Journal Title
XX(X):1–13

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

An MPI/OpenACC Implementation of a High-Order Electromagnetics Solver with GPUDirect Communication

Matthew Otten[1,3], Jing Gong[2], Azamat Mametjanov[3], Aaron Vose[4], John Levesque[4], Paul Fischer[3,5], and Misun Min[3]

Abstract

We present performance results and an analysis of an MPI/OpenACC implementation of an electromagnetic solver based on a spectral-element discontinuous Galerkin discretization of the time-dependent Maxwell equations. The OpenACC implementation covers all solution routines, including a highly tuned element-by-element operator evaluation and a GPUDirect gather-scatter kernel to effect nearest-neighbor flux exchanges. Modifications are designed to make effective use of vectorization, streaming, and data management. Performance results using up to 16,384 GPUs of the Cray XK7 supercomputer Titan show more than 2.5× speedup over CPU-only performance on the same number of nodes (262,144 MPI ranks) for problem sizes of up to 6.9 billion grid points. We discuss performance enhancement strategies and the overall potential of GPU-based computing for this class of problems.

Keywords

Hybrid MPI/OpenACC, GPUDirect, Spectral Element Discontinuous Galerkin

Introduction

Graphics processing units (GPUs) together with central processing units (CPUs) offer the potential for high compute performance and high sustained memory bandwidth at substantially reduced costs and power consumption over traditional CPUs. As such, they have become prevalent in major computing centers around the world and are likely to be at the core of high-performance computing (HPC) platforms for at least another generation of architectures. In this paper, we consider a multi-GPU implementation of the computational electromagnetics code NekCEM, which is based on a spectral-element discontinuous Galerkin (SEDG) discretization in space coupled with explicit Runge-Kutta timestepping. Our objective is twofold. First, we seek to develop a high-performance GPU-based operational variant of NekCEM that supports the full functionality of the existing MPI code in Fortran/C. Second, we use the highly tuned multi-GPU and multi-CPU codes for performance analysis, from which we can identify performance bottlenecks and infer potential scalability for GPU-based architectures of the future.

NekCEM, the vehicle for our analysis, supports globally unstructured meshes comprising body-fitted curvilinear hexahedral elements. NekCEM's SEDG formulation requires data communication only for flux exchanges between neighboring element faces and not the edges or vertices and thus has a relatively low communication overhead per gridpoint. The restriction to hexahedral (brick) elements allows the discrete operators to be expressed as efficient matrix-matrix products applied to arrays of tensor product basis coefficients. The basis functions are Lagrange interpolants based on Gauss-Lobatto-Legendre (GLL) quadrature points, which provide numerical stability and allow the use of diagonal mass matrices with minimal quadrature error.

We use OpenACC (Open Accelerator) as a strategy for porting NekCEM to multiple GPUs, because of the relative ease of the pragma-based porting. OpenACC is a directive-based HPC parallel programming model, using host-directed execution with an attached accelerator device (openacc.org 2011). We utilize OpenACC compiler directives to specify parallel loops and regions within existing routines in NekCEM that are to be targeted for offloading from a host CPU to an attached GPU device. In our programming model, the CPU host initiates the execution of the code and distributes data between CPUs. Before starting time-advancing iterations, each CPU executes data movement from the host CPU memory to GPU memory only once, and computations are performed fully on GPUs during the timestepping iterations. At the completion of the timestepping iterations, the host CPUs transfer GPU results back to the hosts. In this work, we tuned our gather-scatter kernel using GPUDirect, which enables a direct path for data exchange between GPUs (Nvidia 2015). This results in completely bypassing the host CPUs so that no significant data movement occurs between CPU and GPU during the

[1]Department of Physics, Cornell University, Ithaca, NY 14853
[2]KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden
[3]Mathematics and Computer Science, Argonne National Laboratory, Lemont, IL 60439
[4]Cray's Suercomputing Cener of Excellence, Oak Ridge National Laboratory, Oak Ridge, TN 37831
[5]Department of Computer Science & Department of Mechanical Engineering, Univerisity of Illinois at Urbana-Champaign, Champaign, IL 61801

Corresponding author:
Misun Min, Mathematics and Computer Science, Argonne National Laboratory, Lemont, IL 60439
Email: mmin@mcs.anl.gov

timestepping iterations, unless checkpointing is needed. Our performance does not study checkpointing because it is typically not a bottleneck in production simulations.

We comment that this work parallels other GPU + spectral-element developments, particularly the OpenACC work of J. Gong and collaborators at KTH (Markidis et al. 2015) and OCCA-based efforts of Warburton and coworkers at Rice (Medina et al. 2013). We also note the analysis by Klöeckner et al. (Klöeckner et al. 2009) of high-order nodal discontinuous Galerkin methods on GPUs for solving the Maxwell equations that are based on Nvidia CUDA programming model, using tetrahedral elements.

The remainder of the paper is organized as follows. We begin with an overview of the NekCEM code, including the governing equations, discretizations, and computational approach. We then present details of the key accelerated computational kernels and of the modified gather-scatter communication kernel. Following a brief overview of several target platforms, we provide performance results and analysis that includes discussion of profiling and power consumption considerations. We conclude with a few summary remarks.

Code Description

NekCEM is an Argonne-developed computational electromagnetics code that solves the Maxwell, Helmholtz, drift-diffusion, Schrödinger, and density matrix equations. In this paper, we consider the Maxwell equation solver based on a SEDG discretization in space coupled with explicit high-order Runge-Kutta (RK) and exponential integration time-marching (Carpenter and Kennedy 1994; Min and Fischer 2011). It features body-fitting, curvilinear hexahedral elements that avoid the stairstepping phenomena of traditional finite-difference time-domain methods (Taflove and Hagness 2000) and yield minimal numerical dispersion because of the exponentially convergent high-order bases (Hesthaven et al. 2007). Tensor product bases of the one-dimensional Lagrange interpolation polynomials using the GLL grid points and weights result in a diagonal mass matrix with no additional cost for mass matrix inversion, which makes the code highly efficient. The hexahedral elements allow efficient operator evaluation with memory access costs scaling as $O(n)$ and work scaling as $O(nN)$, where $n = E(N+1)^3$ is the total number of grid points, E is the number of elements, and N is the polynomial approximation order (Deville et al. 2002).

Formulations The central computational component of NekCEM is the evaluation of the right-hand side in time advancement of Maxwell's equations,

$$\epsilon \frac{\partial \mathbf{E}}{\partial t} = \nabla \times \mathbf{H}, \qquad \mu \frac{\partial \mathbf{H}}{\partial t} = -\nabla \times \mathbf{E}, \tag{1}$$

which are shown here for the electric and magnetic fields (resp. $\mathbf{E} = (\mathbf{E}_x, \mathbf{E}_y, \mathbf{E}_z)$ and $\mathbf{H} = (\mathbf{H}_x, \mathbf{H}_y, \mathbf{H}_z)$) in source-free form. To (1) we must add initial and boundary conditions as well as auxiliary equations that are relevant to the physics but ignored in the present context where our focus is on the compute-intensive kernel. NekCEM follows standard SEDG formulations (Hesthaven and Warburton 2008; Min and Fischer 2011). In the discontinuous Galerkin approach, we define the following weak formulation on the computational domain $\Omega = \cup_{e=1}^{E} \Omega^e$ with nonoverlapping hexahedral elements Ω^e:

$$\left(\epsilon \frac{\partial \mathbf{E}}{\partial t} + \nabla \cdot \mathbf{F}_H, \phi_s\right)_{\Omega^e} = (\mathbf{n} \cdot [\mathbf{F}_H], \phi_s)_{\partial\Omega^e}, \tag{2}$$

$$\left(\mu \frac{\partial \mathbf{H}}{\partial t} + \nabla \cdot \mathbf{F}_E, \psi_s\right)_{\Omega^e} = (\mathbf{n} \cdot [\mathbf{F}_E], \psi_s)_{\partial\Omega^e}, \tag{3}$$

where $\mathbf{F}_H = e_s \times \mathbf{H}$ and $\mathbf{F}_E = -e_s \times \mathbf{E}$ for the canonical unit vector $e_s (s = 1, 2, 3)$ with proper test functions ϕ_s and ψ_s. Here, the integrands $[\mathbf{F}_{H,E}] := \mathbf{F}_{H,E} - \mathbf{F}^*_{H,E}$ are defined with upwind numerical fluxes $\mathbf{F}^*_H$ and $\mathbf{F}^*_E$ given in (Hesthaven and Warburton 2008). The resulting surface flux integrands are

$$\mathbf{n} \cdot [\mathbf{F}_H] = \frac{1}{2}((-\mathbf{n} \times \{\mathbf{H}\}) - \mathbf{n} \times (-\mathbf{n} \times \{\mathbf{E}\})), \tag{4}$$

$$\mathbf{n} \cdot [\mathbf{F}_E] = \frac{1}{2}((\mathbf{n} \times \{\mathbf{E}\}) - \mathbf{n} \times (-\mathbf{n} \times \{\mathbf{H}\})), \tag{5}$$

using the notations $\{\mathbf{H}\} = \mathbf{H} - \mathbf{H}^+$ and $\{\mathbf{E}\} = \mathbf{E} - \mathbf{E}^+$ ("+" denoting neighboring data). With the relation for the outward normal vector $\mathbf{n} = -\mathbf{n}^+$, we use the following forms in defining the NekCEM communication kernel:

$$-\mathbf{n} \times \{\mathbf{H}\} = (-\mathbf{n} \times \mathbf{H}) + (-\mathbf{n}^+ \times \mathbf{H}^+), \tag{6}$$

$$-\mathbf{n} \times \{\mathbf{E}\} = (-\mathbf{n} \times \mathbf{E}) + (-\mathbf{n}^+ \times \mathbf{E}^+), \tag{7}$$

where we compute only $(-\mathbf{n} \times \mathbf{H})$ and $(-\mathbf{n} \times \mathbf{E})$ on each element and exchange those data between neighboring elements with an addition operator. Then we complete the surface integration with the remaining operations in (4)–(5).

