Noname manuscript No.
(will be inserted by the editor)

Nekbone Performance on GPUs with OpenACC and CUDA Fortran Implementations

Jing Gong · Stefano Markidis · Erwin Laure · Matthew Otten · Paul Fischer · Misun Min

Received: date / Accepted: date

Abstract We present a hybrid GPU implementation and performance analysis of Nekbone, which represents one of the core kernels of the incompressible Navier-Stokes solver Nek5000. The implementation is based on OpenACC and CUDA Fortran for local parallelization of the compute-intensive matrix-matrix multiplication part, which significantly minimizes the modification of the existing CPU code while extending the simulation capability of the code to GPU architectures. Our discussion includes the GPU results of OpenACC interoperating with CUDA Fortran and the gather-scatter operations with GPUDirect communication. We demonstrate performance of up to 552 Tflops on 16,384 GPUs of the OLCF Cray XK7 Titan.

Keywords Nekbone/Nek5000 · OpenACC · CUDA Fortran · GPUDirect · Gather-scatter communication · Spectral-element discretization

1 Introduction

OpenACC is a high-level compiler-directive-based programming approach for parallel computing on graphics processing units (GPUs). The compiler maps the compute and data regions specified by the OpenACC directives to GPUs for higher performance. In contrast to other low-level GPU programming, such as CUDA and OpenCL, where more explicit compute and data management is necessary, porting of legacy CPU-based applications with OpenACC does not require significant structural changes in the original code, which allows

COST IC1305, corresponding author: Misun Min, E-mail: mmin@mcs.anl.gov

J. Gong, S. Markidis, E. Laure, PDC, KTH, E-mail: (gongjing,markidis,erwinl)@pdc.kth.se
M. Otten, Cornell University, U.S.A., E-mail: mjo98@cornell.edu
P. Fischer, Univeristy of Illinois Urbana-Champaign, U.S.A., E-mail: fischerp@illinois.edu and Argonne National Laboratory, U.S.A., E-mail: fischer@mcs.anl.gov
M. Min, Argonne National Laboratory, U.S.A., E-mail: mmin@mcs.anl.gov

considerable simplification and productivity improvement when hybridizing existing applications.

While the directive-based approach of OpenACC greatly simplifies programming it does not provide the flexibility of CUDA or OpenCL. For example, both CUDA and OpenCL provide fine-grained synchronization primitives, such as thread synchronization and atomic operations, whereas OpenACC does not. Efficient implementations of applications may depend on the availability of software-managed on-chip memory, which can be used directly in CUDA and OpenCL, but not in OpenACC. These differences may prevent full use of the available architectural resources, potentially resulting in inferior performance when compared to highly tuned CUDA and OpenCL code. However, OpenACC has significant potential to enable domain scientists to approach GPU computing with minimal effort and few modifications to existing CPU implementations.

To understand the performance implications for programming accelerators with OpenACC, this paper presents case studies of porting and optimization of kernel benchmarks for a spectral element code Nekbone, which is a simplified version of a computational fluid dynamics (CFD) code Nek5000. Nekbone focuses on the Poisson operator evaluation that is a central computational kernel in Nek5000. The discretization is based on a continuous Galerkin formulation using a high-order spectral element discretization on a mesh of hexahedral elements. The Poisson solver uses Jacobi preconditioned conjugate gradients which exercise two of the principal communication kernels in Nek5000—the gather-scatter (*gs*) operation for nearest-neighbor exchanges and the all-reduce operations that yield optimal approximations in the Krylov subspace. As kernel benchmarks, we focus on highly tuned OpenACC and CUDA Fortran versions for fine-grained parallelism of matrix-vector multiplications. To port Nekbone to GPU with OpenACC, we have added approximately 300 lines of OpenACC directives to the CPU version of Nekbone consisting of over 38,000 lines of FORTRAN routines. A CUDA Fortran implementation using OpenACC as an interface is also considered for the most compute-intensive matrix-matrix multiplication routines, which significantly simplifies the CUDA-based data management and copy. We note that, before starting the iterations in the Poisson solver, each CPU moves the necessary data from the host CPU memory to GPU memory only once, and computations are performed fully on GPUs during the iterations, which minimizes the data movement between CPU and GPU at each iteration and reduces the memory latency. As for the communication kernel of Nekbone, we examine our fully optimized GPU-based *gs* library which was demonstrated for a spectral element discontinuous Galerkin solver in our previous work [8] where an OpenACC implementation for the gather-scatter kernel consisting of over 2000 lines of C routines in the CPU version required adding approximately 23 lines of OpenACC directives.

The paper is organized as follows. We give an overview of Nekbone code and discuss in details for the governing equation, formulations and discretizations. We discuss highly optimized OpenACC and CUDA Fortran routines for

matrix-matrix multiplications, and the gather-scatter kernels. Then we provide performance results and their analysis on a single and multi GPUs.

2 Related Work

Here we address some related studies on performance, portability, implementation effort, and issues discussed by others for GPU-enabled computational fluid dynamic (CFD) codes based on OpenACC and CUDA accelerations.

In [1], single GPU performance is discussed for a CFD code, OVERFLOW. A CPU version of OVERFLOW is hand-translated into a CUDA version, achieving 40% speedup in total wall-clock time on a single GPU.

In [2], performance comparisons of OpenACC and CUDA on a single node of the Japanese supercomputer TSUBAME2.0 are studied for two microbenchmarks and a CFD application code, UPACS (Unified Platform for Aerospace Computational Simulation). The comparison indicates the limitation of programmable control of the on-chip memory in OpenACC by showing that some case studies of an OpenACC-based acceleration achieve approximately 50% of the performance of a CUDA-based acceleration, while the OpenACC version can increase its performance up to 98% of CUDA when a careful manual optimization is applied.

In [3], a C++ CFD code ZFS (Zonal Flow Solver) is OpenACC-accelerated and tested on an 8-node cluster (2 Intel Xeon E5-260 v2 and 4 Tesla K40m on each node) connected with FDR InfiniBand. Their OpenACC-accelerated ZFS demonstrates 2.1∼2.4× speedup compared with the CPU-only ZFS.

In [4], a cost-effective OpenACC implementation for a high-order implicit algorithm using a reconstructed discontinuous Galerkin method is discussed. The GPU-accelerated implementation includes a compact in-place direct inversion and a special-element-reordering algorithm, achieving 6∼7× speedup for transonic flow past a Boeing 747 aircraft compared with CPU runs. The authors leave the memory-bound issue of a fine-grained left-hand side for further improvement.

In [5], the progress and challenges in accelerating simulations of fluid flow using GPUs are surveyed. The authors present case studies and discuss successful strategies for performance improvement, including efficiently utilizing global and shared memory, asynchronous memory transfer, and minimal CPU-GPU communication.

3 Nekbone

Nekbone is a lightweight subset of the CFD code Nek5000 [9] that is intended to mimic the essential computational complexity of Nek5000 in relatively few lines of code. This “mini-app” allows software and hardware developers to understand the basic structure and computational costs of Nek5000 over a broad spectrum of architectures ranging from software-based simulators running at

one ten-thousandth the speed of current processors to exascale platforms running millions of times faster than single-core platforms. Indeed, Nekbone has weak-scaled to 6 million MPI ranks on the Blue Gene/Q Sequoia at Lawrence Livermore National Laboratory. (Nek5000 has strong-scaled to over a million ranks on the Blue Gene/Q Mira at Argonne National Laboratory.) Nekbone provides flexibility to adapt new programming approaches for scalability and performance studies on a variety of platforms without having to understand all the features of Nek5000.

Nek5000 is a high-order Navier-Stokes solver based on the spectral-element method (SEM) that is designed for direct numerical simulations (DNS) and large eddy simulations (LES) of turbulent flows in a broad range of application areas, including reactor thermal hydraulics, combustion, oceanography, and astrophysics, that require highly accurate solutions using extreme-scale computing resources. The code is MPI based, written in Fortran and C, and is equipped with highly tuned matrix-matrix product routines and a scalable gather-scatter kernel that has a black-box user interface. The SEM employs unstructured body-fitted hexahedral elements for accurate representations of complex geometries and employs efficient multilevel iterative solvers for implicit solution of the viscous and pressure substeps. As is invariably the case for incompressible Navier-Stokes solvers, the pressure solve accounts for most of the computational overhead in Nek5000—typically about 60 percent of the wall-clock time when using the spectral-element multigrid scheme developed in [10,11].

