilization matrix* which has an important effect on both the stability and accuracy of the resulting scheme. We refer to[11] for a detail discussion on how $\widehat{\boldsymbol{b}}_h$ can be defined for different types of boundary conditions and how $\boldsymbol{S}(\widehat{\boldsymbol{u}}_h)$ can be chosen to ensure stability and consistency.

By applying the Newton-Raphson procedure to solve the nonlinear system (2), we obtain at every Newton step a matrix system of the form

$$\begin{bmatrix} A & B & E\\ C & D & L\\ M & N & P\end{bmatrix}\begin{pmatrix}\mathbf{q}\\ \mathbf{u}\\ \hat{\mathbf{u}}\end{pmatrix} = \begin{pmatrix} H\\ F\\ G\end{pmatrix}, \tag{4}$$

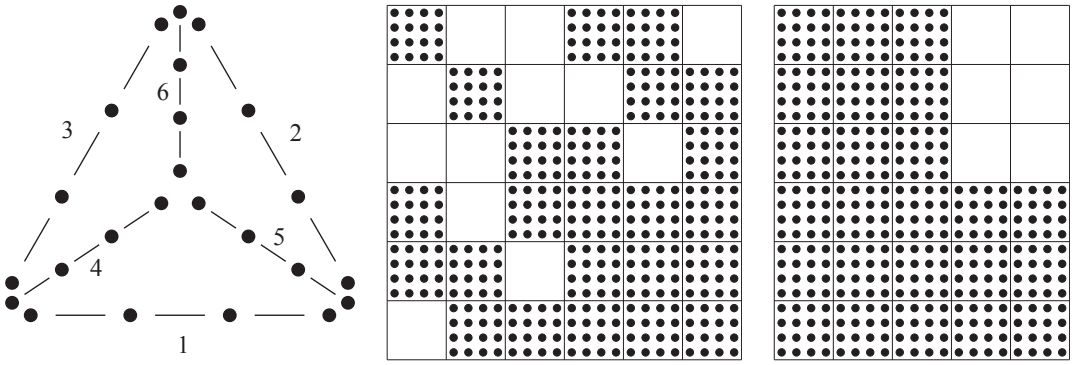
where $\mathbf{q}$, $\mathbf{u}$ and $\hat{\mathbf{u}}$ are the vectors of degrees of freedom of $\boldsymbol{q}_h$, $\boldsymbol{u}_h$ and $\widehat{\boldsymbol{u}}_h$, respectively. We note that when the degrees of freedom for $\boldsymbol{q}_h$, $\boldsymbol{u}_h$ are grouped together and ordered in an element-wise fashion, the corresponding matrix $[A\ B; C\ D]$ has block-diagonal structure. Therefore, we can eliminate both $\mathbf{q}$ and $\mathbf{u}$ to obtain a reduced linear system in terms of $\hat{\mathbf{u}}$ as

$$\mathbf{A}\,\hat{\mathbf{u}} = \mathbf{b}, \tag{5}$$

where

$$\mathbf{A} = P - \begin{bmatrix} M & N\end{bmatrix}\begin{bmatrix} A & B\\ C & D\end{bmatrix}^{-1}\begin{bmatrix} E\\ L\end{bmatrix}, \qquad \mathbf{b} = G - \begin{bmatrix} M & N\end{bmatrix}\begin{bmatrix} A & B\\ C & D\end{bmatrix}^{-1}\begin{bmatrix} H\\ F\end{bmatrix}. \tag{6}$$

For large problems, the linear system (5) has to be solved using iterative methods. Any iterative solver will require the evaluation of the matrix-vector multiplication $\mathbf{y} = \mathbf{A}\mathbf{x}$ many times. The structure of this matrix $\mathbf{A}$ is described below.



(a) (b) (c)

Figure 1. Sparsity structure of the system matrix: (a) a mesh with six faces ($n_f = 6$) and four nodes per face ($n_p = 4$); (b) sparse pattern in dense blocks for a problem with one component ($n_c = 1$); and (c) the storage in dense block format.

III. Sparsity Pattern and Storage of the HDG Matrix

The matrix of a global system $\mathbf{A}\hat{\mathbf{u}} = \mathbf{b}$ for a hybridizable discontinuous Galerkin discretization is sparse and its structure is determined by the numbering of the nodal unknowns within the vector $\hat{\mathbf{u}}$. One can think of the components of $\hat{\mathbf{u}}$ as depending on three indices, the face number k_f, the solution component k_c and the local number k_p of the node within the face k_f. With this notation, the global system of equations can be written as

$$\sum_{k_f=1}^{n_f}\sum_{k_c=1}^{n_c}\sum_{k_p=1}^{n_p}\mathbf{A}_{i_p i_c i_f k_p k_c k_f}\hat{\mathbf{u}}_{k_p k_c k_f} = \mathbf{b}_{i_p i_c i_f},$$

for $i_f = 1,\ldots,n_f,\ i_c = 1,\ldots,n_c$ and $i_p = 1,\ldots,n_p$, where n_f is the number of mesh faces, n_c is the number of solution components and n_p is the number of nodes over a face. It seems natural to consider a numbering that groups the unknowns of the same face and the same component contiguously. Thus, we can define a single index k' as

$$k' := k'(k_p, k_c, k_f) = k_p + n_p(k_c - 1) + n_p n_c(k_f - 1),$$

to linearly order all the unknowns within the vector $\hat{\mathbf{u}}$. Similarly, the rows of the matrix $\mathbf{A}$ can be addressed by a single index i', given by

$$i' := i'(i_p, i_c, i_f) = i_p + n_p(i_c - 1) + n_p n_c(i_f - 1).$$

This numbering leads to a sparse matrix $\mathbf{A}$ which has the two following important properties:

- The nonzero entries are grouped into dense, non-overlapping blocks of size $n_p \times n_c$. The relative position of these blocks within the global matrix, Figure 1(b), is determined by the numbering of the mesh faces, Figure 1(a).