Discretization In our SEDG formulation, we express each of the six components of $\mathbf{E}$ and $\mathbf{H}$ as tensor product polynomials on the reference element $\mathbf{r} = (r, s, t) \in \hat{\Omega} := [-1, 1]^3$ that maps to the corresponding physical domain $\mathbf{x}(\mathbf{r}) = (x, y, z) \in \Omega^e$ using the Gordon-Hall algorithm (Deville et al. 2002). For example, on a single-element Ω^e, each component is expressed as

$$u^e(\mathbf{x}(\mathbf{r})) = \sum_{k=0}^{N} \sum_{j=0}^{N} \sum_{i=0}^{N} u^e_{ijk} h_i(r) h_j(s) h_k(t). \tag{8}$$

Here, $\{u^e_{ijk}\}$ is the set of basis coefficients, and the functions h_i (including h_j and h_k) are the one-dimensional Lagrange interpolation polynomials associated with the $N+1$ GLL points on $[-1, 1]$. (Note that "t" in (8) is the third coordinate in $\hat{\Omega}$ and should not be confused with the time variable t. The distinction will be clear from context.)

Using the expansion (8) for each of the fields in (2)–(3) and using numerical quadrature, we obtain a system of ordinary differential equations for the basis coefficient vectors $\underline{\mathbf{E}} = (\underline{\mathbf{E}}_x, \underline{\mathbf{E}}_y, \underline{\mathbf{E}}_z)^T$ and $\underline{\mathbf{H}} = (\underline{\mathbf{H}}_x, \underline{\mathbf{H}}_y, \underline{\mathbf{H}}_z)^T$ having the form

$$\epsilon \mathbf{M} \frac{d\underline{\mathbf{E}}}{dt} = \mathbf{C}\underline{\mathbf{H}} + \mathbf{F}\underline{\mathbf{H}}, \quad \mu \mathbf{M} \frac{d\underline{\mathbf{H}}}{dt} = -\mathbf{C}\underline{\mathbf{E}} + \mathbf{F}\underline{\mathbf{E}}, \tag{9}$$

where $\mathbf{M}$ is a diagonal mass matrix, $\mathbf{C}$ is a discrete curl operator, and $\mathbf{F}$ is a flux operator that enforces flux continuity between neighboring elements. This system typically uses

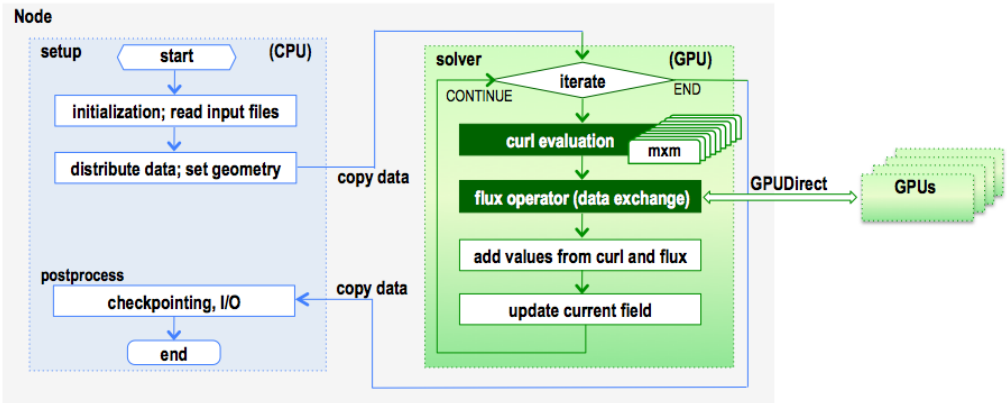

Figure 1. Flowchart for a GPU-enabled NekCEM.

4th-order Runge-Kutta or exponential integrators for time advancement (Min and Fischer 2011). In either case, the central kernel involves repeated evaluation of the matrix-vector products on the right side of (9) for the curl operator $\mathbf{C}$ and near-neighbor data exchanges for the flux operator $\mathbf{F}$. Computing the inverse of the diagonal mass matrices ($\epsilon^{-1}\mathbf{M}^{-1}$ and $\mu^{-1}\mathbf{M}^{-1}$) at insignificant cost and multiplying those to the added values obtained from the curl and flux operators complete the spatial operator evaluations.

Derivatives of u^e in physical space are computed by using the chain rule as

$$\frac{\partial u^e}{\partial x} = \frac{\partial u^e}{\partial r}\frac{\partial r}{\partial x} + \frac{\partial u^e}{\partial s}\frac{\partial s}{\partial x} + \frac{\partial u^e}{\partial t}\frac{\partial t}{\partial x}, \quad (10)$$

$$\frac{\partial u^e}{\partial y} = \frac{\partial u^e}{\partial r}\frac{\partial r}{\partial y} + \frac{\partial u^e}{\partial s}\frac{\partial s}{\partial y} + \frac{\partial u^e}{\partial t}\frac{\partial t}{\partial y}, \quad (11)$$

$$\frac{\partial u^e}{\partial z} = \frac{\partial u^e}{\partial r}\frac{\partial r}{\partial z} + \frac{\partial u^e}{\partial s}\frac{\partial s}{\partial z} + \frac{\partial u^e}{\partial t}\frac{\partial t}{\partial z}. \quad (12)$$

All quantities of interest are evaluated at the GLL point set $\{\xi_i, \xi_j, \xi_k\}$, $i, j, k \in \{0, \ldots, N\}^3$, in which case one has an additional $9n$ memory references and $36n$ operations for the curl operator. The evaluation of each of these derivatives, $\frac{\partial u^e}{\partial r}$, $\frac{\partial u^e}{\partial s}$, $\frac{\partial u^e}{\partial t}$ (simply u^e_r, u^e_s, u^e_t), can be cast as a matrix-matrix product involving the one-dimensional derivative matrix $\hat{D}_{mi} := h'_i(\xi_m) \in \mathbb{R}^{(N+1)\times(N+1)}$ applied to an $(N+1)^3$ block of data and requiring only $2(N+1)^4$ operations per element. In tensor-product form,

$$u^e_r \equiv D_r u^e = (I \otimes I \otimes \hat{D})u^e = \Sigma_{l=0}^N \hat{D}_{il} u^e_{qjk}, \quad (13)$$

$$u^e_s \equiv D_s u^e = (I \otimes \hat{D} \otimes I)u^e = \Sigma_{l=0}^N \hat{D}_{jl} u^e_{ilk}, \quad (14)$$

$$u^e_t \equiv D_t u^e = (\hat{D} \otimes I \otimes I)u^e = \Sigma_{l=0}^N \hat{D}_{kl} u^e_{ijl}, \quad (15)$$

where $I \in \mathbb{R}^{(N+1)\times(N+1)}$ is the identity matrix. Differentiation with respect to r, s, and t is the most compute-intensive phase in advancing (9). All other steps involve only $O(N^3)$ or $O(N^2)$ work and storage per element. For (9), there are 12 derivatives in the curl operator in the physical domain, $\frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z}$, $\frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x}$, $\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}$, $\frac{\partial E_z}{\partial y} - \frac{\partial E_y}{\partial z}$, $\frac{\partial E_x}{\partial z} - \frac{\partial E_z}{\partial x}$, and $\frac{\partial E_y}{\partial x} - \frac{\partial E_x}{\partial y}$, involving 18 derivatives to be computed in the reference domain using (13)–(15)—three each for each of three components for $\underline{\mathbf{E}}$ and $\underline{\mathbf{H}}$. The weak form is completed by evaluating the inner product of the curl operators with the test functions, which is effected by pointwise multiplication of the integrand with the quadrature weights $w_{ijk} = \hat{w}(\xi_i)\hat{w}(\xi_j)\hat{w}(\xi_k)$, where $\hat{w}(\xi_i)$ are the $N+1$ GLL weights on $[-1, 1]$.

The surface flux terms $\mathbf{F}(\underline{\mathbf{H}}, \underline{\mathbf{E}})$ are sparse because of the use of GLL quadrature, but they do require face-to-face data exchanges between adjacent elements. The exchanges involve $(N+1)^2$ words per face for each of the six components.