To capture the essential computational aspects of Nek5000, Nekbone solves the Poisson equation using conjugate gradient iteration with the same matrix-vector product routine as Nek5000 but on a domain that is a tensor product of rectilinear spectral elements. The simple geometry was chosen to facilitate scaling from one element to millions of elements at run time. No attempt is made to exploit this simple configuration, since that would inveigh against the generality of the target application space. Because Nekbone retains the essentials of the algorithmic and numerical approaches of Nek5000, including the general-purpose gather-scatter kernel, investigation and profiling of its performance can lend insight into the performance potential of Nek5000 on future architectures.

Nekbone is intrinsically well load-balanced, with each process having the same number of spectral elements and therefore the same amount of computational work. This strategy was followed in order to control macroscopic variables and to allow users to focus on more subtle performance features. The principal communication consists of nearest-neighbor (in the mesh topology, not necessarily in the network topology) exchanges with up to 26 surrounding elements, involving potential data communication for 6 faces, 12 edges, and 8 vertices. Because Nekbone generates a convex subset of elements for each rank, the 26 neighbor configuration also extends to the number of ranks that any single processor communicates with in the gather-scatter operation. In Nek5000, this number can be as high as 50 to 80 for irregular decompositions. Other communications in Nekbone include mpi_all_reduce. The tree-based coarse-

grid solve [12,13] featured in Nek5000 is not included in Nekbone. Because of boundary conditions, the amount of data communicated with neighboring processes can vary between processes, but the effects of this imbalance have been observed to be minimal on most systems. For systems with scalable (i.e., constant-time) MPI reduction operations and where point-to-point communications are noninterfering between network nodes, the performance of the benchmark has been observed to scale nearly linearly with increasing number of ranks across several orders of magnitude.

3.1 Formulation

The viscous and pressure substeps in Nek5000 require solutions of second-order variable-coefficient elliptic boundary value problems of the form

$$-\nabla \cdot (\alpha \nabla u) + \beta u = f \quad \text{in} \quad \Omega, \tag{1}$$

subject to Neumann, Dirichlet, and Robin conditions on the boundary $\partial\Omega$. Nekbone solves a restricted class of problems with constant coefficients ($\alpha = 1$ and $\beta \geq 0$), subject to homogeneous Dirichlet boundary conditions. The SEM is based on the weak formulation of (1): *Find a solution* $u \in X_0^N \subset H_0^1(\Omega)$ *such that*

$$a(u,v) + b(u,v) = (f,v) \qquad \text{for all } v \in X_0^N, \tag{2}$$

where the bilinear functionals are defined as

$$a(u,v) = \int_\Omega \nabla u \cdot \nabla v d\Omega = \int_\Omega \left(\frac{\partial u}{\partial x}\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\frac{\partial v}{\partial y} + \frac{\partial u}{\partial z}\frac{\partial v}{\partial z} \right) d\Omega, \tag{3}$$

$$b(u,v) = \int_\Omega uv \, d\Omega \quad \text{and} \quad (f,v) = \int_\Omega fv \, d\Omega. \tag{4}$$

Here, H_0^1 is the usual Sobolev space of square-integrable functions vanishing on $\partial\Omega$ and having derivatives that are also square-integrable, and X_0^N is a discrete subspace to be introduced shortly.

3.2 Discretizations

We denote our computational domain as $\Omega = \cup_{e=1}^E \Omega^e$, where Ω^e represents nonoverlapping body-conforming hexahedral elements. We define a finite-dimensional approximation space $V^N = \text{span}\{\psi_{ijk}(\xi^e, \eta^e, \gamma^e)\}$, with $i,j,k \in \{0,\ldots,N\}^3$ and $e \in \{1,\ldots,E\}$. Here, $(\xi^e, \eta^e, \gamma^e) \in \hat{\Omega} := [-1,1]^3$ are local computational coordinates associated with $\mathbf{x}^e = (x^e, y^e, z^e) \in \Omega^e$ that are defined implicitly by the isoparametric map

$$\mathbf{x}^e(\xi,\eta,\gamma) = \sum_{i,j,k=0}^{N} \mathbf{x}_{ijk}^e \psi_{ijk}(\xi,\eta,\gamma), \ \xi,\eta,\gamma \in \hat{\Omega}. \tag{5}$$

The local basis $\psi_{ijk}(\xi,\eta,\gamma) = \ell_i(\xi)\ell_j(\eta)\ell_k(\gamma)$, or simply ψ_{ijk}, is a tensor product of the one-dimensional Nth-order Legendre-Lagrange interpolation polynomials,

$$\ell_i(\xi) = [N(N+1)^{-1}(1-\xi^2)L'_N(\xi)]/[(\xi-\xi_i)L_N(\xi_i)] \quad \text{for } \xi \in [-1,1], \tag{6}$$

based on the Gauss-Lobatto-Legendre (GLL) quadrature nodes ξ_i with L_N denoting the Nth-order Legendre polynomial and L'_N its derivative.

For the SEM, we seek solutions $u(\mathbf{x}) \in X_0^N := V^N \cap H_0^1$. X_0^N is the subspace of the piecewise polynomial approximation space V^N that is continuous and that satisfies homogeneous Dirichlet conditions on $\partial\Omega$. As is the case with the geometry (5), u (and v) is expressed locally on $\hat{\Omega}$ by the tensor-product Lagrange polynomials,

$$u^e(\xi,\eta,\gamma) = \sum_{i,j,k=0}^{N} u^e_{ijk}\ell_i(\xi)\ell_j(\eta)\ell_k(\gamma). \tag{7}$$

The choice of this basis implies that the coefficients are nodal values at the GLL points. Specifically, $u(\mathbf{x}^e_{ijk}) = u^e_{ijk}$. Interelement function continuity for any function $u \in X^N$ is enforced by requiring that

$$u^e_{ijk} \;=\; u^{e'}_{i'j'k'} \;\text{ if }\; \mathbf{x}^e_{ijk} \;=\; \mathbf{x}^{e'}_{i'j'k'}. \tag{8}$$

In addition, boundary conditions ($u \in X_0^N$) are enforced by setting $u^e_{ijk} = 0$ for all $\mathbf{x}^e_{ijk} \in \partial\Omega$. The procedure for implementing these two conditions is described shortly.

All computations are performed on the reference domain. Denoting $(x,y,z) = (x_1,x_2,x_3)$ and $(\xi,\eta,\gamma) = (r_1,r_2,r_3)$, we evaluate derivatives in the bilinear form using the chain rule. For i=1, 2, and 3,

$$\frac{\partial u}{\partial x_i} = \left(\frac{\partial u}{\partial \xi}\frac{\partial \xi}{\partial x_i} + \frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial x_i} + \frac{\partial u}{\partial \gamma}\frac{\partial \gamma}{\partial x_i}\right) \;=\; \sum_{j=1}^{3}\frac{\partial u}{\partial r_j}\frac{\partial r_j}{\partial x_i}. \tag{9}$$

The bilinear form in (3) is expressed as

$$a(u,v) = \sum_{e=1}^{E}\sum_{j=1}^{3}\sum_{i=1}^{3}\int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1}\frac{\partial v}{\partial r_i}\mathcal{G}^{ij}\frac{\partial u}{\partial r_j}\,dr_1\,dr_2\,dr_3, \tag{10}$$

where the symmetric tensor comprising the geometric factors

$$\mathcal{G}^{ij} := \sum_{k=1}^{3}\frac{\partial r_i}{\partial x_k}\frac{\partial r_j}{\partial x_k}\,\mathcal{J}(r_1,r_2,r_3), \tag{11}$$

involves the metrics from the mapping $\mathbf{r}(\mathbf{x})$ and the Jacobian, $\mathcal{J} = \left|\frac{\partial \mathbf{x}}{\partial \mathbf{r}}\right|$.