- The maximum number of nonzero dense blocks in each row, referred as n_a, is fixed and always equal to five for triangular meshes, and seven for tetrahedral meshes, see Figures 1(a) and 1(b). That is, for the block row associated to a face i_f, there is a nonzero block for each face that shares an element with i_f in addition to the diagonal block accounting for the face self influence. We denote by n_{af} the number of mesh faces that share an element with the face i_f. The number n_{af} is equal to n_a for the inner faces and is less than n_a for the boundary faces.

To take advantage of the structural properties of the matrix $\mathbf{A}$, we consider a dense block format as proposed in Ref. 15. To this end, given a system matrix $\mathbf{A}$, we define the entries of the dense block format multi-array $\mathcal{A}$, of dimensions $(n_p n_c) \times n_f \times (n_p n_c n_a)$, by

$$\mathcal{A}_{i_p i_c i_f k_p k_c k_a} := \begin{cases} \mathbf{A}_{i_p i_c i_f k_p k_c k_f(k_a, i_f)} & \text{if } k_a \le n_{af}, \\ 0 & \text{if } k_a > n_{af}, \end{cases}$$

where $k_f(k_a, i_f)$ denotes the k_a ($1 \leq k_a \leq n_a$) mesh face that shares and element with i_f. Figure 1(c) shows the dense block format for a mesh with six faces. Equivalently, given a vector of unknowns $\hat{\mathbf{u}}$ we define the entries of the (inflated) dense block format multi-array $\hat{\mathcal{U}}$, of dimensions $(n_p n_c n_a) \times n_f$, by

$$\hat{\mathcal{U}}_{k_p k_c k_a i_f} := \begin{cases} \hat{\mathbf{u}}_{k_p k_c k_f(k_a, i_f)} & \text{if } k_a \leq n_{af}, \\ 0 & \text{if } k_a > n_{af}. \end{cases}$$

Using these dense block formats, the global system is equivalent to

$$\sum_{k_a=1}^{n_a} \sum_{k_c=1}^{n_c} \sum_{k_p=1}^{n_p} \mathcal{A}_{i_p i_c i_f k_p k_c k_a} \hat{\mathcal{U}}_{k_p k_c k_a i_f} = \mathbf{b}_{i_p i_c i_f},$$

for $i_f = 1, \ldots, n_f$, $i_c = 1, \ldots, n_c$ and $i_p = 1, \ldots, n_p$. If we now map the multi-indices (i_p, i_c), (i_f) and (k_p, k_c, k_a) to i, l and k, as follows

$$\begin{aligned} i = i(i_p, i_c) &:= i_p + n_p(i_c - 1), \\ l = l(i_f) &:= i_f, \\ k = k(k_p, k_c, k_a) &:= k_p + n_p(k_c - 1) + n_p n_c(k_a - 1), \end{aligned}$$

we can rewrite the global system as

$$\sum_{k=1}^{q} \mathcal{A}_{ilk} \hat{\mathcal{U}}_{kl} = \mathbf{b}_{il},$$

for $i = 1, \ldots, m$, $l = 1, \ldots, r$, and $k = 1, \ldots, q$, where $m = n_p n_c$, $r = n_f$ and $q = n_p n_c n_a$. We shall refer to $\mathcal{A}_{m \times r \times q} = [\mathcal{A}_{ilk}] = \left[\mathcal{A}_{i_p i_c i_f k_p k_c k_a}\right]$ and $\hat{\mathcal{U}}_{q \times r} = \left[\hat{\mathcal{U}}_{kl}\right] = \left[\hat{\mathcal{U}}_{k_p k_c k_a i_f}\right]$ as the *dense block format* (DBF) of $\mathbf{A}$ and $\hat{\mathbf{u}}$, respectively.

IV. Sparse matrix-vector product on the GPU

To perform the matrix-vector products required to solve the system $\mathbf{A}\hat{\mathbf{u}} = \mathbf{b}$ stored in dense block format, we propose to implement the following tensor contraction:

$$\mathbf{y}_{il} = [\mathbf{A}\mathbf{x}]_{il} = \sum_{k=1}^{q} \mathcal{A}_{ilk} \mathcal{X}_{kl},$$

for $i = 1, \ldots, m$ and $l = 1, \ldots, r$, where $\mathcal{A}$ and $\mathcal{X}$ are in dense block format. Several general-purpose matrix-vector product algorithms for GPUs have been proposed for dense[18–20] and sparse[21,22,25–29] matrices. Here, we describe a specific algorithm for the matrix-vector product of HDG matrices, which is more efficient than other general-purpose algorithms.

The straightforward implementation of the proposed algorithm on the CPU when $\mathcal{A}$ and $\mathcal{X}$ are stored in the host memory in dense block format is given by Algorithm 1.