OpenACC Implementation

Time advancement of (9) affords significant opportunities for GPU-based parallelism. In this section, we describe our OpenACC implementation for thread-based parallelism of the essential kernels of the code, including the discrete curl evaluation and the tuned gather-scatter kernel for the nearest-neighbor data exchanges. Throughout the section, we demonstrate pseudocodes that we used for performance studies in the Algorithms, while describing pseudocodes of older versions in the text. All operations are in double precision.

Data Movement

In our OpenACC implementation, we assume one MPI rank per GPU. Each MPI rank is pinned to a dedicated CPU core that manages execution of its corresponding GPU device. The CPU host initiates data transfer, sends code, passes arguments to the region, waits for completion, and transfers GPU results back to the CPU host. Figure 1 illustrates the flowchart of our GPU-enabled version of NekCEM using GPUDirect communication. The code begins with MPI initialization on CPUs. MPI rank 0 reads the mesh data, boundaries, connectivities, and parameters and redistributes the data across the ranks on the fly. Each CPU core sets geometric variables on the distributed mesh data and assigns the field values at an initial time. Because GPU memory is separate from CPU memory, the host CPU cores initiate all data movement to the corresponding GPUs, a task that is managed by the compiler according to OpenACC directives. Before iterating our timestepping loop, we specify `DATA COPYIN` for all the repeatedly used data to be moved from the host CPU to the corresponding GPU. We specify `DATA CREATE` for GPU-local arrays. The pseudocode in Algorithm 1 demonstrates some of the OpenACC procedures with the notation `D=` $\hat{D}$ for the derivative operator and `w=` w_{ijk} for the quadrature weights. Here, `rx`, `ry`,..., `tz`

Algorithm 1 Data copy.

```
#ifdef _OPENACC
!$ACC  DATA COPYIN(D,w,H,E)
!$ACC&      COPYIN(rx,ry,rz,sx,sy,sz,tx,ty,tz)
!$ACC&      CREATE(u1r,u2r,u3r,u1s,u2s,u3s,u1t,u2t,u3t)
#endif
```

are time-invariant variables that hold the product of the Jacobian (J) and the metrics, $\frac{\partial r}{\partial x}, \frac{\partial r}{\partial y}, \ldots, \frac{\partial t}{\partial z}$, required to map derivatives from the reference (r, s, t) domain to the physical space (x, y, z) by using the chain rule.

Curl Evaluation

Note that we never explicitly form the matrix of the discretized curl operator $\mathbf{C}$. The requisite matrix-matrix products (mxm) are instead effected through direct operator evaluation based on Eqs. (13)–(15). Algorithm 2 demonstrates a pseudocode for the subroutine `local_curl_grad3_acc` that computes the derivatives of (`u1`,`u2`,`u3`):=`E` and `H` on the reference domain. OpenACC parallelism is split into gangs (thread blocks), workers (warps), and vector elements (threads). OpenACC threads use a hierarchy of memory spaces. Each thread has its own private local memory, and each gang of workers of threads has shared memory visible to all threads of the gang. All OpenACC threads running on a GPU have access to the same global memory. Global memory on the accelerator is accessible to the host CPU (Cray Inc. 2012). In Algorithm 2, the OpenACC directives `COLLAPSE` instruct the compiler to collapse the quadruply nested loop into a single loop and then to chunk up that loop into the most efficient parallel decomposition across the streaming multiprocessors (`GANG`), collection of warps (`WORKER`), and single-instruction multithreading (SIMT) dimension (`VECTOR`). The collapsed outer loop is divided into multiparallel chunks with SIMT vectors of length 128, where the default vector size is 128 because the Kepler architecture has quad warp scheduler allowing four warps (32 threads/warp) simultaneously. It is critical that the innermost loop be performed in scalar mode. This allows the nine scalar accumulators to be relegated to registers, thus minimizing the data movement. In this routine we specify `!dir$ NOBLOCKING` (Cray compiler-specific Fortran directive) in the OpenACC directive to stop the compiler from performing cache blocking on the multinested loop. At this point in the compiler's optimization, the compiler does not know it is targeting an accelerator, and it is thus blocking for effective cache utilization. This blocking reduces the vector length on the accelerator; hence, this compiler optimization should be inhibited.

We note that although portability is supported between different compilers, the CCE (Cray Compiler Environment) and PGI (The Portland Group, Inc.) compilers may deliver different performance. One must pay attention to the system environment or compiler-dependent directives in order to get the best performance. As an example, Table 1 demonstrates the accumulated "ptime" timings of 10^4 repeated computations of the `local_curl_grad3_acc` routine from different versions, using 1 GPU on Titan for $n = 125 \cdot 15^3$. The operation count for this routine is

Algorithm 2 Final tuned derivative operations.

local_curl_grad3_acc (Version 1)

```
      ptime=dclock()
!$ACC DATA PRESENT(u1r,u1s,u1t,u2r,u2s,u2t,u3r,u3s,u3t)
!$ACC&     PRESENT(u1,u2,u3,D)
!$ACC PARALLEL LOOP COLLAPSE(4) GANG WORKER VECTOR
!$ACC&     private(tmp1_r,tmp2_r,tmp3_r)
!$ACC&     private(tmp1_s,tmp2_s,tmp3_s)
!$ACC&     private(tmp1_t,tmp2_t,tmp3_t)
!dir$ NOBLOCKING
      do e = 1,E
         do k = 1,N+1
         do j = 1,N+1
         do i = 1,N+1
            tmp1_r = 0.0
            tmp2_r = 0.0
            tmp3_r = 0.0
            tmp1_s = 0.0
            tmp2_s = 0.0
            tmp3_s = 0.0
            tmp1_t = 0.0
            tmp2_t = 0.0
            tmp3_t = 0.0
!$ACC LOOP SEQ
            do l=1,N+1
               tmp1_r = tmp1_r + D(i,l) * u1(l,j,k,e)
               tmp2_r = tmp2_r + D(i,l) * u2(l,j,k,e)
               tmp3_r = tmp3_r + D(i,l) * u3(l,j,k,e)
               tmp1_s = tmp1_s + D(j,l) * u1(i,l,k,e)
               tmp2_s = tmp2_s + D(j,l) * u2(i,l,k,e)
               tmp3_s = tmp3_s + D(j,l) * u3(i,l,k,e)
               tmp1_t = tmp1_t + D(k,l) * u1(i,j,l,e)
               tmp2_t = tmp2_t + D(k,l) * u2(i,j,l,e)
               tmp3_t = tmp3_t + D(k,l) * u3(i,j,l,e)
            enddo
            u1r(i,j,k,e) = tmp1_r
            u2r(i,j,k,e) = tmp2_r
            u3r(i,j,k,e) = tmp3_r
            u1s(i,j,k,e) = tmp1_s
            u2s(i,j,k,e) = tmp2_s
            u3s(i,j,k,e) = tmp3_s
            u1t(i,j,k,e) = tmp1_t
            u2t(i,j,k,e) = tmp2_t
            u3t(i,j,k,e) = tmp3_t
         enddo
         enddo
         enddo
      enddo
!$ACC END PARALLEL LOOP
!$ACC END DATA
      ptime=dclock()-ptime
```

Table 1. Accumulated (10,000 repeated) timings with GFlops for CCE and PGI compilers for $n = E(N+1)^3$ with $E = 125$ and $N = 14$ on 1 GPU.

local_curl_grad3_acc	CCE		PGI	
	sec	GFlops	sec	GFlops
Version 1	8.00	142	9.47	120
Version 2	16.76	68	9.00	126
Version 3	8.00	142	7.84	145

$18(N+1)n \times 10^4$. For the routine in Algorithm 2, referred to as Version 1, CCE and PGI take 8.00 sec and 9.47 sec with 142 GFlops and 120 GFlops, respectively. During these measures we unload the CrayPat module (performance analysis tool on Titan) on the system to avoid potential slowdown.

The timings for `ACC DATA PRESENT` are negligible, with $\sim 10^{-4}$ sec for both compilers. On the other hand, our `local_curl_grad3_acc` routine using OpenACC directives `KERNELS` and `LOOP COLLAPSE` (shown as Version 2 below) takes 16.76 sec at 68 GFlops with the CCE compiler while taking 9.00 sec at 120 GFlops with PGI.

```
local_curl_grad3_acc (Version 2)

     ptime = dclock()
!$ACC DATA PRESENT(u1r,u1s,u1t,u2r,u2s,u2t,u3r,u3s,u3t)
!$ACC&     PRESENT(u1,u2,u3,D)
!$ACC KERNELS
      do e = 1, E
!$ACC LOOP COLLAPSE(3)
         do k = 1,N+1
         do j = 1,N+1
         do i = 1,N+1
            tmp1_r = 0.0
            tmp2_r = 0.0
            tmp3_r = 0.0
            tmp1_s = 0.0
            tmp2_s = 0.0
            tmp3_s = 0.0
            tmp1_t = 0.0
            tmp2_t = 0.0
            tmp3_t = 0.0
!ACC LOOP SEQ
            do l=1,N+1
               tmp1_r = tmp1_r + D(i,l) * u1(l,j,k,e)
               tmp2_r = tmp2_r + D(i,l) * u2(l,j,k,e)
               tmp3_r = tmp3_r + D(i,l) * u3(l,j,k,e)
               tmp1_s = tmp1_s + D(j,l) * u1(i,l,k,e)
               tmp2_s = tmp2_s + D(j,l) * u2(i,l,k,e)
               tmp3_s = tmp3_s + D(j,l) * u3(i,l,k,e)
               tmp1_t = tmp1_t + D(k,l) * u1(i,j,l,e)
               tmp2_t = tmp2_t + D(k,l) * u2(i,j,l,e)
               tmp3_t = tmp3_t + D(k,l) * u3(i,j,l,e)
            enddo
            u1r(i,j,k,e) = tmp1_r
            u2r(i,j,k,e) = tmp2_r
            u3r(i,j,k,e) = tmp3_r
            u1s(i,j,k,e) = tmp1_s
            u2s(i,j,k,e) = tmp2_s
            u3s(i,j,k,e) = tmp3_s
            u1t(i,j,k,e) = tmp1_t
            u2t(i,j,k,e) = tmp2_t
            u3t(i,j,k,e) = tmp3_t
         enddo
         enddo
         enddo
      enddo
!$ACC KERNELS
!$ACC END DATA
      ptime=dclock()-ptime
```