A central step for efficient evaluation of the SEM operators is to approximate all integrals in the bilinear forms (3)–(4) by quadrature on the GLL nodal points [14]. If the local set of test and trial functions is represented by

the respective vectors $\underline{v}^e = \{v^e_{ijk}\}$ and $\underline{u}^e = \{u^e_{ijk}\}$, the discrete form for (3) reads

$$a_N(u,v) = \sum_{e=1}^{E} (\underline{v}^e)^T \begin{bmatrix} \mathbf{D}_\xi \\ \mathbf{D}_\eta \\ \mathbf{D}_\gamma \end{bmatrix}^T \begin{bmatrix} \mathbf{G}^{11} & \mathbf{G}^{12} & \mathbf{G}^{13} \\ \mathbf{G}^{21} & \mathbf{G}^{22} & \mathbf{G}^{23} \\ \mathbf{G}^{13} & \mathbf{G}^{23} & \mathbf{G}^{33} \end{bmatrix}^e \begin{bmatrix} \mathbf{D}_\xi \\ \mathbf{D}_\eta \\ \mathbf{D}_\gamma \end{bmatrix} \underline{u}^e \tag{12}$$

$$= \sum_{e=1}^{E} (\underline{v}^e)^T \mathbf{D}^T \mathbf{G}^e \mathbf{D} \underline{u}^e = \sum_{e=1}^{E} (\underline{v}^e)^T \mathbf{A}^e \underline{u}^e, \tag{13}$$

where $\mathbf{G}^{st} = \mathcal{G}^{st}_{ijk} \rho_i \rho_j \rho_k$ $(s,t = 1,2,3)$ represents the product of the geometric factors evaluated at the nodes with the corresponding GLL quadrature weights, $\rho_i \rho_j \rho_k$, and the matrices $\mathbf{D}_\xi$, $\mathbf{D}_\eta$, $\mathbf{D}_\gamma$ evaluate the local derivatives of their operands at the GLL points. Specifically, the tensor product forms

$$\mathbf{D}_\xi = \mathbf{I} \otimes \mathbf{I} \otimes \hat{\mathbf{D}}, \quad \mathbf{D}_\eta = \mathbf{I} \otimes \hat{\mathbf{D}} \otimes \mathbf{I}, \quad \mathbf{D}_\gamma = \hat{\mathbf{D}} \otimes \mathbf{I} \otimes \mathbf{I} \tag{14}$$

involve the one-dimensional differentiation matrix $\hat{\mathbf{D}}_{ki} = l_i'(\xi_k)$ and the identity matrix $\mathbf{I}$ in $\mathbb{R}^{(N+1)\times(N+1)}$.

To complete the problem statement, we need to assemble the system and apply the boundary conditions, both of which imply restrictions on the nodal values u^e_{ijk} and v^e_{ijk}. For any $u(\mathbf{x}) \in X^N$ we can associate a single nodal value u_g for each unique $\mathbf{x}_g \in \Omega$, where $g \in \{1,\ldots,\bar{n}\}$ and $\bar{n}$ is the cardinality of X^N. Let $g = g^e_{ijk}$ be an integer that maps any $\mathbf{x}^e_{ijk}$ to $\mathbf{x}_g$; let $l = i + (N+1)(j-1) + (N+1)^2(k-1) + (N+1)^3(e-1)$ represent a lexicographical ordering of the local nodal values; and let $m = E(N+1)^3$ be the total number of local nodes. We define $\mathbf{Q}^T$ as the $\bar{n} \times m$ Boolean *gather* matrix whose lth column is $\hat{\underline{e}}_{g(l)}$, where $g(l)$ is the local-to-global pointer and $\hat{\underline{e}}_g$ is the gth column of the $\bar{n} \times \bar{n}$ identity matrix. For any $u \in X^N$ we have the global-to-local map $\underline{u}_L = \{\underline{u}^e\}_{e=1}^E = \mathbf{Q}\underline{u}$ (the *scatter* operation), such that the discrete bilinear form (3) becomes

$$a_N(u,v) = \sum_{e=1}^{E} (\underline{v}^e)^T \mathbf{A}^e \underline{u}^e \;=\; (\mathbf{Q}\underline{v})^T \mathbf{A}_L \, \mathbf{Q}\underline{u} \;=\; \underline{v}^T \mathbf{Q}^T A_L \, \mathbf{Q}\underline{u}, \;=\; \underline{v}^T \bar{A} \underline{u},$$

where $\mathbf{A}_L$= block-diag$\{\mathbf{A}^e\}$ is the *unassembled* stiffness matrix and $\bar{\mathbf{A}}$ is the *assembled* stiffness matrix. The preceding definition of $\mathbf{Q}$ is strictly formal. One never actually creates $\mathbf{Q}$ nor $\mathbf{Q}^T$. Rather, the actions of these matrices, applied as a gather-scatter pair, are implemented as a single communication routine as noted below in (22). We refer to [14] for more details and examples of the gather-scatter operation.

Boundary conditions are enforced by formally numbering the boundary nodes last such that $n < \bar{n}$ represents the number of unique interior nodes. Defining $\mathbf{R} = [\mathbf{I}_n \;\; \mathbf{O}]$ as the $n \times \bar{n}$ restriction matrix comprising the $n \times n$ identity $\mathbf{I}_n$ and an $n \times (\bar{n} - n)$ zero matrix $\mathbf{O}$, one defines the global stiffness matrix $\mathbf{A} = \mathbf{R}\bar{\mathbf{A}}\mathbf{R}^T$, which is symmetric positive definite.

To obtain the mass matrix, we consider the inner product,

$$b(u,v) = \int_\Omega v\, u\, d\Omega, \tag{15}$$

which, when combined with the SEM basis and quadrature rule, becomes

$$b_N(v,u) = \sum_{e=1}^{E} (\underline{v}^e)^T \mathbf{J}^e \left(\hat{\mathbf{M}} \otimes \hat{\mathbf{M}} \otimes \hat{\mathbf{M}}\right) \underline{u}^e \;=\; \sum_{e=1}^{E} (\underline{v}^e)^T \mathbf{B}^e \underline{u}^e. \tag{16}$$

Here, the local mass matrix $\mathbf{B}^e := \mathbf{J}^e \left(\hat{\mathbf{M}} \otimes \hat{\mathbf{M}} \otimes \hat{\mathbf{M}}\right)$ is diagonal with entries $J^e_{ijk}\rho_i\rho_j\rho_k$ arising from the product of the Jacobian evaluated at the GLL points with the one-dimensional mass matrix, $\hat{\mathbf{M}} = \text{diag}\{\rho_k\}$. As is the case with the stiffness matrix, the assembled mass matrix is $\bar{\mathbf{B}} = \mathbf{Q}^T\mathbf{B}_L\mathbf{Q}$, where $\mathbf{B}_L$= block-diag$\{\mathbf{B}^e\}$.

For $f(\mathbf{x}) \in L^2$, we do not impose interelement continuity nor vanishing values on $\partial\Omega$. Consequently, for $\alpha = 1$ and $\beta = 0$, the discretized form of (1) has the form

$$\mathbf{A}\,\underline{u} \;=\; \mathbf{R}\mathbf{Q}^T\bar{\mathbf{B}}\underline{f}_L, \tag{17}$$

where $\underline{f}_L$ enumerates all entries of f in V^N. As it is most common to express spectral element functions in terms of local nodal values, we formally write the solution as $\underline{u}_L = \mathbf{Q}\mathbf{R}^T\underline{u}$, which extends the entries of $\underline{u}$ in (17) to the boundary by zero and copies the global values to each local element.