Algorithm 1: Matrix-vector multiplication for block dense format (CPU)

Input: $\mathcal{A}_{m \times r \times q}$, dense block format of $\mathbf{A}$ in the host memory
Input: $\mathcal{X}_{q \times r}$, dense block format of $\mathbf{x}$ stored in the host memory
Output: $\mathbf{y}_{m \times r} = \mathbf{A}\mathbf{x}$

```
for l = 1 : r do
    for i = 1 : m do
        for k = 1 : q do
            y_il ← y_il + A_ilk X_kl        // Compute Σ_{k=1}^{q} A_ilk X_kl for fixed i and l
```

The GPU version of the proposed method is given by Algorithm 2. This algorithm assumes that the multi-arrays $\mathcal{A}^{\texttt{GPU}}$ and $\mathcal{X}^{\texttt{GPU}}$ are already stored in the device memory. Also, the array $\mathcal{X}^{\texttt{GPU}}$ is bound to

Algorithm 2: Matrix-vector multiplication for block dense format (GPU)

Input: $\mathcal{A}^{\texttt{GPU}}_{m\times r\times q}$, dense block format of $\mathbf{A}$ stored in the GPU memory
Input: $\mathcal{X}^{\texttt{GPU}}_{q\times r}$, dense block format of $\mathbf{x}$ stored in the GPU memory and bound to texture memory
Input: n_t, number of threads per block
Output: $\mathbf{y}^{\texttt{GPU}}_{m\times r} = \mathbf{Ax}$, stored in the GPU memory

```
1  b ← blockId.x
2  t_b ← threadId.x
3  t ← b × n_t + t_b
4  i ← t (mod m) + 1
5  l ← t/m + 1
6  if l ≤ r then
7      y ← 0
8      for k = 1 : q do
9          x ← read X^GPU_kl from texture memory
10         y ← y + A^GPU_ilk x        // Compute Σ_{k=1}^{q} A^GPU_ilk X^GPU_kl for fixed i and l
11     y^GPU_il ← y
```

the texture memory. This binding allows to use the cache of the texture memory to increase the re-use of $\mathcal{X}^{\texttt{GPU}}$. Before the computation starts, the total number of execution threads needs to be determined. We adopt a standard one thread-per-output approach so that each thread is responsible for the computation of one value of the output result $\mathbf{y}^{\texttt{GPU}}_{il}$. In principle, we would only need to launch mr threads. However, the CUDA framework requires that threads be launched in blocks of the same size n_t. Thus, we have to ensure that we launch enough thread blocks to have a thread for each output value $\mathbf{y}^{\texttt{GPU}}_{il}$. To this end, we launch $(mr + n_t - 1)/n_t$ thread blocks with n_t threads in each block. This means that, in general, we will launch $n_t - mr \pmod{n_t}$ more threads that required. It is easy to ensure that these additional threads do not perform any computation.

Once the blocks and threads are determined, we can run the same code in parallel on the GPU. Since the same code is going to be executed in all threads, we need to determine the current block (Line 1) and local thread (Line 2) IDs. These information, allow us to obtain a global thread ID t (Line 3) that is required to access the appropriate data. That is, the values of i and l (Lines 4 and 5) that we compute uniquely map each thread t to an output value $\mathbf{y}^{\texttt{GPU}}_{il}$. This mapping together with column-major storage ensures that the threads of the same block write on stride one contiguous memory addresses of the output array $\mathbf{y}^{\texttt{GPU}}_{il}$. The check $l \leq r$ (Line 6) guarantees that only those threads with an associated output value $\mathbf{y}^{\texttt{GPU}}_{il}$ carry out the multiplication. Each computing thread reads $\mathcal{X}^{\texttt{GPU}}_{kl}$ (Line 9) and $\mathcal{A}^{\texttt{GPU}}_{ilk}$ (Line 10) for consecutive values of the index k, and accumulates the result of their product in y (Line 10). The accumulated value y is then assigned to the corresponding output value $\mathbf{y}^{\texttt{GPU}}_{il}$ (Line 11).

Note that the arrays $\mathcal{A}^{\texttt{GPU}}_{ilk}$, $\mathcal{X}^{\texttt{GPU}}_{kl}$, and $\mathbf{y}^{\texttt{GPU}}_{il}$ are stored in GPU memory in column-major format. That is, the left index is for the rows, the middle index for the columns and the right index for the pages. Thus, for a fixed k, the proposed algorithm allows for several threads in the same block to read and write at the same time (*coalescing*) several contiguous memory values of $\mathcal{A}^{\texttt{GPU}}$ and $\mathbf{y}^{\texttt{GPU}}_{il}$, respectively. Since each value of $\mathcal{A}^{\texttt{GPU}}$ and $\mathbf{y}^{\texttt{GPU}}_{il}$ have to be read (Line 10) and written (Line 11) from the device memory, it is important to use the coalescing features of the GPU to ensure performance. Additional efficiency is obtained by promoting re-use of $\mathcal{X}^{\texttt{GPU}}_{kl}$ within each thread block through the use of the cache in the texture memory (Line 9).

V. Results

In this section, we analyze the performance of the proposed method. We first profile and compare the CPU and GPU implementations of the tensor contraction, the core of our specific sparse matrix-vector product. Then, we compare the execution time on the GPU of the proposed sparse matrix-vector product with the general-purpose sparse matrix-vector products of MATLAB (CPU) and the CUSPARSE (GPU) library.

For all the tests, we use a Mac Pro (2009) running Mac OS X 10.6.3 with two Quad-Core Intel Xeon (CPU) at 2.93 GHz, 24 GBytes of host memory (CPU), and one NVIDIA GeForce GTX 285 (GPU) with 1 GByte of device memory. The GPU is connected to the host through a PCIE 2.0 16X bus and we use CUDA 3.2 API to program the device code. For the host code, we use the GNU C++ compiler (g++) with flags `-O3 -msse3` (full optimization and SSE3 instructions). While for the GPU code, we use the CUDA compiler (nvcc) with flags `-O3 -arch=sm_13` (full optimization and double precision support).