A third version (not shown), identical to Version 1 save that

```
!\$ACC PARALLEL LOOP COLLAPSE(4) GANG WORKER VECTOR
```

is replaced by,

```
!\$ACC PARALLEL LOOP COLLAPSE(4)
```

takes 8.00 sec at 142 GFlops with CCE and 7.84 sec at 145 GFlops with PGI. In the remaining sections, all computational results are based on Version 1 of `local_curl_grad3_acc`, shown in Algorithm 2. We used the CCE compiler if the system supported CCE.

The routine `curl_acc` in Algorithm 3 completes the curl operation in the physical domain through application of the chain rule and the quadrature weights. The OpenACC directives are applied in a manner similar to that in Algorithm 2.

Communication Kernel

We discuss here our tuned gather-scatter kernel, shown in Algorithm 4, using vectorization, streaming, and GPUDirect. Figure 2 demonstrates a schematic of data communication in our SEDG framework, using a two-dimensional mesh for simplicity. The mesh is distributed on two nodes, having two elements on each node and four grid points on each element face. The figure demonstrates the local data within each node and the nonlocal data between nodes. Unlike the continuous Galerkin approach, which has communication patterns induced by face-to-face, vertex-to-vertex, and edge-to-edge exchanges, the SEDG method requires only face-to-face data exchanges. Each grid point is assigned a single identifying integer, referred to as a global number. Grid points on a face shared by different elements, representing the same physical coordinates, have the same global number. The NekCEM gather-scatter library `gs_lib` provides a black-box user interface that requires only this global numbering.

Algorithm 3 Final tuned curl volume integration operations.

curl_acc

```
!$ACC DATA PRESENT(u1r,u1s,u1t,u2r,u2s,u2t,u3r,u3s,u3t)
!$ACC&     PRESENT(w,rx,ry,rz,sx,sy,sz,tx,ty,tz)
!$ACC&     PRESENT(curlU_x,curlU_y,curlU_z)
!$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
      curlU_x(i,j,k,e)
           = (  u3r(i,j,k,e) * ry(i,j,k,e)
              + u3s(i,j,k,e) * sy(i,j,k,e)
              + u3t(i,j,k,e) * ty(i,j,k,e)
              - u2r(i,j,k,e) * rz(i,j,k,e)
              - u2s(i,j,k,e) * sz(i,j,k,e)
              - u2t(i,j,k,e) * tz(i,j,k,e) ) * w(i,j,k)

      curlU_y(i,j,k,e)
           = (  u1r(i,j,k,e) * rz(i,j,k,e)
              + u1s(i,j,k,e) * sz(i,j,k,e)
              + u1t(i,j,k,e) * tz(i,j,k,e)
              - u3r(i,j,k,e) * rx(i,j,k,e)
              - u3s(i,j,k,e) * sx(i,j,k,e)
              - u3t(i,j,k,e) * tx(i,j,k,e) ) * w(i,j,k)

      curlU_z(i,j,k,e)
           = (  u2r(i,j,k,e) * rx(i,j,k,e)
              + u2s(i,j,k,e) * sx(i,j,k,e)
              + u2t(i,j,k,e) * tx(i,j,k,e)
              - u1r(i,j,k,e) * ry(i,j,k,e)
              - u1s(i,j,k,e) * sy(i,j,k,e)
              - u1t(i,j,k,e) * ty(i,j,k,e) ) * w(i,j,k)
   enddo
   enddo
   enddo
enddo
!$ACC END PARALLEL LOOP
!$ACC END DATA
```

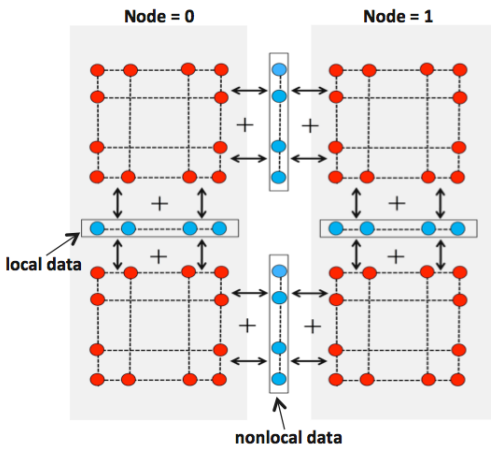

Figure 2. Data pattern in SEDG framework: nonlocal data exchange between nodes and local data within a node.

Vectorization These global numbers represent a map of the data's connectivity; the field values on the grid points with the same global number need to communicate with each other. In order to efficiently carry out this communication, the global numbers are translated into a map, which is constructed by listing groups of connected grid points terminated by a -1. Our original gather-scatter kernel, written for the CPU, was based on WHILE loops that search for the terminator. This is shown below for the local-gather operation.

```
for(k=0;k<vn;k++){
  while ((i=*map++)!=-1){
    t=u[i+k*dstride];
    j=*map++;
    do {
      t += dbuf[j*vn];
    } while((j=*map++)!=-1);
    u[i+k*dstride]=t;
  }
  dbuf++;
}
```

Because we have six field (vn$=6$) components, describing electric and magnetic fields, sharing the same data communication pattern on the same mesh geometry using the same map, we store the values of $-\mathbf{n}\times\mathbf{H}$ and $-\mathbf{n}\times\mathbf{E}$ in (4)–(5) defined on the faces of all the elements on each node into a single array (u) with the length of (dstride) for each field component. This local-gather operation can be equivalently rewritten as follows, using for loops, which was the first step in the transition to our tuned routines.

```
for(k=0;k<vn;++k){
  for(i=0;map[i]!=-1;i=j+1){
    t = u[map[i]+k*dstride];
    for(j=i+1;map[j]!=-1;j++){
      t += u[map[j]+k*dstride];
    }
    u[map[i]+k*dstride] = t;
  }
}
```

The compilers are unable to vectorize the first for loop nest because of using complex pointer arithmetic and loops with an unknown length at the time of the loop start. The second for loop-nest removes the troublesome pointer arithmetic, but the unknown length still remains. In our GPU-enabled Version, we precompute an additional array that stores extra information about the map, including the index of the start of a group and the length of a group. Our full communication kernel uses four maps (map,snd_map, rcv_map, and t_map), provided with the number of groups in each map (m_nt, snd_m_nt, rcv_m_nt, t_m_nt), and their corresponding index arrays (mapf, snd_mapf, rcv_mapf, t_mapf). For each map, the starting index of a group and the length of the associated group are stored at mapf[i*2] and mapf[i*2+1] ($i=0,1,...$), respectively. As an example, an arbitrary map representing connectivity is defined as

```
map[0:12]=[1, 8, 5, -1, 11, 6, -1, 32, 9, 17, 24, -1, -1].
```

The data corresponding to the group {1,8,5} are connected, and the total number of groups is m_nt$=3$. The two instances of "-1 -1" indicate the end of the map. In this case, an index array for this map is

```
mapf=[0, 2, 4, 1, 7, 3],
```

where mapf[0]$=0$, mapf[2]$=4$, and mapf[4]$=7$ indicate the starting index of the groups in map[0:12], and mapf[1]$=2$, mapf[3]$=1$, and mapf[5]$=3$ indicate the length of the groups in map[0:12], subtracted by one. We note that map and t_map are related by a transpose, as are snd_map and rcv_map.

Algorithm 4 shows a pseudocode of our tuned gather-scatter kernel fgs_fields_acc using the map and index arrays for the local-gather (map, mapf), global-scatter (snd_map, snd_mapf), global-gather (rcv_map, rcv_mapf), and local-scatter (t_map, t_mapf) procedures. While the tuned gather-scatter kernel and the original WHILE-based kernel have similar levels of performance on the CPU, the tuned kernel is almost 100× faster than the original on the GPU. The i loop is large and allows for efficient use of the massive parallelism of the accelerator.

Streaming We stream the six field vectors in the array (u) as shown in Algorithm 4. Note that each parallel loop has an async(k+1) that allows all instances of the for loop to run in parallel, since they are independent. An #pragma acc wait is then used to synchronize all the streams at the end of the loop, and then the next map is similarly streamed. This approach allows multiple kernels to be scheduled in order to amortize the latency of fetching the operands in the kernels. When one stream stalls waiting for an operand, another stream can proceed.