3.3 Arithmetic Operations

The matrix $\mathbf{A}$ is never explicitly formed. In an iterative solver, matrix-vector multiplication is effected by applying $\mathbf{A}^e$ in the factored form (12), which requires $6E(N+1)^3$ storage for the geometric factors and $12E(N+1)^4$ work for application of the derivative matrices. Assuming that u is already represented in its local form, $\underline{u}_L = \{\underline{u}^e\}_{e=1}^E$, local operator evaluation begins with the tensor product–based derivatives,

$$\underline{u}^e_\xi := \mathbf{D}_\xi\underline{u}^e = (\mathbf{I}\otimes\mathbf{I}\otimes\hat{\mathbf{D}})\underline{u}^e = \Sigma_{l=0}^N \hat{D}_{il}u^e_{qjk}, \tag{18}$$

$$\underline{u}^e_\eta := \mathbf{D}_\eta\underline{u}^e = (\mathbf{I}\otimes\hat{\mathbf{D}}\otimes\mathbf{I})\underline{u}^e = \Sigma_{l=0}^N \hat{D}_{jl}u^e_{ilk}, \tag{19}$$

$$\underline{u}^e_\gamma := \mathbf{D}_\gamma\underline{u}^e = (\hat{\mathbf{D}}\otimes\mathbf{I}\otimes\mathbf{I})\underline{u}^e = \Sigma_{l=0}^N \hat{D}_{kl}u^e_{ijl}. \tag{20}$$

These are followed by pointwise multiplications with the geometric factors and application of $\mathbf{D}^T$. The product is completed by applying the assembly and restriction matrices, $\mathbf{R}Q^T$. However, given the desire to have only one storage format, this last step is immediately followed by extension to local form. Consequently, the matrix-vector product $\underline{w} = \mathbf{A}\underline{u}$ in Nekbone is written

$$\underline{w}_L = \mathbf{Q}\mathbf{R}^T\mathbf{R}\mathbf{Q}^T\mathbf{A}_L\underline{u}_L \tag{21}$$

$$= \mathcal{M}_L\mathbf{Q}\mathbf{Q}^T\mathbf{A}_L\underline{u}_L, \tag{22}$$

where $\mathcal{M}_L$ is a diagonal *mask* matrix in local form that is one for all interior nodes and zero for all nodes $\mathbf{x}^e_{ijk} \in \partial\Omega$. With the form (22), the local-to-global mapping steps $\mathbf{QQ}^T$ are combined into a single operation that sums values at shared vertices and redistributes the result to the local representation. This is the gather-scatter operation that involves nearest-neighbor communication in Nekbone.

We note that, aside from (22), the work for all operations in the Jacobi-preconditioned conjugate gradient iteration scale as the total number of grid-points $n = E(N+1)^3$. Accounting for these operations, the work per iteration is approximately

$$W = [12(N+1)^4 + 34(N+1)^3]E = [12(N+1) + 34]n. \tag{23}$$

The number of memory references is linear in n, while the work per grid point asymptotically scales as $12(N+1)$, which leads to an increase in the flops per byte. In actual applications, this increased computational intensity leads to improved accuracy at fixed resolution or reduced resolution (n) for a given accuracy and thus there is a tangible benefit to increasing N.

4 Poisson Operator Evaluation

We now turn to the OpenACC and CUDA Fortran implementation of the spectral element Poisson operator (12), which is the central compute-intensive part of Nekbone. The operator evaluation involves two phases. We first compute the matrix-vector product $\underline{v} = (\mathbf{GD})\underline{u}_L$ to obtain the weighted first-order derivatives. We then proceed with the second matrix-vector multiplication $\underline{w} = \mathbf{D}^T\underline{v}$ which yields the second-order derivative part of the Poisson operator. The data management using OpenACC directives is introduced in Algorithm 1. Algorithms 2–4 show the pseudo-code of the subroutine `ax_acc` involving the two parallel-loops for these procedures. The OpenACC directives in Algorithms 2–3 are optimized for CCE and PGI compilers, respectively. Algorithm 4 is an optimized CUDA version using OpenACC as an interface.

4.1 Data Management

We initiate Nekbone on the CPU. The host CPU manages the data transfers between the host and its corresponding GPU using the OpenACC directives and the GPU executes parallel regions. In Nekbone, we first allocate memory on GPU for the GPU arrays as shown in Algorithm 1 using OpenACC directives `DATA CREATE`. For the repeatedly used arrays such as the differentiation matrices and geometric variables, we compute them on CPU and copy them to GPU by specifying `UPDATE DEVICE` within the OpenACC data region.

In our pseudocodes, we use the arrays for the differentiation matrices

$$\texttt{dxm1(:,:)}{:=}\hat{\mathbf{D}}, \quad \texttt{dxtm1(:,:)}{:=}\hat{\mathbf{D}}^T,$$

Algorithm 1 OpenACC Data Management.

```
!$ACC  DATA CREATE(u,w,wr,ws,wt)
!$ACC&      CREATE(dxm1,dxtm1,gxyz)

            compute the repeatedly used arrays (dxm1,dxtm1,gxyz)

!$ACC UPDATE DEVICE(dxm1,dxtm1,gxyz)

            compute the conjugate iterations (Poisson operator evaluation)

!$ACC END DATA
```

and the geometric variables (note that $\mathbf{G}^{21} = \mathbf{G}^{12}$, $\mathbf{G}^{31} = \mathbf{G}^{13}$, $\mathbf{G}^{32} = \mathbf{G}^{23}$)

$$\begin{array}{ll} \texttt{gxyz(:,:,:,1,:)}{:=}\mathbf{G}^{11}, & \texttt{gxyz(:,:,:,2,:)}{:=}\mathbf{G}^{12}, \\ \texttt{gxyz(:,:,:,3,:)}{:=}\mathbf{G}^{13}, & \texttt{gxyz(:,:,:,4,:)}{:=}\mathbf{G}^{22}, \\ \texttt{gxyz(:,:,:,5,:)}{:=}\mathbf{G}^{23}, & \texttt{gxyz(:,:,:,6,:)}{:=}\mathbf{G}^{33}, \end{array}$$

where these arrays are computed once and stored on the GPU. For the derivatives of $\texttt{u(:,:,:,:)}{:=}\underline{u}$ with respect to ξ, η, and γ, we use scalar variables within the innermost `DO` loop as

$$\texttt{ur}{:=}\mathbf{D}_\xi\underline{u}, \quad \texttt{us}{:=}\mathbf{D}_\eta\underline{u}, \quad \texttt{ut}{:=}\mathbf{D}_\gamma\underline{u}.$$

We also declare some additional working arrays including

$$\texttt{w(:,:,:,:)}, \quad \texttt{wr(:,:,:,:)}, \quad \texttt{ws(:,:,:,:)}, \quad \texttt{wt(:,:,:,:)}.$$

4.2 OpenACC implementation

The requisite matrix-vector products to apply the discretized Poisson operator to the current iterate are effected through the Eqs. (12) and (18)–(20).

Algorithm 2 shows the optimized version for the CCE compiler. We use the `PRESENT` clause in the OpenACC data region to tell that the arrays in the list are already present in GPU memory so that the application will find and use that existing GPU data. If the arrays are not properly placed on the GPU, the program halts with a runtime error. For the CCE version, we optimized the OpenACC parallelism based on gangs, workers, and vector elements. `GANG` is the highest level of parallelism, equivalent to CUDA threadblocks `grid`. `WORKER` is equivalent to CUDA thread within a threadblock. `VECTOR` is the tightest level of single-instruction multithreading (SIMT) dimension, equivalent to CUDA warp. In Algorithm 2, the OpenACC directives COLLAPSE instructs the compiler to collapse the quadruply nested loop into a single loop and then the `GANG WORKER VECTOR` divides the collapsed outer loop into the most efficient parallel decomposition, where the default vector size is 128 for the Kepler architecture. The innermost loop is performed in scalar mode by specifying `LOOP SEQ` with `PRIVATE` clause for the scalar variables, which minimizes the data movement. We note that the combined OpenACC directives with `PARALLEL LOOP COLLAPSE(4) GANG WORKER VECTOR` and `LOOP SEQ`