V.A. Performance of the tensor contraction

To determine the performance of the tensor contraction operation on the CPU and on the GPU, we measure the computational speed in GFLOP/s and the effective memory bandwidth as a percentage of the platform (CPU or GPU) peak memory bandwidth (GBytes/s). In addition, we measure the speedup provided by the GPU using the CPU implementation as the reference. To obtain reliable measurements, we perform several tensor contractions

$$\mathbf{y}_{il} = \sum_{k=1}^{q} \mathcal{A}_{ilk} \mathcal{X}_{kl},$$

where $\mathbf{y}_{il}$, $\mathcal{A}_{ilk}$, and $\mathcal{X}_{kl}$ have sizes $m \times r$, $m \times r \times q$, and $q \times r$ determined by varying the solution interpolation degree in the HDG method $p = 1, \ldots, 10$, the number of spatial dimensions $d = 2, 3$ and the amount of used memory (MBytes). Thus, we have $m = n_p n_c$, $r = n_f$, and $q = n_p n_c n_a$, where n_p is the number of nodes per face, n_c is the number of solution components, n_f is the number of mesh faces and n_a is the maximum number of coupled faces n_a. In the 2D case, $n_p = (p+1)$, $n_c = 4$ and $n_a = 5$, whereas in the 3D case $n_p = (p+1)(p+2)/2$, $n_c = 5$ and $n_a = 7$. The results are presented as a function of the total memory (Mbytes) that, for a given problem, is determined by the number of faces. Note that the amount of memory in double precision (8 bytes per real value) is twice the amount of memory in single precision (4 bytes per real value). The total memory usage considered in our tests ranges from 5 MBytes (10 MBytes) to 250 MBytes (500 MBytes) for single (double) precision. To simplify the plots, we do not include a curve for each interpolation degree p. Specifically, for a fixed amount of memory, we show only the minimum and maximum measured values for all the p. This means that the results obtained for all values of p are between the two curves shown.

In the presented results, the computational performance in GFLOP/s is calculated as the total number of GFLOP which is calculated as,

$$2 \times m \times r \times q / 10^9 = 2 \times (n_p n_c) \times n_f \times (n_p n_c n_a) / 10^9,$$

divided by the execution time in seconds. The effective memory bandwidth in GBytes/s is obtained as the amount of GBytes divided by the execution time in seconds. Specifically, the number of GBytes required for $\mathbf{y}_{il}$, $\mathcal{A}_{ilk}$ and $\mathcal{X}_{kl}$ is given by,

$$m \times r + m \times r \times q + q \times r = (n_p n_c) \times n_f + (n_p n_c) \times n_f \times (n_p n_c n_a) + (n_p n_c n_a) \times n_f,$$

times $4/2^{30}$ ($8/2^{30}$) for single (double) precision. Combining the above expressions, we can calculate a ratio of memory to floating point operations,

$$(m \times r + m \times r \times q + q \times r)/(2 \times m \times r \times q) = 1/(2q) + 1/2 + 1/(2m),$$

which is always greater than $1/2$ and hence bounded from below independently of the problem size. This shows that the tensor contraction operation considered is a memory demanding operation and we expect the performance to be bounded by the memory bandwidth of the device. This observation is important to understand the behavior of the implemented tensor contraction on the CPU and the GPU.

In Figure 2, we present the computational speed (GFLOP/s) as a function of the amount of used memory (MBytes) for single and double precision tensor contractions performed on the CPU and the GPU. As mentioned above, the operation considered is a memory-bound operation in which the amount of read and written memory is twice bigger in double precision than in single precision. This is reflected in the results which clearly show a higher computational performance for single precision than for double precision both in the CPU and the GPU. For big enough problems, the single-threaded implementation on the CPU performs between 0.25 and 1.75 GFLOP/s for single precision, and between 0.2 and 1.25 GFLOP/s for double precision.