GPUDirect Gather-Scatter The gather routine involves an addition operation, and the scatter routine involves a copy operation. In the NekCEM gs_lib, local data are first gathered within each node. The nonlocal exchange is then effected by the standard MPI_Irecv(), MPI_Isend(), MPI_Waitall() sequence, in order to allocate receive buffers prior to sending data. After the MPI_Waitall, the received data is scattered to the elements.

MPICH2 allows GPU memory buffers to be passed directly to MPI function calls, eliminating GPU data copy to the host before passing data to MPI. Thus we use the normal MPI_Isend/MPI_Irecv functions for internode data transfer directly from/to the GPU memory using OpenACC directives #pragma acc host_data use_device to enable the GPUDirect. When the systems do not support GPUDirect, we update the host with asynchronous GPU data copies with pinned host memory by specifying async, as shown in Algorithm 4. In a later section, we demonstrate the performance on GPUs for both GPUDirect and GPUDirect-disabled cases in comparison with that on CPUs.

We note that in this newly tuned gather-scatter kernel, we have maintained the black-box nature of the communication routine. Users need to call just a single Fortran interface gs_op_fields. We have a runtime check acc_is_present that determines whether to call the GPU-enabled fgs_fields_acc or CPU-only fgs_fields kernel. If fgs_fields_acc is called, either GPUDirect or GPUDirect-disabled is chosen by a user-determined preprocessor compiler option for USE_GPU_DIRECT.

Before we ported our newly tuned gather-scatter kernel to OpenACC, we had the local-gather operations computed on GPUs, as shown below; moved the nonlocal data ug2 in the size of nloc back to the host CPU; and used our original CPU-only communication kernel gs_op_fields that computes the global-gather and global-scatter procedures

(using the WHILE-based loop on CPU) and performs CPU-to-CPU MPI communication. Then we copied the updated ug2 back to the GPU device and proceeded with the local-scatter operation.

```
      do k = 0,vn-1
!$ACC PARALLEL LOOP GANG VECTOR ASYNC(k+1)
        do i=1,nglobal
!$ACC LOOP SEQ
          do j = ids_ptr(i),ids_ptr(i+1)-1
             il= ids_lgl1(j)
             sil = k*n+il
             if (i.le.nloc ) then
                 sig = k*nloc+i
                 ug2(sig) = ug2(sig)+u(sil)
             else
                 sig = k*n+i
                 ug (sig) = ug (sig)+u(sil)
             endif
          enddo
        enddo
!$ACC UPDATE HOST(ug2(k*nloc+1:(k+1)*nloc)) ASYNC(k+1)
      enddo
!$ACC WAIT

      call gs_op_fields(gsh_face_acc,ug2,nloc,vn,1,1,0)

      do k = 0,vn-1
!$ACC UPDATE DEVICE(ug2(k*nloc+1:(k+1)*nloc)) ASYNC(k+1)
!$ACC PARALLEL LOOP GANG VECTOR ASYNC(k+1)
        do i=1,nglobal
!$ACC LOOP SEQ
          do j = ids_ptr(i),ids_ptr(i+1)-1
            il = ids_lgl1(j)
            sil = k*n+il
            if (i.le.nloc ) then
                sig   = k*nloc+i
                u(sil)= ug2(sig)
            else
                sig   = k*n+i
                u(sil)= ug(sig)
            endif
          enddo
        enddo
      enddo
!$ACC WAIT
!$ACC END DATA
```

While this version is a tuned version of Gong's previous work (Markidis et al. 2015), it excludes accelerating global-scatter and global-gather operations on the GPU and does not support GPU-to-GPU MPI communication. In our new kernel, we moved the local-gather and local-scatter operations shown above into fgs_fields_acc as well as added OpenACC to the global-scatter and global-gather operations in a highly tuned form so that the full procedures involving local/global gather/scatter operations all can be performed on the GPU with GPU-to-GPU MPI_Irec and MPI_Isend, while still providing a single black-box user interface with gs_op_fields for either the GPU or CPU runs.

Performance and Analysis

Table 2 demonstrates an overview of the systems used for our performance tests. We note that Tesla and Maud are local/institutional machines that were used for development and baseline experiments. We present performance results for the GPU- and CPU-based SEDG Maxwell solvers. We also analyze their scalability and compare their relative efficiencies under several different metrics including power consumption. Timing runs were performed on the platforms shown in Table 2 with E elements of polynomial order $N = 7$ or $N = 14$. Timings are for 1,000 timesteps with all I/O turned off during those steps.

Algorithm 4 A pseudocode: gather-scatter kernel.

fgs_fields_acc

```
//* (1) local gather *//
    for(k=0;k<vn;++k){
   #pragma acc parallel loop gang vector async(k+1)
        for(i=0;i<m_nt;i++){
          t = u[map[mapf[i*2]]+k*dstride];
   #pragma acc loop seq
          for(j=0;j<mapf[i*2+1];j++) {
            t += u[map[mapf[i*2]+j+1]+k*dstride];
          }
          u[map[mapf[i*2]]+k*dstride] = t;
        }
    }
   #pragma acc wait

    //* multiple messages multiple destinations *//
    MPI_Irecv(rbuf,len,DATATYPE,source,tag,comm,req);

//* (2) global scatter *//
        for(k=0;k<vn;++k) {
   #pragma acc parallel loop gang vector async(k+1)
          for(i=0;i<snd_m_nt;i++){
   #pragma acc loop seq
            for(j=0;j<snd_mapf[i*2+1];j++) {
              sbuf[k+snd_map[snd_mapf[i*2]+j+1]*vn]
                  = u[snd_map[snd_mapf[i*2]]+k*dstride];
            }
          }
        }
   #pragma acc wait

#if USE_GPU_DIRECT
   #pragma acc host_data use_device(sbuf)
    //* multiple messages multiple destinations *//
    MPI_Isend(sbuf,len,DATATYPE,dest,tag,comm,req);
    MPI_Waitall(len,req,status);
#else
   #pragma acc update host(sbuf[0:bl]) async
   #pragma acc wait
    //* multiple messages multiple destinations *//
    MPI_Isend(sbuf,len,DATATYPE,dest,tag,comm,req);
    MPI_Waitall(len,req,status);
   #pragma acc update device(sbuf[0:bl]) async
   #pragma acc wait
#endif

//* (3) global gather *//
        for(k=0;k<vn;++k) {
   #pragma acc parallel loop gang vector async(k+1)
          for(i=0;i<rcv_m_nt;i++){
   #pragma acc loop seq
            for(j=0;j<rcv_mapf[i*2+1];j++) {
              u[rcv_map[rcv_mapf[i*2]]+k*dstride] + =
                  rbuf[k+rcv_map[rcv_mapf[i*2]+j+1]*vn];
            }
          }
        }
   #pragma acc wait

//* (4) local scatter *//
      for(k=0;k<vn;++k) {
   #pragma acc parallel loop gang vector async(k+1)
        for(i=0;i<t_m_nt;i++){
          t = u[t_map[t_mapf[i*2]]+k*dstride];
   #pragma acc loop seq
          for(j=0;j<t_mapf[i*2+1];j++) {
            u[t_map[t_mapf[i*2]+j+1]+k*dstride] = t;
          }
        }
      }
   #pragma acc wait
```

For all but the smallest single-node cases the domain is a tensor-product of elements with periodic boundary conditions. (The smallest cases are just a line of elements.) Our domains are typically partitioned with recursive spectral bisection. We did not monitor the number of connected ranks for each MPI rank but it can be higher than six if the partition

Table 2. Systems Overview

	Cray XK7 Titan	IBM BG/Q Vesta	Tesla	Maud
Processors	AMD Opteron	PowerPC A2	Intel Xeon	Intel Xeon
Nodes #	18,688	2048	2	2
CPU cores #	299,008	32,768	32	32
	16 cores/node	16 cores/node	16 cores/node	16 cores/node
CPU clock rate	2.2 GHz	1.6 GHz	2.0 GHz	3.4 GHz
	Nvidia K20X		Nvidia K20c	Nvidia K40m
GPUs #	18,688	–	2	2
	1 GPU/node	–	1 GPU/node	1 GPU/node
GPU cores # per node	2,688	–	2,496	2,880
GPU clock rate	732 MHz		706 MHz	745 MHz
DRAM Bandwidth	51.2 GB/s	30 GB/s	51.2 GB/s	59.7 GB/s
GRAM Bandwidth	250 GB/s	–	208 GB/s	288 GB/s
Peak (max)	27 PF	419 TF	512 GF	870 GF
Interconnect	3D torus	5D torus	Direct	Direct

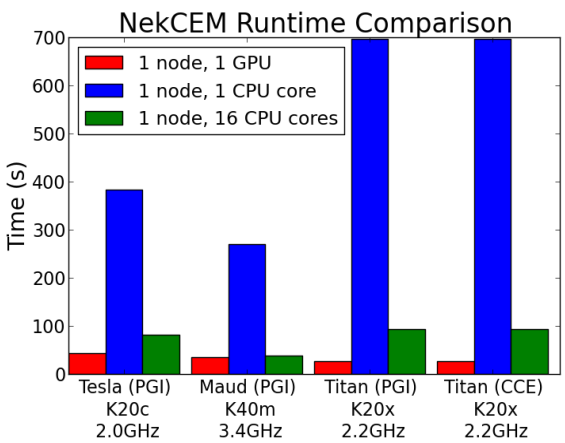

Figure 3. Timings on different systems on 1 node using Cray CCE and PGI compilers; $n = 125...15^3$ with 1,000 timesteps.