Algorithm 2 OpenACC (CCE Compiler Version): Poisson Operator

```
subroutine ax_acc(u,w,wr,ws,wt,dxm1,dxtm1,gxyz)
!$ACC DATA PRESENT(u,w,wr,ws,wt,dxm1,dxtm1,gxyz)
!$ACC PRIVATE(ur,us,ut)
!$ACC PARALLEL LOOP COLLAPSE(4) GANG WORKER VECTOR
     do e = 1,E
        do k = 1,N+1
        do j = 1,N+1
        do i = 1,N+1
           ur = 0
           us = 0
           ut = 0
!$ACC LOOP SEQ
           do l = 1,N+1    ! serial loop, no reduction needed
              ur = ur + dxm1(i,l)*u(l,j,k,e)
              us = us + dxm1(j,l)*u(i,l,k,e)
              ut = ut + dxm1(k,l)*u(i,j,l,e)
           enddo
           wr(i,j,k,e) = gxyz(i,j,k,1,e)*ur + gxyz(i,j,k,2,e)*us + gxyz(i,j,k,3,e)*ut
           ws(i,j,k,e) = gxyz(i,j,k,2,e)*ur + gxyz(i,j,k,4,e)*us + gxyz(i,j,k,5,e)*ut
           wt(i,j,k,e) = gxyz(i,j,k,3,e)*ur + gxyz(i,j,k,5,e)*us + gxyz(i,j,k,6,e)*ut
        enddo
        enddo
        enddo
     enddo
!$ACC END PARALLEL LOOP

!$ACC PARALLEL LOOP COLLAPSE(4) GANG WORKER VECTOR
     do e = 1,E
        do k = 1,N+1
        do j = 1,N+1
        do i = 1,N+1
!$ACC LOOP SEQ
           do l = 1,N+1
              w(i,j,k,e) = w(i,j,k,e) + dxtm1(i,l)*wr(l,j,k,e)
                                      + dxtm1(j,l)*ws(i,l,k,e)
                                      + dxtm1(k,l)*wt(i,j,l,e)
           enddo
        enddo
        enddo
        enddo
     enddo
!$ACC END PARALLEL LOOP
!$ACC END DATA
     return
     end
```

deliever a better performance, compared to a simple instruction with `PARALLEL LOOP COLLAPSE(4)`.

Algorithm 3 is an optimized version for PGI compiler based on OpenACC directives using `KERNELS` construct with the combined `GANG` and `LOOP VECTOR` directives. The `GANG` parallelism is equivalent to CUDA grid-level parallelism and the `LOOP VECTOR` parallelism is equivalent to CUDA thread-level parallelism. Within `KERNELS` construct, the PGI compiler optimizes the actual code into the most efficient parallelism mapping. Thus, we obtain different performance depending on different versions of PGI compilers (mostly the latest version of PGI compiler delivers a better performance than an older version of PGI compiler). We also note that specifying the vector length with `VECTOR(N+1)`

Algorithm 3 OpenACC (PGI Compiler Version): Poisson Operator

```
subroutine ax_acc(u,w,wr,ws,wt,dxm1,dxtm1,gxyz)
!$ACC DATA PRESENT(u,w,wr,ws,wt,dxm1,dxtm1,gxyz)
!$ACC KERNELS
!$ACC GANG
     do e = 1,E
!$ACC LOOP VECTOR(N+1)
        do k = 1,N+1
!$ACC LOOP VECTOR(N+1)
        do j = 1,N+1
!$ACC LOOP VECTOR(N+1)
        do i = 1,N+1
           ur = 0
           us = 0
           ut = 0
!$ACC LOOP SEQ
           do l = 1,N+1
              ur = ur + dxm1(i,l)*u(l,j,k,e)
              us = us + dxm1(j,l)*u(i,l,k,e)
              ut = ut + dxm1(k,l)*u(i,j,l,e)
           enddo
           wr(i,j,k,e) = gxyz(i,j,k,1,e)*ur + gxyz(i,j,k,2,e)*us + gxyz(i,j,k,3,e)*ut
           ws(i,j,k,e) = gxyz(i,j,k,2,e)*ur + gxyz(i,j,k,4,e)*us + gxyz(i,j,k,5,e)*ut
           wt(i,j,k,e) = gxyz(i,j,k,3,e)*ur + gxyz(i,j,k,5,e)*us + gxyz(i,j,k,6,e)*ut
        enddo
        enddo
        enddo
     enddo
!$ACC END KERNELS

!$ACC KERNELS
!$ACC GANG
     do e=1,E
!$ACC LOOP VECTOR(N+1)
        do k = 1,N+1
!$ACC LOOP VECTOR(N+1)
        do j = 1,N+1
!$ACC LOOP VECTOR(N+1)
        do i = 1,N+1
!$ACC LOOP SEQ
           do l = 1,N+1
              w(i,j,k,e) = w(i,j,k,e) + dxtm1(i,l)*wr(l,j,k,e)
                                      + dxtm1(j,l)*ws(i,l,k,e)
                                      + dxtm1(k,l)*wt(i,j,l,e)
           enddo
        enddo
        enddo
        enddo
     enddo
!$ACC END KERNELS
!$ACC END DATA
     return
     end
```

gives a better performance than the default length with `VECTOR(64)`, while the typical range of our vector length `N+1` goes up to 16.

4.3 CUDA Fortran Implementation with OpenACC

Finally we discuss OpenACC interoperability. In our approach, we consider OpenACC for data allocation, copy, and compute kernel launch, which would

Algorithm 4 OpenACC Calling CUDA Fortran Kernel for Poisson Operator

```
      subroutine ax_acc(w,u,ur,us,ut,gxyz,dxm1,dxtm1)
      use cudafor

!$ACC DATA PRESENT(w,u,gxyz,ur,us,ut,dxm1,dxtm1)
!$ACC HOST_DATA USE_DEVICE(w,u,ur,us,ut,gxyz,dxm1,dxtm1)

      call ax_cuda<<<E,dim3(N+1,N+1,N+1)>>>(w,u,ur,us,ut,gxyz,dxm1,dxtm1)

!$ACC END HOST_DATA
      return
      end
```

Algorithm 5 CUDA Fortran Kernel: Poisson Operator

```
      attribute (global) subroutine ax_cuda(w,u,ur,us,ut,gxyz,dxm1,dxtm1)
      real, shared :: dxm1_s (N+1,N+1)
      real, shared :: dxtm1_s(N+1,N+1)

      e = blockIdx%x
      k = threadIdx%z
      j = threadIdx%y
      i = threadIdx%x

      dxm1_s (i,j) = dxm1 (i,j)
      dxmt1_s(i,j) = dxtm1(i,j)

      call syncthreads()

      rtmp = 0.0
      stmp = 0.0
      ttmp = 0.0
      do l = 1,N+1
         rtmp = rtmp + dxm1_s(i,l)*u(l,j,k,e)
         stmp = stmp + dxm1_s(j,l)*u(i,l,k,e)
         ttmp = ttmp + dxm1_s(k,l)*u(i,j,l,e)
      enddo
      ur(i,j,k) = gxyz(i,j,k,1,e)*rtmp + gxyz(i,j,k,2,e)*stmp + gxyz(i,j,k,3,e)*ttmp
      us(i,j,k) = gxyz(i,j,k,2,e)*rtmp + gxyz(i,j,k,4,e)*stmp + gxyz(i,j,k,5,e)*ttmp
      ut(i,j,k) = gxyz(i,j,k,3,e)*rtmp + gxyz(i,j,k,5,e)*stmp + gxyz(i,j,k,6,e)*ttmp

      call syncthreads()

      do l = 1,N+1
         w(i,j,k,e) = w(i,j,k,e) + dxtm1_s(i,l)*ur(l,j,k,e)
                                 + dxtm1_s(j,l)*us(i,l,k,e)
                                 + dxtm1_s(k,l)*ut(i,j,l,e)
      enddo
      return
      end
```

remain no change in the data management structure of our OpenACC version that we have discussed in previous sections. Then we enhance the performance for compute-intensive routines using CUDA Fortran implementation.

The interoperability feature of OpenACC allows us to call CUDA functions using the arrays that we have already copied to the GPU using OpenACC. Algorithm 4 demonstrates the Poisson operator subroutine `ax_acc` on the host, calling a CUDA Fortran function `ax_cuda` that launches a CUDA kernel shown in Algorithm 5. Code running on the host manages the memory on both the

host and device, and also launches kernels which are subroutines executed on the device. These kernels are executed by many GPU threads in parallel. The launch is asynchronous so that the host program continues and may issue other launches.

In Algorithm 4, we use a module `cudafor` which contains the CUDA Fortran definitions, provided by the PGI CUDA Fortran compiler. The `HOST_DATA` OpenACC construct makes the address of device data available on the host, so you can pass it to functions that expect CUDA device pointers. Whenever we use the arrays listed in the `USE_DEVICE` clause within the `HOST_DATA` region, the compiler generates code to use the device copy of the arrays, instead of the host copy. The CUDA launch configuration uses the chevron syntax, `<<<`grid,block`>>>`, which dictates how many device threads execute the kernel in parallel. CUDA launches a kernel with a `grid` of thread blocks. The second argument in the execution configuration specifies the number of threads in a thread block. The first argument specifies the number of thread blocks in the grid. Threads in a thread block can be arranged in a multidimensional manner, containing `x`, `y`, and `z` components for the derived type `dim3`.