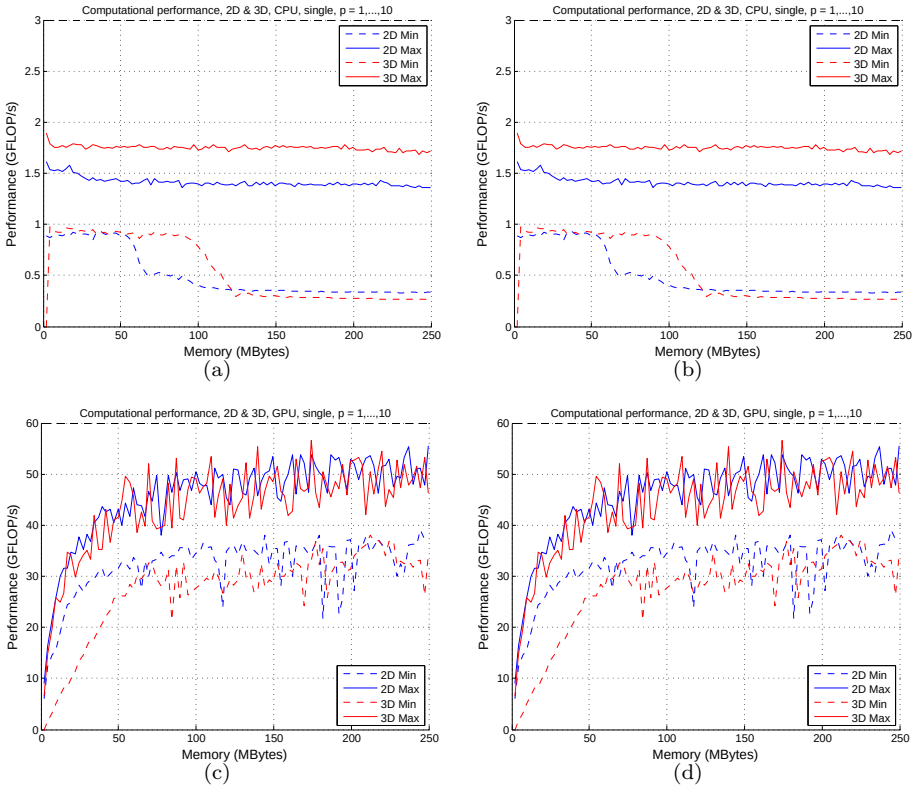


Figure 2. Minimum and maximum computational performance (GFLOP/s) as a function of the required memory (MBytes) for 2D and 3D problems and $p = 1, \ldots, 10$: on the CPU for (a) single and (b) double precision; and on the GPU for (c) single and (d) double precision.

The GPU implementation performs between 30 and 55 GFLOP/s for single precision, and between 20 and 25 GFLOP/s for double precision.

Although the effective memory bandwidth is not a direct computational performance metric, it shows how effective is the memory access and how close to peak performance is the current algorithm implementation. In Figure 3, we present the effective memory bandwidth as a percentage of the platform (CPU and GPU) peak memory bandwidth (16 GBytes/s and 159.7 GBytes/s) as a function of the amount of used memory (MBytes), for single and double precision. For big enough problems, the CPU implementation obtains an efficiency between 2.5% and 20% of the peak memory bandwidth in single precision, and between 5% and 30% for double precision. The GPU implementation obtains an efficiency between 30% and 65% of the peak memory bandwidth in single precision, and between 40% and 65% for double precision.

The results presented in Figs. 2 and 3 show a significant difference between the minimum and maximum performance curves on the CPU. While the maximum performance curves are approximately constant, the minimum performance curves show that the performance suddenly drops at a specific value of the used memory. The minimum performance curves correspond to the low order interpolation degrees, $p = 1, 2$, where each value $\mathcal{X}_{kl}$ is re-used for less values of $\mathcal{A}_{ilk}$. Thus, the fast drop of the minimum performance curves can be traced to the low re-use of $\mathcal{X}_{kl}$ for low p together with the fact that $\mathcal{X}_{kl}$ does not fit in the cache memory for a big enough number of faces n_f. As the results show, the performance drop appears later in the 3D case than in the 2D case because the amount of memory to store the values $\mathcal{A}_{ilk}$ is much bigger in 3D than in 2D.

On the other hand, the results presented in Figs. 2 and 3 show a similar behavior for the minimum and maximum performance curves on the GPU. The curves show a global and a local behavior. Globally, they grow with the amount of used memory, and locally, they oscillate. The global behavior can be easily understood. For a small number of faces (low memory usage) the number of threads used ($n_p \times n_c \times n_f$) is low. This leads to low hardware occupancy and therefore, to low performance. The local behavior is due to memory alignment issues. The proposed method ensures that, for a fixed value of k, contiguous threads read values of $\mathcal{A}_{ilk}$ in contiguous memory addresses. However, we cannot ensure that the first thread of

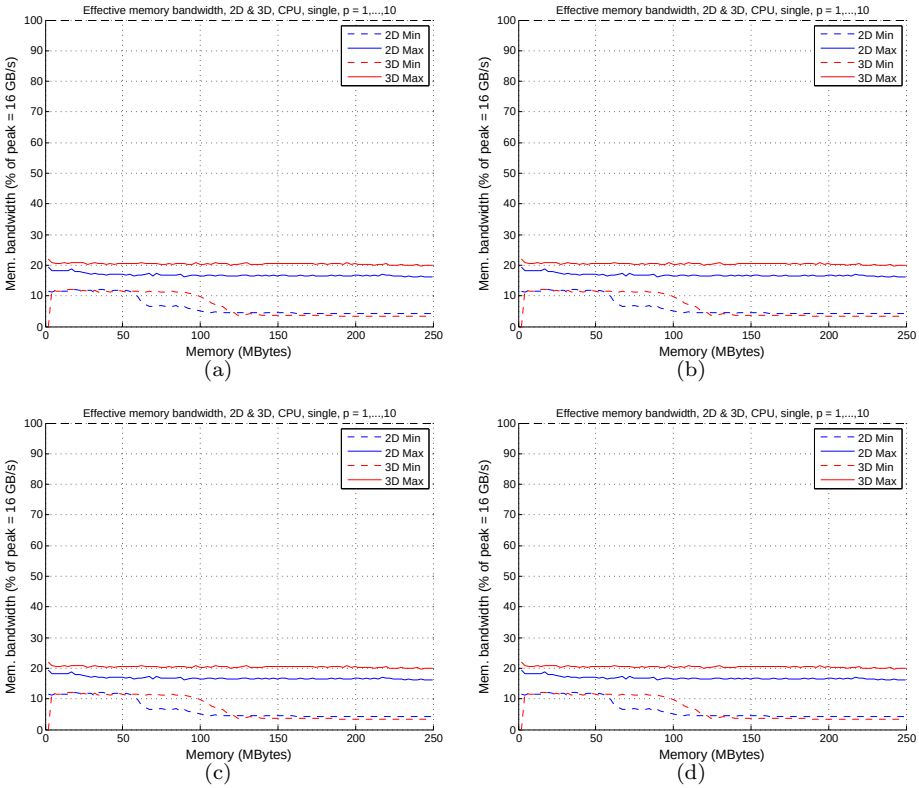


Figure 3. Minimum and maximum effective memory bandwidth (GBytes/s) as a function of the required memory (MBytes) for 2D and 3D problems and $p = 1, \ldots, 10$: on the CPU for (a) single and (b) double precision; and on the GPU for (c) single and (d) double precision.