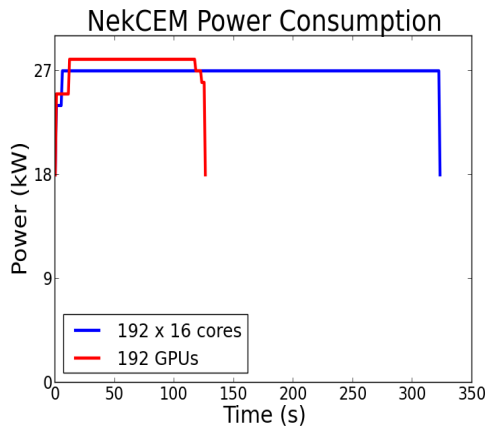

Figure 4. Power consumption on Titan using 2 cabinets (96 nodes per cabinet) with 18 kW when the system is idle; 1,000 timestep runs with $n = 216M$ ($E = 40^3$ and $N = 14$).

is a bit irregular. We note that NekCEM is designed for unstructured domains, and thus no attempt has been made to optimize for the stuctured domains used in the timing tests.

Figure 3 shows timings when E=125 elements of order N=14 ($n = 125 \cdot 15^3$). The bars contrast 1- and 16-core results with those based on a single GPU. The GPU time is lower than that for the 16-core case across all platforms. The small systems (Tesla/Maud) support only the PGI compiler (no GPUDirect support). We note that Maud does not show significant speedup on the K40m GPU device in comparison with its 16 cores, which run at 3.5 GHz. From a performance perspective, the choice between a GPU or 16 cores on Maud appears to be a tie that would need to be broken by price and power considerations. On Titan, however, the GPU speedup is significant, with a single GPU outperforming 16 cores by a factor of 2.5.

To quantify the energy consumption of the CPU and GPU versions of NekCEM, we collaborated with staff from the Oak Ridge Leadership Computing Facility (OLCF) to obtain power data from application runs on Titan. Specifically, we ran a set of two-cabinet jobs $(E, N) = (40^3, 14)$ ($n = 216$ million), while collecting energy use data for the CPU and GPU runs. Figure 4 shows the power consumption for each of the two 192-node runs: one using 192 GPUs and the other using 192×16 CPU cores. As seen in the figure, the GPU run draws slightly more power during computation than the CPU run does. Because the GPU finishes the computation faster, however, it requires only 39% of the energy needed for 16 CPUs to do the same computation. We note that both the GPU and the CPU runs are burdened with power load from the unused resources, which makes it difficult to accurately assess the impact of these results for other node architectures having no attached GPU or fewer CPU cores. Nonetheless, we note that the GPU runs do incur a power savings on Titan.

Next, we consider parallel performance for varying problem sizes n and processor counts P (the number of GPUs or CPU cores). Figures 5–6 show runtimes for varying problem sizes n on OLCF's Titan and on the BG/Q "Vesta" at the Argonne Leadership Computing Facility (ALCF). Graph (a) in each of the figures is GPU-based with MPI to communicate using GPUDirect. Graphs (b) and (c) are based on all-CPU implementations using MPI to communicate between the P cores, with $P = 1, 2, 4, \ldots, 128$.

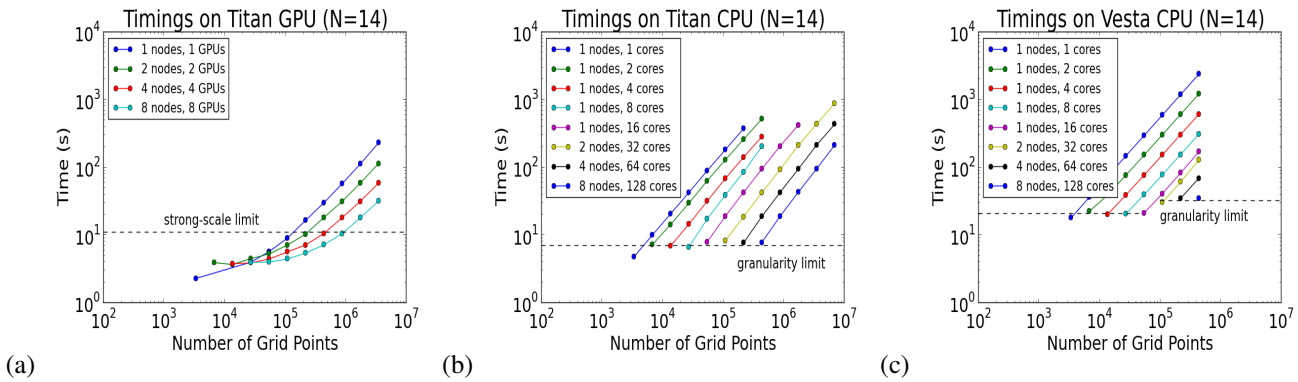

Figure 5. Timings on different number of GPUs and CPU cores on OLCF Titan and ALCF BG/Q Vesta; 1,000 timestep runs with the number of grid points $n = E(N+1)^3$ increased with $E = 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048$ and $N = 14$.

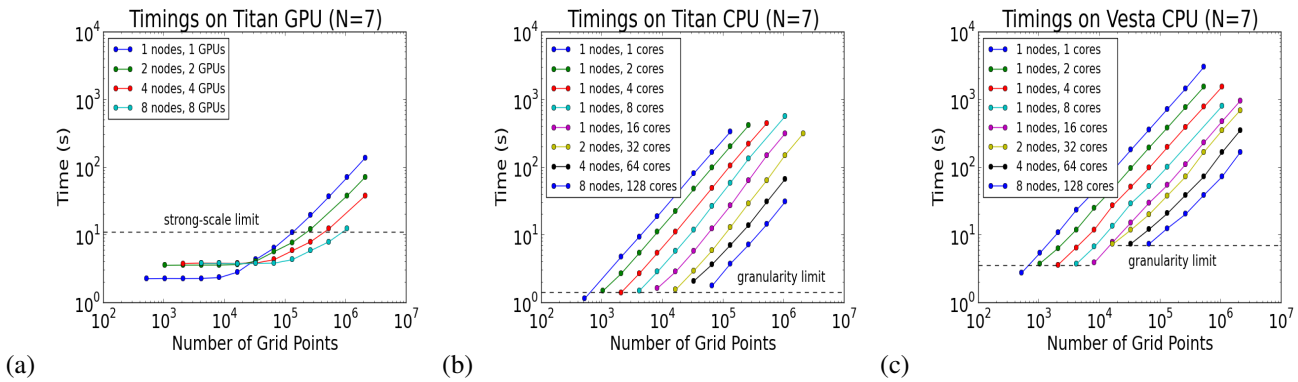

Figure 6. Timings on different number of GPUs and CPU cores on OLCF Titan and ALCF BG/Q Vesta; 1,000 timestep runs with the number of grid points $n = E(N+1)^3$ increased with $E = 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096$ and $N = 7$.

To understand the data in Figures 5 and 6, we first note that a *column* of dots corresponds to classic strong scaling. As one moves down a vertical line, the problem size n is fixed, and P doubles from curve to curve. On the other hand, a *row* of dots corresponds to weak scaling. Starting from the left and moving right, the processor counts and problem sizes double. If the points are on the same horizontal line (i.e., have the same runtime), then we have perfect weak scaling, as is the case for $n/P > 10^5$ in Figure 5(a). We see that the largest problems (large, fixed, n) realize a twofold reduction in solution time with each doubling of processor count over the range of P considered. We have indicated a *strong-scale limit* line where the strong-scale performance starts to fall off when n/P is too small. For the GPU-only (i.e., $P = 1$) case, the strong-scale limit corresponds to $n/P \approx 10^5$ for both the $N = 7$ and $N = 14$ cases. We note that the strong-scale limit is not observed for the CPU-only cases (graphs (b) and (c) in Figures 5–6). They continue to have good strong scaling down to the point of one element per core, which is the natural granularity limit for the SEDG formulation. Their lower-bound runtime is thus set by the granularity-limit line indicated in the plots.