In Algorithm 5, the CUDA Fortran predefined variables are `blockidx` and `threadidx`. We assign the index `e` ($e = 1, ..., E$) for the mesh elements for `blockid%x` as shown in Algorithm 4 and the indices `i,j,k` for the x, y, z coordinates for `threadid%x`, `threadid%y`, and `threadid%z`, respectively. The utilization of shared memory in GPUs is very important for writing optimized CUDA code since access to shared memory is much faster than to global memory. Shared memory is allocated per thread block (with 64KB shared memory limit), therefore all threads in a block have access to the same shared memory. We declare the arrays `dxm1_s` and `dxtm1_s` as shared memory variables for `dxm1` and `dxtm1`, respectively. We note that the vector length of a threadblock is limited to 1024. For $N > 9$, the block size $(N + 1)^3$ exceeds 1024. Thus, for example, we set `dim3(N+1,N+1,(N+1)/2)` with the repeated procedure of the `DO` loops in `ax_cuda` for the different set of indices for the third component $k = 1, ..., (N + 1)/2$ and $k = (N + 1)/2, ..., N + 1$ for $N = 11$. A similar setting can be applied for different sizes of N with `dim3(N+1,N+1,(N+1)/4)` and `dim3(N+1,N+1,(N+1)/8)`. We omit the detailed pseudocode here.

5 Communication Kernel

Nekbone uses a C-based gather-scatter routine, `gs_op`, that supports local-gather, global-scatter, global-gather, and local-scatter procedures for exchanging data between neighboring elements and executing the direct-summation operation $\mathbf{Q}\mathbf{Q}^T$, referred as `dssum`. In the CPU version, users pass field data by calling `gs_op` from Fortran or C and `gs_op` automatically handles the communication, selecting the fastest option among *pairwise*, *all-reduce*, or *crystal-router* exchanges. Dense stencils, with 100s of nonzeros per row (e.g., as arise in lower levels of algebraic multigrid) tend to run faster with the all-reduce or crystal-router strategies. To effect $\mathbf{Q}\mathbf{Q}^T$ on the GPU without modifying

the user interface we have extended the *gs* library with OpenACC pragmas for the gather-scatter operations that are local to the node, as demonstrated in Algorithm 6 [7]. The local *gs* operations are performed with OpenACC directives `PARALLEL LOOP`. In Algorithm 6, global data that is to be exchanged between neighboring elements on different nodes are passed to the standard (MPI-based) `gs_op` for non-local exchanges. This process requires a GPU-CPU data copy, followed by the MPI exchange, and then a copy back to the GPU. While this is not a fully optimized approach it delivers reasonable results when the system does not support GPUDirect.

More recently, we developed a fully optimized GPU-based `gs` library [8] and demonstrated the performance advantages of this approach in the context of the time-dependent Maxwell's equations, discretized by a spectral element discontinuous Galerkin method (SEDG). Nekbone uses the standard SEM, which requires face, edge, and vertex exchanges whereas SEDG requires data exchanges only between faces. In this paper, we show the performance of the Poisson solver Nekbone using [8]. The fully optimized version accelerates all four parts of *gs*: local-gather, global-scatter, global-gather, and local-scatter where accelerating the global loops allows us to use the GPUDirect pragmas, as the buffers are prepared on the GPU. In this new version, data communication between CPUs is not necessary since the GPU would efficiently perform the local additions and does not need any information from other nodes. We compare the performance of the GPUDirect-enabled communication kernel [8] with that of Algorithm 6 [7] without GPUDirect support.

6 Performance Results

We demonstrate and analyze Nekbone's performance on Titan and Curie.

Algorithm 6 Local-gather and local-scatter operations on GPU [7].

```
 unew_l = u_l
 % u_g = Q u_l Local Gather on GPU
 !$ACC PARALLEL LOOP
 u_g = 0
 do i = 1, nl
    li = lgl(1,i)
    gi = lgl(2,i)
    u_g(gi) = u_g(gi)+u_l(li)
 enddo

!$ACC UPDATE HOST(u_g)    % copy GPU->CPU
call gs_op(u_g,1,1,0)     % MPI communication between CPUs
!$ACC UPDATE DEVICE(u_g)  % copy CPU->GPU

% u_l = Q^T u_g Local Scatter on GPU
!$ACC PARALLEL LOOP
do i = 1, nl
   li = lgl(1,i)
   gi = lgl(2,i)
   unew_l(li) = u_g(gi)
enddo
```

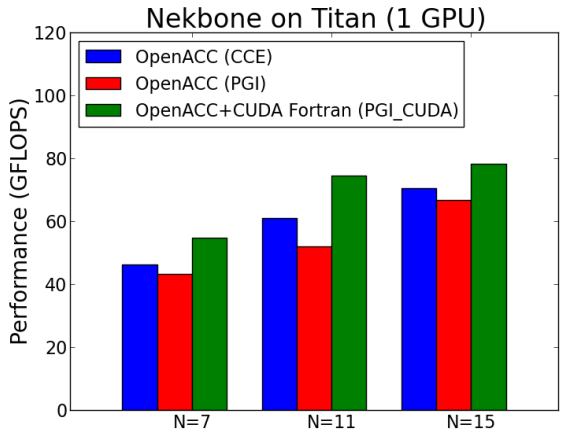


Fig. 1 Single-GPU performance on Titan for OpenACC CCE/PGI versions and OpenACC+CUDA Fortran version; $n = E(N+1)^3$ for $E = 512$ with varying $N = 7, 11, 15$.

Titan is a Cray XK7 system at the Oak Ridge Leadership Computing Facility (OLCF), consisting of 18,688 AMD Opteron 6274 16-core CPUs at 2.6 GHz and 18,688 Nvidia Tesla K20X GPUs. Titan has a hybrid architecture with peak performance of 27 Pflops. The pair of nodes shares a Gemini high-speed interconnect router in a 3D torus topology.

Curie is a PRACE Tier-0 system, operated at the French Alternative Energies and Atomic Energy Commission (CEA) in France. Curie has total of 144 hybrid nodes. Each hybrid node contains two 4-core Westmere-EP CPUs at 2.67 GHz and 2 Nvidia M2090 GPUs. The compute nodes are connected through a QDR InfiniBand network with a full fat-tree topology.

Titan supports both Cray CCE and PGI compilers with GPUDirect. Curie supports only the PGI compiler with no GPUDirect. For a single-GPU performance, we demonstrate the results on Titan with CCE, PGI, and PGI CUDA compilers and the results on Curie with the PGI compiler. For multi-GPU performance, we demonstrate strong scaling and weak scaling on Curie with the PGI compiler using up to 256 GPUs without GPUDirect and weak scaling on Titan with the CCE compiler using up to 16,384 GPUs with and without GPUDirect. All our performance tests are based on double-precision runs. For PGI compiler we used the flag `-ta=tesla:managed` to allocate managed memory, i.e. it actually allocates host pinned memory as well as device memory. Cray CCE compiler supports such flag for host pinned memory by default.