each block reads a value of $\mathcal{A}_{ilk}$ which is stored in a memory address that is a multiple of the *half-warp* (one half of the number of hardware threads per block). Note that the maximum memory performance (*full coalescing*) can only be obtained when the threads read contiguous memory and the first thread of each block reads a memory address that is multiple of half-warp (16 for the GTX 285). Nevertheless, the tests on the GPU show that stride one and contiguous memory reads of $\mathcal{A}_{ilk}$ run at a significant 60% of the peak memory bandwidth. We also note that part of this memory bandwidth is used to read the values $\mathcal{X}_{kl}$. This cost of reading $\mathcal{X}_{kl}$ is reduced automatically in the CPU by the cache memory and manually in the GPU by binding the multi-array $\mathcal{X}_{kl}$ to the cache of the texture memory. Finally, the rest of the bandwidth is used to write the values of $\mathbf{y}_{il}$. This operation is efficient both in the CPU and the GPU because it is performed with stride one and contiguous aligned memory accesses. In particular, the values of $\mathbf{y}_{il}$ are written with full coalescing on the GPU.

Since proposed contraction is a memory-bound operation, we would expect that it would perform about ten times faster in the tested GPU (159.7 GBytes/s) than in the tested CPU (16 GBytes/s). To verify this point, we present in Figure 4, the minimum and maximum speedup curves (CPU time/GPU time) for single and double precision. As we mentioned above, the CPU implementation presents a very low performance for low $p = 1, 2$. Consequently, the speedup for these cases is very large. Even if we ignore maximum speedup curves and consider only the minimum speedup curves, the results show that the GPU is more than 20 times faster than the CPU, for single and double precision. This is two times bigger than the ratio of the peak memory bandwidths. Therefore, we would expect that the CPU maximum performance could be improved by optimizing the memory access.

V.B. Comparison with MATLAB and CUSPARSE library

The main purpose of is this section is to show that the proposed GPU-accelerated sparse-matrix vector product for HDG is more efficient than general-purpose sparse-matrix vector products for both the CPU and the GPU. The price for this higher performance is that the proposed method can only be applied to

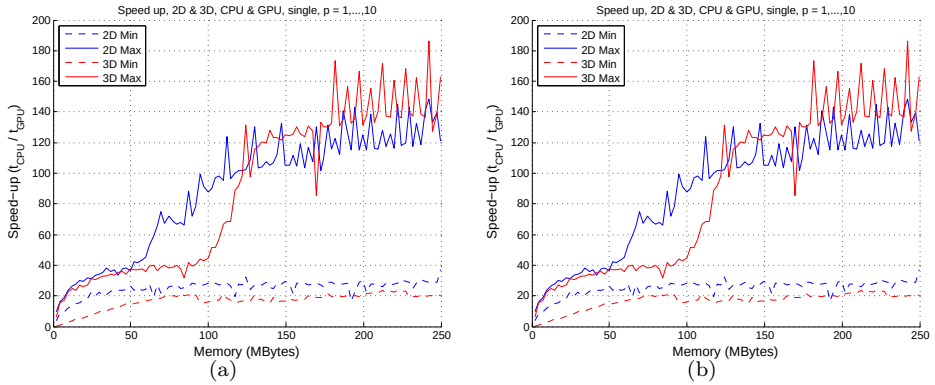


Figure 4. Minimum and maximum speedup (CPU time / GPU time) as function of required memory (MBytes) for 2D and 3D problems, $p = 1, \ldots, 10$: (a) single and (b) double precision.

p	$n_p n_c$	$n_p n_c n_f$	CSR (nnz)	DBF (nnz)	nnz ratio	CSR (MB)	DBF (MB)	Memory ratio
1	8	94240	3743188	3769600	1.0071	44.63	33.29	0.75
2	12	141360	8423010	8481600	1.0070	99.09	71.40	0.72
3	16	188480	14975728	15078400	1.0069	174.98	123.89	0.71
4	20	235600	23395950	23560000	1.0070	272.24	190.75	0.70
5	24	282720	33696648	33926400	1.0068	391.02	272.00	0.70

Table 1. Properties of the five tested sparse matrix-vector products for $p = 1, \ldots, 5$ corresponding to a triangular mesh with 11780 faces: number of unkowns per face ($n_p n_c$), total number of unkowns ($n_p n_c n_f$), number of non-zero values for CSR (nnz) and DBF (nnz), ratio of non-zero values (DBF / CSR), memory (MBytes) required to perform the operation for CSR and DBF, ratio of required memory (DBF / CSR).

matrices which have the specific structure considered here. We compare the proposed algorithm with the CUSPARSE library running on the GPU, and sparse-matrix vector product of MATLAB running on the CPU.