We now focus on the single-GPU performance in Figures 5(a) and 6(a), which involves no interprocessor communication. Moving from right to left on the blue (GPU 1) curve, the number of elements decreases by a factor of 2 for each point. The time decreases linearly until $n \approx 10^5$, at which point there is insufficient work to saturate the GPU work queue. Thus, even in the absence of communication,

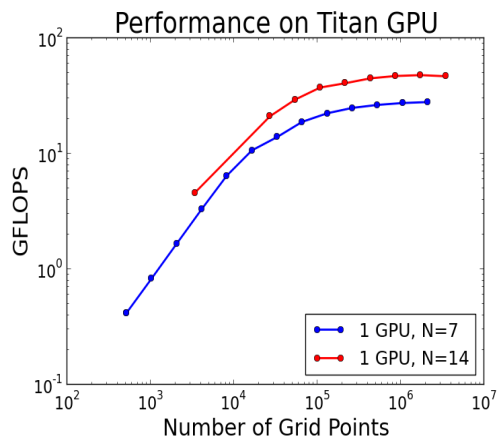

Figure 7. Single-GPU performance on OLCF Titan, based on 1000 timestep runs with number of grid points $n = E(N+1)^3$ for $N = 7$ and 14, and varying E.

we can recognize that $n/P \approx 10^5$ is an approximate lower bound for effective utilization of the GPU. If we attempt to strong-scale the problem at this size by increasing P from 1 to 2, say, then *each* GPU is moving away from its saturated performance state because the local problem size is reduced. To further quantify this behavior, we plot Gflop/s vs n for the single-GPU case in Figure 7. An estimate of the flops rate is derived from (9) and (10)–(15). The present simulations use a 5-stage Runge-Kutta (RK) integrator to advance (9).

Accounting for all six fields and for the few vector-vector operations associated with each RK stage, the work per step for E elements of order N is

$$\begin{aligned} W &= 5 \times \left[36(N+1)^4 + 84(N+1)^3\right] E \\ &= \left[180(N+1) + 420\right] n. \end{aligned} \tag{16}$$

For example, the work per timestep is $W \approx 3120n$ for $N = 14$ and $W \approx 1860n$ for $N = 7$. This estimate is conservative in that it does not account for the additional $O(N+1)^2$ terms associated with the flux-exchanges at the faces. For the single-GPU case in Figure 7 with $N = 7$ and $n = 4096 \times (8)^3$, the wall-clock time for 1000 steps is 140 seconds, which corresponds to a saturated peak of $\approx$ 28 Gflops. For the same case with $N = 14$ and $n = 1024 \times (15)^3$, the wall-clock time is 230 seconds, corresponding to 48 Gflops. The $n_{1/2}$ values where the code realizes half of these peak values are $n_{1/2}$=33,308 and 34,525, respectively, for N=7 and 14. The problem sizes must be large on each GPU to get close to the (application-specific) realizeable peak. By contrast, no such performance drop off (as $n/P \longrightarrow 0$) is observed in the single-core CPU curves of Figures 5(b)-(c) or 6(b)-(c).

The multi-GPU cases for $N = 14$ and 7 exhibit slightly different behaviors. For $N = 14$, each GPU has essentially the same performance as the single-GPU case does. That is, aside from the smallest value of n, each curve in Figure 5(a) could be shifted to the left by a factor of P (corresponding to the horizontal axis being n/P instead of n), and the curves would be nearly on top of each other. This situation indicates that communication overhead is *not* a significant factor in the performance drop-off for the small-n limit. Rather, as is evident from the $P = 1$ curve, it is the standard vector start-up cost limitation that mandates the use of relatively large problem sizes for the multi-GPU case. As is the case with classic vector architectures, this overhead can be characterized by $n_{1/2}$, the value of n required to reach half of the peak performance rate for this problem. A large $n_{1/2}$ limits the amount of strong scaling that can be extracted from a particular problem of fixed size, n. We note that although communication overhead is absent in the $N = 14$ case, it is evident in the $N = 7$ case (Figure 6(a)) where all the multi-GPU curves are shifted above the GPU 1 curve in the small-n limit. The importance of communication in this case follows from the relative reduction in local workload, which is a factor of $(15/8)^3 \approx 8$ times smaller than in the $N = 14$ case.

We next turn to the CPU results in Figures 5(b)–(c) and 6(b)–(c) on Titan and Vesta. Here, we see generally good weak-scaling behavior except that, in each case, the single-core ($P = 1$) case is faster than the corresponding multicore cases because of the absence of communication and lack of contention for local memory resources. Also, on Vesta, there is a significant step up moving from a single node ($P \leq 16$) to more than one node because of the added overhead of internode communication. Aside from these one-time step-ups, the strong scaling continues to be reasonable right up to the fine-grained limit of $E/P = 1$. Thus, from an efficiency standpoint, no penalty is incurred for running with E/P=1 on P CPU cores. In particular, we note in comparing Figures 6(a) and (b) that the all-CPU variant of Titan can run the $n = 10^5$ faster than a multi-GPU case. At this granularity limit, the all-CPU approach is fastest.

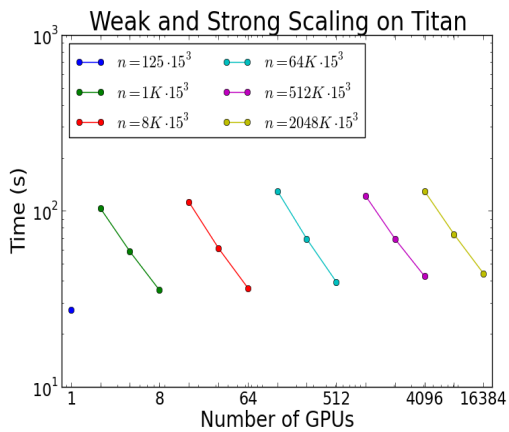

Figure 8. Timings on different number of GPUs for 1,000 timestep runs with n=$E(N+1)^3$, varying E with N=14.

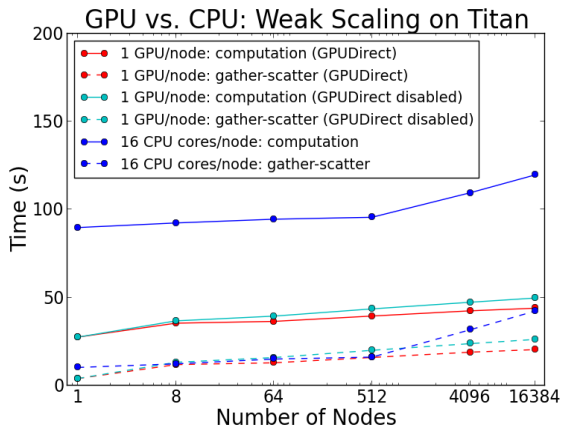

Figure 9. Timings on different numbers of GPUs and CPUs for 1,000 timestep runs with $n = 125 \cdot 15^3$ per node. 1 GPU and 16 CPU cores are used per node. Computation timings represent total simulation time including the gather-scatter.

Figure 8 shows strong- and weak-scale performance on up to 16,384 GPUs on Titan using GPUDirect-based MPI communication. Here, the connected lines correspond to strong scaling while the horizontal row of dots represents weak-scaling performance. We observe ~80% weak-scale efficiency from 8 GPUs to 16,384 GPUs.

Figure 9 compares timings of GPU runs for the case of $n/P = 125 \cdot 15^3$ from Figure 8 with those of CPU runs, up to 16,384 nodes (262,144 CPU cores) and with the equivalent GPU runs without GPUDirect communication. The GPU times for both GPUDirect and GPUDirect-disabled cases demonstrate 2.5× speedup over the all-CPU runs for the same number of nodes. We note that GPUDirect results in a 4%~12% performance gain over the GPU case with GPUDirect disabled.

We also note that the all-CPU case in Figure 9 exhibits a (repeatable) up-tick in communication overhead above 512 nodes. The cause of this is not clear, but one plausible explanation is the amount of message traffic induced by the large number (65,536 or 262,144) of MPI ranks in the all-CPU case. We examined the accumulated gather-scatter timings measured for the routine `fgs_fields_acc`. The

timing difference between the GPUDirect and GPUDirect-disabled runs represents the data transfer time between GPUs and CPUs within the gather-scatter routine. As for the CPU-only runs on Titan, the gather-scatter timings increase dramatically from 13% to 35% of the computation times as the number of MPI ranks increase, while those timings stay flat for the GPU-enabled version, having almost perfect weak scaling up to 16,384 GPUs, with 33%∼46% of computation times on GPU with GPUDirect and 36%∼53% of computation times on GPU with GPUDirect disabled.

Table 3 shows the profile produced by Cray's CrayPAT profiling tool for 8-GPU runs with $n/P = 125 \cdot 15^5$. The columns display, respectively, the percentage of total time, the wall clock time, the variation in times across the MPI tasks, the percentage of variation, and the number of occurrences of the program element. The OpenACC operations are divided into the time to COPY data to and from the device and the KERNEL time, which represents the amount of time the host is waiting for the device to complete the computation of the OpenACC kernel. If a program element does not contain ACC, it is the time the unit takes executing on the host. Ideally one would like to have the time dominated by the times waiting for the device execution and to have the COPY times minimized. The profile shown in Table 3 is dominated by kernel times. Although profiling increases the total computation time to 49.83 sec, compared with 35.40 sec without profiling, it demonstrates that majority of the time is spent in the curl evaluation, with 24% for the `local_curl_grad3_acc` and `curl_acc` routines. The second most significant portion is spent in the communication routine, with 18% for the `fgs_fields_acc` routine.