6.1 Single-GPU Performance

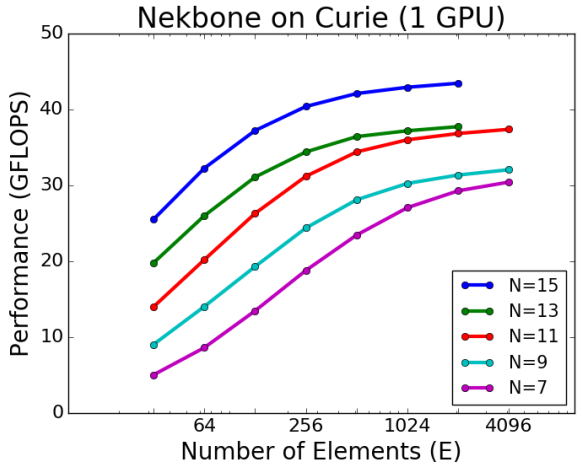
Fig. 1 shows the results for a single GPU on Titan based on Algorithms 2–3 with optimized OpenACC directives for the CCE and PGI compilers and Algorithms 4–5 with the optimized OpenACC+CUDA Fortran version for the

PGI CUDA Fortran compiler. We fixed the number of elements $E = 512$ and varied the polynomial order as $N = 7, 11, 15$. As noted in (23), the computational intensity scales linearly with N, and thus the performance on the GPU increases dramatically as N increases. For $N = 7$ with $n = 262,144$ we achieve 46.2, 43.3, and 54.7 Gflops for the CCE, PGI, and PGI CUDA versions, respectively. For $N = 15$ with $n = 2,097,152$ $(512 \cdot 16^3)$, we achieve 70.4, 66.7, and 78.2 Gflops for the CCE, PGI, and PGI CUDA versions, respectively. For the CCE and PGI versions, the performance continues to increase linearly as N increases. For the PGI CUDA Fortran version, the performance with $N = 11$ is close to that with $N = 15$. The CUDA Fortran version is around $1.2 \sim 1.5\times$ faster than the OpenACC code. Although CUDA Fortran with OpenACC required additional management of the arrays for the larger N cases because of the block size limit (1,024), we note that the amount of effort to port Nekbone to CUDA using OpenACC is not comparable with that of a CUDA-only port. A porting strategy that mixes OpenACC with CUDA seems to be a reasonable option for high performance.

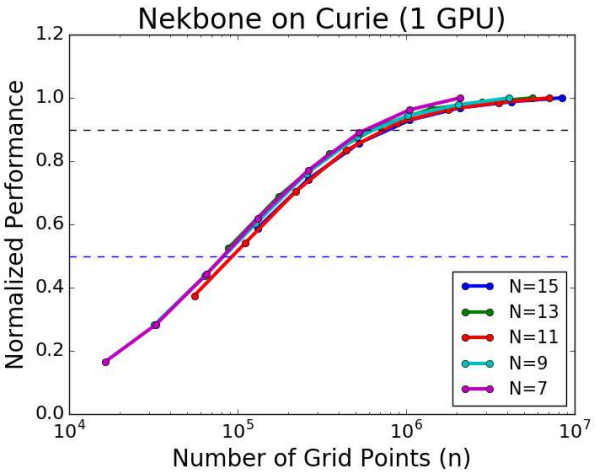
Fig. 2 shows the performance of Algorithm 3 on Curie with the PGI complier for a single GPU as a function of E and N. We observe in Fig. 2(a) an increase in flop rate with both E and N that levels off as E is increased. The performance gain due to increased computational intensity in Eq. (23) is noticeable as N is increased. We note that the largest case we can consider for N=15 is E=2048, which achieves 43.46 Gflops. Larger problems do not fit into memory (i.e., are memory bound), but this situation poses no particular difficulty because in practice one can add more nodes and have an increase in memory capacity (and bandwidth). In fact, for E=256, the single-GPU performance drops only slightly, to 40.5 Gflops. Thus, one can effectively use 8 GPUs for a case with $(E, N) = (2048, 15)$ and anticipate running 8 times as fast as with a single GPU. We consider multi-GPU performance in the next section. In Fig. 2(b) we plot the same data, but the horizontal axis is now $n = E(N + 1)^3$, and the vertical axis is $\text{Gflops}(E, N)/\text{Gflops}(E_{\max}, N)$, which represents the *fraction of realizable peak* for a given (E, N) pairing. For example, for N=9, the peak performance is 32 Gflops and the N=9 curve is normalized by this value. Fig. 2(b) shows that $n_{\frac{1}{2}}$, the (local) problem size required to reach one-half of the realizable peak performance, is around 80,000 to 90,000 for this configuration. Similarly, one needs about 500,000 gridpoints to saturate at 90% of the realizable peak. Remarkably, these numbers are only weakly dependent on N.

6.2 Multi-GPU Performance

Our multi-GPU performance involves either CPU-CPU MPI communication (no GPUDirect) or GPU-GPU MPI communication (GPUDirect). We begin with discussing multi-GPU performance on Curie based on the OpenACC PGI version in Algorithm 3 with no GPUDirect support.



(a)



(b)

Fig. 2 Single-GPU performance on Curie with PGI version; $n = E(N+1)^3$ for varying $E = 32, 64, 128, 256, 512, 1024, 2048, 4096$ and $N = 7, 9, 11, 13, 15$; (a) performance in FLOPS and (b) normalized performance.

Fig. 3 shows strong scaling of Nekbone on Curie, measured in Tflops. Here $(E, N) = (32768, 15)$ ($n = 134217728$) for a number of GPUs ranging from P=32 to 256. Linear scaling is observed for n/P out to 2 million (i.e., for P=64). The performance is 3.4 Tflops on 128 GPUs and 5.7 Tflops on 256 GPUs, demonstrating 88% and 72% scaling efficiency, respectively, compared with P=32. We note, however, that 5.7 Tflops is only 55% of the 10.4 Tflops performance that one might expect from Fig. 2. Part of the loss is due to communication overhead, but part of it also comes from using two GPUs per node, as shown in Fig. 4.

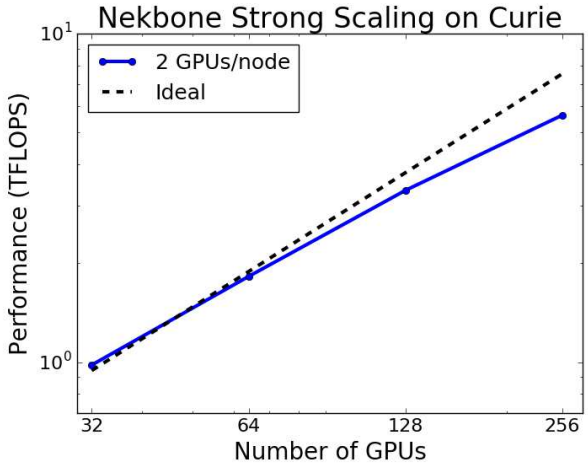


Fig. 3 Strong scaling on Curie with PGI version using 2 GPUs/node; $n = (1024 \cdot 32) \cdot 16^3$.

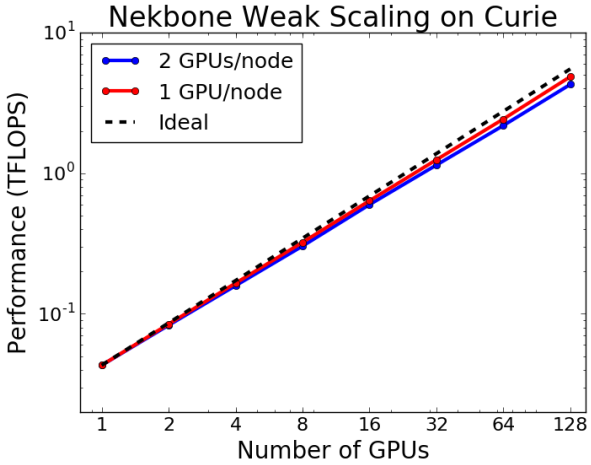


Fig. 4 Weak scaling on Curie with PGI version using 1 GPU/node (up to 128 nodes) and 2 GPUs/node (up to 64 nodes); 1, 32, 64, 128 GPUs with $n = 1024 \cdot 16^3$ per GPU.