To perform the comparison, we consider five HDG matrices corresponding to the same steady-state compressible Navier-Stokes problem on the same triangular mesh (11780 faces) for different interpolation degrees, $p = 1, \ldots, 5$. Table 1 shows for each matrix the number of faces and global unknowns, the number of stored non-zero values for the compressed sparse row (CSR) format and the dense block format (DBF), and the amount of memory required to perform the sparse matrix-vector product for the CSR and the DBF formats (input vector, matrix, output vector and sparse indirection vectors). As expected, the number of non-zero values stored with the dense block format is slightly bigger than for the CSR format. This additional storage is required because in the dense format we do not take into account the fact that the boundary faces are only connected to three faces rather than five. Nevertheless, the DBF requires only 3/4 of the memory used with the CSR format. Memory efficiency is a clear advantage of the proposed specific-purpose storage format.

The proposed sparse-matrix vector product is performed in two steps, first the input vector is converted to dense block format, and then the tensor contraction is computed. Table 2 summarizes the amount of time spent on each procedure for the five test matrices. The results show that the tensor contraction is the most expensive procedure and its time percentage grows with the interpolation degree. This is the expected behavior, for different p the size of the matrix grows with n_p^2 while the size of the vector grows with n_p.

VI. Conclusion

We have presented a sparse-matrix vector product for the matrices arising from HDG discretizations. These matrices have a sparse structure with equal-sized, dense, non-overlapping blocks. These sparse matrices are stored as a dense 3-dimensional array. Using this storage format, the sparse-matrix vector product can be cast as dense tensor contraction. The performance of the resulting tensor contraction is determined by the peak memory bandwidth of the selected platform. The dense tensor contraction operation has been

p	t_v(ms)	t_c(ms)	t_v+t_c (ms)	t_v (%)	t_c (%)
1	0.27	0.44	0.72	37.90%	62.10%
2	0.39	0.77	1.15	33.45%	66.55%
3	0.50	1.28	1.77	27.92%	72.08%
4	0.60	1.93	2.53	23.71%	76.29%
5	0.71	2.70	3.41	20.80%	79.20%

Table 2. Sparse matrix-vector product as a tensor contraction on the GPU for $p = 1, \ldots, 5$: time (ms) to convert the input vector to dense block format (t_v), time (ms) to perform the tensor contraction (t_c), total time (ms) (t_v+t_c), time percentage to convert the input vector to dense block format, and time percentage to perform the tensor contraction.

	Execution time (ms)			Speedup (MATLAB as reference)		
p	MATLAB	CUSPARSE	Proposed	MATLAB	CUSPARSE	Proposed
1	22.81	1.87	0.72	1.00	12.22	31.91
2	47.07	3.05	1.15	1.00	15.41	40.79
3	77.37	4.17	1.77	1.00	18.54	43.64
4	118.51	5.91	2.53	1.00	20.05	46.82
5	179.37	7.39	3.41	1.00	24.28	52.54

Table 3. Speedup of the proposed and CUSPARSE sparse matrix-vector product compared with MATLAB for $p = 1, \ldots, 5$: MATLAB sparse matrix-vector product on the CPU, CUSPARSE matrix-vector product on the GPU, proposed sparse matrix-vector product on the GPU, speedup taking MATLAB as the reference speed for MATLAB, CUSPARSE, and the proposed method.

implemented on the GPU. The selected GPU (GEFORCE GTX 285) has a peak memory bandwidth ten times greater than the one of the host computer (Mac Pro 2009). The proposed tensor contraction exploits the characteristics of the HDG matrices to obtain full benefit of the GPU: it performs partial coalesced reads of the values of the HDG matrix, it uses the cache of the texture memory to read the unordered values of the input vector, and performs full coalesced writes of the output values. As the double precision tests show, the obtained tensor contraction performs at 20 to 25 GFLOP/s with a sustained effective memory bandwidth greater than the 40% of the peak memory bandwidth. This results in a specific sparse-matrix vector product for HDG that performs 2 times faster than the CUSPARSE library on the GPU and 30 times faster than MATLAB on the CPU. It is important to point out that the sparse-matrix vector product proposed can be applied to other discretization methods with similar sparse matrix structure. In particular, it is straightforward to apply the proposed algorithm to matrices derived from other DG discretizations.

Acknowledgments

X. Roca, N. C. Nguyen, and J. Peraire would like to acknowledge the Singapore-MIT Alliance and the Air Force Office of Scientific Research under the MURI project on Biologically Inspired Flight for partially supporting this work.

References

[1]Cockburn, B., Gopalakrishnan, J., and Lazarov, R., "Unified hybridization of discontinuous Galerkin, mixed and continuous Galerkin methods for second order elliptic problems," *SIAM J. Numer. Anal.*, Vol. 47, 2009, pp. 1319–1365.

[2]Cockburn, B., Dong, B., and Guzmán, J., "A superconvergent LDG-hybridizable Galerkin method for A superconvergent LDG-hybridizable Galerkin method for second-order elliptic problems," *Math. Comp.*, Vol. 77, 2008, pp. 1887–1916.

[3]Cockburn, B., Dong, B., Guzmán, J., Restelli, M., and Sacco, R., "A Hybridizable Discontinuous Galerkin Method for Steady-State Convection-Diffusion-Reaction Problems," *SIAM J. Sci. Comput.*, Vol. 31, 2009, pp. 3827.

[4]Nguyen, N., Peraire, J., and Cockburn, B., "An implicit high-order hybridizable discontinuous Galerkin method for linear convection-diffusion equations," *J. Comput. Phys.*, Vol. 228, 2009, pp. 3232–3254.