Conclusions

We have developed a fully functional and highly tuned MPI/OpenACC version of the computational electromagnetics code NekCEM. The implementation covers all solution routines for NekCEM's spectral-element/discontinuous Galerkin (SEDG) discretization of the time-dependent Maxwell equations, including tuned element-by-element operator evaluation and an optimized GPUDirect-based gather-scatter kernel to effect nearest-neighbor flux exchanges. Performance results on up to 16,384 GPUs of the Cray XK7 supercomputer Titan show more than 2.5× speedup over CPU-only performance on the same number of nodes (262,144 MPI ranks) for problem sizes of up to 6.9 billion grid points. This performance is realized for problems having more than 100,000 points per node. While the overall performance is respectable, some significant GPU performance issues nonetheless do limit scalability. In particular, the large $n_{1/2}$, and not the communication overhead, sets the strong-scale limit. We note that there is room for further improvement in the GPU-enabled version of NekCEM, including covering communication costs with computation (which is relatively easy) and the possibility of hybrid GPU/CPU computation (with more effort).

A major conclusion from this effort is that one can get reasonable performance for a full application using OpenACC plus GPUs and that this is a promising avenue for our much more complex (mutigrid-enabled, semi-implicit) Navier-Stokes solver, Nek5000. The OpenACC-based GPU algorithms and gather-scatter library discussed in this paper can readily be extended or used directly for these and other applications.

Acknowledgments

We thank Don Maxwell at OLCF for providing the power consumption data and Matt Colgrove at PGI for helping with timing tuning for PGI runs.

Funding

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 resources of the Argonne Leadership Computing Facility, which is supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.

References

Carpenter M and Kennedy C (1994) Fourth-order $2N$-storage Runge-Kutta schemes. *NASA Report TM 109112* .

Cray Inc (2012) *Cray Fortran Reference Manual*. Cray Inc.

Deville M, Fischer P and Mund E (2002) *High-order methods for incompressible fluid flow,* Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press.

Hesthaven J, Gottlieb S and Gottlieb D (2007) *Spectral methods for time-dependent problems,* Volume 21 of Cambridge monographs on applied and computational mathematics. Cambridge University Press.

Hesthaven J and Warburton T (2008) *Nodal discontinuous Galerkin methods, algorithms, analysis, and applications*. Springer.

Klöeckner A, Warburton T, Bridge J and Hestahven J (2009) Nodal discontinuous galerkin methods on graphics processors. *Journal of Computational Physics* 228: 7863–7882.

Markidis S, Gong J, Schliephake M, Laure E, Hart A, Henty D, Heisey K and Fischer PF (2015) Openacc acceleration of nek5000, spectral element code. *International Journal of High Performance Computing Applications* : 1–9.

Medina DS, St-Cyr A and Warburton T (2013) Occa: a unified approach to multi-threading. Under revision.

Min M and Fischer P (2011) An efficient high-order time integration method for spectral-element discontinuous Galerkin simulations in electromagnetics. MCS, ANL, Preprint ANL/MCS-P1802-1010.

Nvidia (2015) *Developing a Linux kernel module using RDMA for GPUDirect*. Nvidia Corporation, TB-06712-001 _v7.0.

openaccorg (2011) *The OpenACC Application Programming Interface*. Openacc Inc., Version 1.0.

Taflove A and Hagness S (2000) *Computational Electrodynamics, The Finite Difference Time Domain Method*. Artech House, Norwood, MA.

Table 3. CrayPAT profile on 8 GPUs; 1000 timestep runs with $(E, N) = (1000, 14)$.

```
Table 1:  Profile by Function Group and Function
  Time% |      Time |     Imb. |   Imb. |     Calls |Group
        |           |     Time |  Time% |           | Function
        |           |          |        |           |  PE=HIDE
 100.0% | 54.255936 |       -- |     -- | 1564223.4 |Total
|-----------------------------------------------------------------------------------------------------------
|  91.9% | 49.838669 |       -- |     -- | 1359683.0 |USER
||----------------------------------------------------------------------------------------------------------
||  15.3% |  8.283073 | 0.002307 |   0.0% |   10000.0 |local_curl_grad3_acc_.ACC_ASYNC_KERNEL@li.1476
||   8.8% |  4.790874 | 0.001347 |   0.0% |   10000.0 |curl_acc_.ACC_ASYNC_KERNEL@li.1568
||   7.3% |  3.979034 | 0.005755 |   0.2% |   30000.0 |rk4_upd_acc_.ACC_ASYNC_KERNEL@li.1470
||   3.7% |  2.000398 | 0.001428 |   0.1% |    5000.0 |cem_maxwell_add_flux_to_res_acc_.ACC_ASYNC_KERNEL@li.1169
||   3.2% |  1.711280 | 0.005003 |   0.3% |   30000.0 |fgs_fields_acc.ACC_ASYNC_KERNEL@li.490
||   3.1% |  1.678558 | 0.007621 |   0.5% |   30000.0 |fgs_fields_acc.ACC_ASYNC_KERNEL@li.430
||   3.0% |  1.644597 | 0.000371 |   0.0% |    5000.0 |cem_maxwell_invqmass_acc_.ACC_ASYNC_KERNEL@li.1284
||   3.0% |  1.623351 | 0.004822 |   0.3% |   30000.0 |fgs_fields_acc.ACC_ASYNC_KERNEL@li.526
||   2.9% |  1.588147 | 0.002331 |   0.2% |   30000.0 |fgs_fields_acc.ACC_ASYNC_KERNEL@li.550
||   2.1% |  1.115909 | 0.001834 |   0.2% |    5000.0 |cem_maxwell_restrict_to_face_acc_.ACC_ASYNC_KERNEL@li.896
||   2.0% |  1.105743 | 0.001329 |   0.1% |   15000.0 |chsign_acc_.ACC_ASYNC_KERNEL@li.2364
||   2.0% |  1.077413 | 0.003511 |   0.4% |    5000.0 |fgs_fields_acc.ACC_DATA_REGION@li.419
||   1.9% |  1.036148 | 0.000405 |   0.0% |    5000.0 |cem_maxwell_flux3d_acc_.ACC_ASYNC_KERNEL@li.1108
||   1.8% |  0.997496 | 0.001525 |   0.2% |    5000.0 |pw_exec_sends_acc.ACC_COPY@li.178
```

Author biographies

Matthew Otten is a Ph.D. candidate in the Department of Physics at Cornell University, with a minor in computer science. Otten worked as a Givens Associate in 2014 at the Mathematics and Computer Science Division at Argonne National Laboratory and has been a co-op student at Argonne since then. His research focuses on developing scalable algorithms for solving complex quantum systems.

Azamat Mametjanov is a postdoctoral associate in the Mathematics and Computer Science Division at Argonne. He received his Ph.D. from the University of Nebraska at Omaha in 2011. His research focuses on performance characterization and optimization of codes for modern architectures using automated program transformation techniques.

Jing Gong completed his Ph.D. in scientific computing on Hybrid Methods for Unsteady Fluid Flow Problems in Complex Geometries at Uppsala University in 2007. He also holds an M.Sc. in scientific computing from the Royal Institute of Technology (KTH), Stockholm, and an M.Eng. in mechatronics from Beihang University, Beijing, China. He joined the PDC Center for High Performance Computing at KTH as a researcher in computational fluid dynamics in January 2012.

Aaron Vose is an HPC software engineer at Cray's Supercomputing Center of Excellence at Oak Ridge National Laboratory. Vose helps domain scientists at ORNL port and optimize scientific software to achieve maximum scalability and performance on world-class HPC resources, such as the Titan supercomputer. Prior to joining Cray, Vose spent time at the National Institute for Computational Sciences as well as the Joint Institute for Computational Sciences, where he worked on scaling and porting bioinformatics software to the Kraken supercomputer. Vose holds a master's degree in computer science from the University of Tennessee at Knoxville.

John Levesque is the director of Cray's Supercomputing Center of Excellence for the Trinity system based in Los Alamos National Laboratory. He is responsible for the group performing application porting and optimization for break-through science projects. Levesque has been in high-performance computing for 45 years. Recently he was promoted to Cray's Chief Technology Office, heading the company's efforts in application performance.

Paul Fischer is a professor in the Department of Computer Science and Mechanical Science and Engineering at UIUC. Prior to that, he had been at Argonne from 1998 to 2014 and a faculty member at Brown University from 1991 to 1998. He received his Ph.D. in mechanical engineering from MIT in 1989 and was a postdoctoral associate in applied mathematics at Caltech in 1990-1991. Fischer received the first Center for Research on Parallel Computation Prize Fellowship from Caltech in 1990 and the Gordon Bell Prize for High-Performance Computing in 1999. He was elected as a Fellow of the American Association for the Advancement of Science (AAAS) in 2012 for technical contributions to computational fluid dynamics on extreme-scale computers.

Misun Min is a computational scientist at Argonne National Laboratory. She received her Ph.D. from Brown University in 2003. After a postdoctoral research position at Argonne, she joined the Mathematics and Computer Science Division as an assistant computational scientist and was promoted to computational scientist in 2011. Her research focuses on developing numerical algorithms based on high-order spatial and time discretizations for solving electromagnetics and quantum systems. She is the designer and developer of the computational electromagnetics solver package NekCEM.