We note that Fig. 3 shows 2 GPUs per node on Curie. Each Curie hybrid node has 2 sockets, and each GPU is bound to a socket. By default both processes run on only one GPU. As a result, the `CUDA_VISIBLE_DEVICES` variable should be set to 0 or 1 depending on the rank of the process. In addition, one must bind the processes to the GPU in order to ensure that they run on the same socket that hosts the desired GPU. However, binding the processes slows the application while both processes share some resources. Fig. 4 shows the weak-scaling performance on Curie, comparing the performance of those two

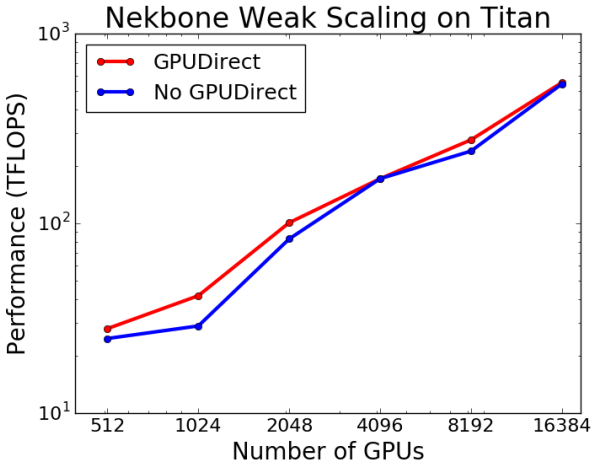


Fig. 5 Weak scaling on Titan with CCE version using GPUDirect and no GPUDirect on 512, 1024, 2048, 4096, 8192, 16384 GPUs (1 GPU/node); n=$1024 \cdot 16^3$ per GPU.

cases (1 GPU/node and 2 GPUs/node). Both cases are using $n = 1024 \cdot 16^3$ per GPU. We observe the superior performance of 4.8 Tflops using 128 GPUs on 128 nodes, compared with 4.2 Tflops using 128 GPUs on 64 nodes. Of course, since one is charged for node hours, using the second GPU results in a 1.875 speedup for a given set of resources.

We performed weak-scaling tests of Nekbone with up to 16,384 GPUs on Titan with the CCE compiler with and without GPUDirect communication. Fig. 5 shows flops rates with GPUDirect and $n = 4,194,304$ $(1024 \cdot 16^3)$ grid-points per GPU. We achieve 552 Tflops on $16,384$ GPUs with GPUDirect and 542 Tflops on $16,384$ GPUs with no GPUDirect. As the results shown in [8], we achieve 5–10% improvement in performance with GPUDirect, compared with no GPUDirect.

6.3 Discussion of Performance Data

We note that there are several factors that limit strong scalability of distributed memory computing. In addition to communication as the traditional source of overhead, GPU-enabled nodes have other limitations that put a lower bound on the granularity, n/P, that ultimately limits the amount of parallelism for applications on these architectures. As it is important for users to recognize these bounds when interpreting performance results we give a brief synopsis of the performance data presented above.

Fig. 2 illustrates for all values of N considered, one needs at least 100,000 points per GPU to exceed 50% parallel efficiency and that one would need 500,000 points per GPU to exceed 90% efficiency. These numbers are the

respective $n_{\frac{1}{2}}$ and $n_{.9}$ values for OpenACC-enabled Nekbone running on a single GPU on *Curie* with no communication overhead. When one accounts for node resource contention that arises when running two GPUs per node, the efficiency drops by another factor of 1.875/2. (Here, 1.875 is the node speedup observed when the second GPU is enabled on a node.) Thus, for the strong-scaling study of Fig. 3, the parallel losses leading to an observed efficiency of 55% at P=256 GPUs can be attributed roughly as follows: 10% due to single-node performance drop for $n/P = 524288 \approx n_{.9}$, 6.25% due to insufficient node resources to support two GPUs at full speed, and the remaining 30% due to communication/synchronization overhead.

7 Conclusions

We implemented a hybrid GPU version of Nekbone to exploit the processing power of multi-GPU systems using OpenACC and CUDA Fortran programming. Our focus is on optimizing the use of OpenACC directives in order to boost GPU performance on different platforms and to gain insight into strategies that will be beneficial for tuning the more complicated production codes.

We note that CUDA Fortran with OpenACC is slightly more efficient than straight OpenACC. Given that the amount of effort to port Nekbone to CUDA using OpenACC is significantly less than for a CUDA-only port, this mixed approach might prove a viable strategy for ultimate tuning of accelerated code. Using the GPUDirect communication with the highly tuned gather-scatter kernel developed in [8], Nekbone achieves 552 Tflops on $16,384$ GPUs of the OLCF Cray XK7 Titan.

We have also examined some of the node-specific sources of performance loss. Among these we note that there is a reduction in peak single-GPU performance on Curie from 43 GFLOPS to 30 GFLOPS as N is reduced from 15 to 7. This reduction is consistent with the complexity estimate (23), which indicates a computational intensity (flops/word) scaling as $\approx (12N + 46)/7$. In addition, one needs at least 100,000 points per GPU to exceed 50% parallel efficiency and 500,000 points per GPU to exceed 90% efficiency. Using two GPUs per node, one realizes a 1.875 speedup over a single GPU, implying about a 6% efficiency loss for this configuration.

In summary, using Nekbone as a surrogate for Nek5000 or other similar spectral element codes, we anticipate sustained perfomance of approximately 40 GFLOPS per GPU if one has at least 500,000 points per GPU. There is a 5–10% gain through the use of GPUDirect communication and similar gain through combined OpenACC/CUDA programming for key kernels.

Acknowledgements This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357, and partially supported by the Swedish e-Science Research Centre (SeRC). This research used resources of the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The research also used

computing resources of the French Alternative Energies and Atomic Energy Commission (CEA) in France via the Partnership for Advanced Computing in Europe (PRACE).

References

1. D. C. Jespersen, "Acceleration of a CFD code with a GPU", *Scientific Programming*, vol. 18, no. 3–4, pp. 193–201, 2010
2. T. Hoshino and N. Maruyama and S. Matsuoka and R. Takaki, "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application" in *the Proceeding of 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, Delft, The Netherlands, May 13-16, 2013.
3. J. Kraus and M. Schlottke and A. Adinetz and D. Pleiter, "Accelerating a C++ CFD code with OpenACC", in *the Proceedings of the First Workshop on Accelerator Programming using Directives* SC14, pp. 47–54, LA, USA, 2014
4. Y. Xia and H. Luo and L. Luo and J. Edwards and J. Lou, OpenACC acceleration of an unstructured CFD solver based on a reconstructed discontinuous Galerkin method for compressible flows, International Journal for Numerical Methods in Fluids, Vol. 78(3), 2015, pp.123–139
5. K. Niemeyer and C. Sung, "Recent progress and challenges in exploiting graphics processors in computational fluid dynamics", *The Journal of Supercomputing*, vol. 67, no. 2, pp. 528–564, 2014.
6. J. Gong, S. Markidis, M. Schliephake, E. Laure, D. Henningson, P. Schlatter, A. Peplinski, A. Hart, J. Doleschal, D. Henty, and P. Fischer, Nek5000 with OpenACC, in *Solving Software Challenges for Exascale, the International Conference on Exascale Applications and Software, EASC 2014 Stockholm, Sweden, April 20-23, 2014*, Stefano Markidis, Erwin Laure (Eds.), Springer LNCS8759, 2015.
7. S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, and P. Fischer, "OpenACC acceleration of the Nek5000 spectral element code", *International Journal of High Performance Computing Applications*, vol. 29, pp. 311–319, 2015.
8. M. Otten, J. Gong, A. Mametjanov, A. Vose, J. Levesque, P. Fischer, and M. Min, "An MPI/OpenACC Implementation of a High Order Electromagnetics Solver with GPUDirect Communication", accepted in *International Journal of High Performance Computing Applications*, 2015.
9. P. F. Fischer, J. W. Lottes, and S. G. Kerkemeier, Nek5000 web page, Web page: http://nek5000.mcs.anl.gov.
10. P. Fischer and J. W. Lottes, "Hybrid Schwarz-multigrid methods for the spectral element method: extensions to Navier-Stokes", *Domain Decomposition Methods in Science and Engineering Series, edited by Kornhuber, R. and R. Hoppe and J. Périaux and O. Pironneau and O. Widlund and J. Xu*, Springer, Berlin, 2004.
11. J. W. Lottes, P. Fischer, "Hybrid multigrid/Schwarz algorithms for the spectral element method",*Journal of Scientific Computing*, vol. 24, pp. 45–78, 2005.
12. P. Fischer, J. Lottes, W. D. Pointer, and A. Siegel, "Petascale Algorithms for Reactor Hydrodynamics", *Journal of Physics: Conference Series*, vol. 125, 012076, 2008.
13. H. M. Tufo and P. Fishcer, "Fast Parallel Direct Solvers for Coarse-Grid Problems", *Journal of Parallel Distribution Computing*, vol. 61, pp. 151–177, 2001.
14. M. Deville, P. Fischer, and E. Mund, *High-order methods for incompressible fluid flow*, Cambridge University Press, 2002