[5]Nguyen, N., Peraire, J., and Cockburn, B., "An implicit high-order hybridizable discontinuous Galerkin method for nonlinear convection-diffusion equations," *J. Comput. Phys.*, Vol. 228, 2009, pp. 8841–8855.

[6]Cockburn, B., Gopalakrishnan, J., Nguyen, N., Peraire, J., and Sayas, F.-J., "Analysis of an HDG method for Stokes flow," *Math. Comp.*, To appear.

[7]Nguyen, N., Peraire, J., and Cockburn, B., "A hybridizable discontinuous Galerkin method for the incompressible Navier-

Stokes equations," *Proceedings of the 48th AIAA Aerospace Sciences Meeting and Exhibit*, 2010, pp. AIAA Paper 2010–362.

[8]Nguyen, N., Peraire, J., and Cockburn, B., "Hybridizable discontinuous Galerkin methods," *Spectral and High Order Methods for Partial Differential Equations*, edited by J. S. Hesthaven and E. M. Ronquist, Vol. 76, 2011, pp. 63–84.

[9]Nguyen, N., Peraire, J., and Cockburn, B., "An implicit high-order hybridizable discontinuous Galerkin method for the incompressible Navier-Stokes equations," *J. Comput. Phys.*, In press.

[10]Nguyen, N., Peraire, J., and Cockburn, B., "A comparison of HDG methods for Stokes flow," *Journal of Scientific Computing*, Vol. 45, 20010, pp. 215–237.

[11]Peraire, J., Nguyen, N., and Cockburn, B., "A hybridizable discontinuous Galerkin method for the compressible Euler and Navier-Stokes equations," *Proceedings of the 48th AIAA Aerospace Sciences Meeting and Exhibit*, 2010, pp. AIAA Paper 2010–363.

[12]Bassi, F. and Rebay, S., "A high-order accurate discontinuous finite element method for the numerical solution of the compressible Navier-Stokes equations," *J. Comput. Phys.*, Vol. 131, No. 2, 1997, pp. 267–279.

[13]Peraire, J. and Persson, P., "The compact discontinuous Galerkin (CDG) method for elliptic problems," *SIAM J. Sci. Comput.*, Vol. 30, No. 4, 2008, pp. 1806–1824.

[14]Klöckner, A., Warburton, T., Bridge, J., and Hesthaven, J., "Nodal discontinuous Galerkin methods on graphics processors," *Journal of Computational Physics*, 2009.

[15]Persson, P. and Peraire, J., "Newton-GMRES preconditioning for discontinuous Galerkin discretizations of the Navier-Stokes equations," *SIAM J. Sci. Comput.*, Vol. 30, No. 6, 2008, pp. 2709–2733.

[16]Cockburn, B. and Shu, C.-W., "Runge-Kutta Discontinuous Galerkin Methods for Convection-Dominated Problems," *J. Sci. Comput.*, Vol. 16, No. 3, 2001, pp. 173–261.

[17]Fidkowski, K., Oliver, T., Lu, J., and Darmofal, D., "p-Multigrid solution of high-order discontinuous Galerkin discretizations of the compressible Navier-Stokes equations," *J. Comput. Phys.*, Vol. 207, 2005, pp. 92–113.

[18]Volkov, V. and Demmel, J., "Benchmarking GPUs to tune dense linear algebra," *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008, pp. 1–11.

[19]Tomov, S., Dongarra, J., and Baboulin, M., "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, 2009.

[20]Li, Y., Dongarra, J., and Tomov, S., "A Note on Auto-tuning GEMM for GPUs," *Computational Science–ICCS 2009*, 2009, pp. 884–892.

[21]Cevahir, A., Nukada, A., and Matsuoka, S., "Fast conjugate gradients with multiple GPUs," *Computational Science–ICCS 2009*, 2009, pp. 893–903.

[22]Buatois, L., Caumon, G., and Lévy, B., "Concurrent number cruncher: An efficient sparse linear solver on the GPU," *Lecture Notes in Computer Science*, Vol. 4782, 2007, pp. 358.

[23]Wang, M., Klie, H., Parashar, M., and Sudan, H., "Solving Sparse Linear Systems on NVIDIA Tesla GPUs," *Computational Science–ICCS 2009*, 2009, pp. 864–873.

[24]Saad, Y. and Schultz, M., "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems." *SIAM J. Sci. Stat. Comput.*, Vol. 7, No. 3, 1986, pp. 856–869.

[25]Bell, N. and Garland, M., "Implementing sparse matrix-vector multiplication on throughput-oriented processors," *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ACM, 2009, pp. 1–11.

[26]Baskaran, M. and Bordawekar, R., "Optimizing sparse matrix-vector multiplication on GPUs," Tech. rep., IBM Research Report RC24704, 2009.

[27]Li, R. and Saad, Y., "GPU-Accelerated Preconditioned Iterative Linear Solvers," Tech. rep., Department of Computer Science & Engineering, University of Minnesota.

[28]Monakov, A. and Avetisyan, A., "Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs," *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2009, pp. 289–297.

[29]Dehnavi, M., Fernandez, D., and Giannacopoulos, D., "Finite-Element Sparse Matrix Vector Multiplication on Graphic Processing Units," *Magnetics, IEEE Transactions on*, Vol. 46, No. 8, 2010, pp. 2982–2985.

[30]Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J., "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, Vol. 35, No. 3, 2009, pp. 178–194.

[31]Anderson, D., Tannehill, J., and Pletcher, R., *Computational fluid mechanics and heat transfer*, Hemisphere Publishing, New York, NY, 1984.