

NekRS, a GPU-Accelerated Spectral Element Navier–Stokes Solver

Paul Fischer^{b,c,a}, Stefan Kerkemeier^a, Misun Min^{a,*}, Yu-Hsiang Lan^a, Malachi Phillips^b, Thilina Rathnayake^b, Elia Merzari^{d,a}, Ananias Tomboulides^{e,a}, Ali Karakus^f, Noel Chalmers^g, Tim Warburton^h

^aMathematics and Computer Science, Argonne National Laboratory, Lemont, IL 60439

^bDepartment of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801

^cDepartment of Mechanical Science and Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801

^dDepartment of Nuclear Engineering, Penn State, PA 16802

^eDepartment of Mechanical Engineering, Aristotle University of Thessaloniki, Greece 54124

^fMechanical Engineering Department, Middle East Technical University, 06800, Ankara, Turkey

^gAMD Research, Advanced Micro Devices Inc., Austin, TX 78735

^hDepartment of Mathematics, Virginia Tech, Blacksburg, VA 24061

Abstract

The development of NekRS, a GPU-oriented thermal-fluids simulation code based on the spectral element method (SEM) is described. For performance portability, the code is based on the open concurrent compute abstraction and leverages scalable developments in the SEM code Nek5000 and in libParanumal, which is a library of high-performance kernels for high-order discretizations and PDE-based miniapps. Critical performance sections of the Navier–Stokes time advancement are addressed. Performance results on several platforms are presented, including scaling to 27,648 V100s on OLCF Summit, for calculations of up to 60B gridpoints.

Keywords: NekRS, Nek5000, libParanumal, OCCA, GPU, Scalability, Performance, Spectral Element Method, Incompressible Navier–Stokes, Exascale Applications

1. Introduction

A fundamental challenge in fluid mechanics and heat transfer is to accurately simulate physical interactions over a large range of spatial and temporal scales. Such simulations can involve billions of degrees of freedom evolved over hundreds of thousands of timesteps. Simulation campaigns for these problems can require weeks or months of wall-clock time on the world’s fastest supercomputers. One of the principal objectives of high-performance computing (HPC) is to reduce these runtimes to manageable levels.

We are interested in modeling turbulent flows using either direct numerical simulation (DNS) to capture all scales of motions, large eddy simulation (LES) to capture the modes that dominate momentum and thermal transport, or Reynolds-averaged Navier–Stokes (RANS) formulations that emulate both small- and large-scale transport with closure models. Applications include reactor thermal hydraulics, internal combustion engines, ocean and atmospheric flows, vascular flows, astrophysical problems, and basic turbulence questions for theory and model development. Simulations in these

areas present significant challenges with respect to scale resolution, multiphysics, and complex computational domains. In many cases, experimental data are expensive or impossible to obtain, making simulation on leadership computing platforms critical to informed analysis.

With current exascale computing programs in the U.S. and elsewhere developing GPU-based HPC platforms it is imperative to exploit the performance potential of these powerful node architectures. In this paper, we describe the development of a new GPU-oriented open-source code for thermal-fluid analysis, NekRS, which has emerged out of two HPC software projects. *Nek5000* [1] was one of the first production-level single-program multiple-data (SPMD) codes deployed on distributed-memory parallel computers [2]. It has demonstrated scalability to leading-edge platforms through the SPMD era [3, 4] and readily scales to millions of MPI ranks [5]. Early GPU efforts for Nek5000 commenced with OpenACC ports [6] and [7] (for NekCEM). *libParanumal* [8, 9] is a self-contained high-order finite element library that uses highly optimized kernels based on the portable Open Concurrent Compute Abstraction (OCCA) [10, 11]. It includes sublibraries for dense linear algebra, Krylov solvers, parallel mesh han-

*Corresponding author

Email address: mmin@mcs.anl.gov (Misun Min)

dling and polynomial approximation, p -type and algebraic multigrid, time stepping, gather-scatter operations and halo exchanges, and core miscellaneous operations. The libParanumal sublibraries support meshes consisting of triangles, quadrilaterals, tetrahedra, or hexes. The libParanumal project also includes mini-apps providing GPU accelerated solvers for a wide variety of transport-dominated physics applications. A significant feature of the libParanumal kernels is that, in the majority of cases, they are tuned to meet the roofline performance limits. For example, FP64 performance in excess of 1 TFLOPS is realized for local SEM matrix-vector product (matvec) kernels on the NVIDIA V100 [8, 12]. Each solver supports multi-GPU simulation via Nek5000's *gslib* for efficient MPI-based gather-scatter operations and halo exchanges [13].

In the present work, we describe a new code, NekRS, which is written in C++/OCCA. The performant kernels in NekRS started as an early fork from libParanumal and were tailored and expanded to meet the specific requirements of large-scale turbulent flow applications in complex domains. NekRS provides access to the standard Nek5000 interface and features (e.g., conjugate heat transfer), which allows users to leverage existing application-specific source code and data files on GPU-based platforms.

The remainder of this article is organized as follows. In Section 2 we provide relevant details of the governing equations and spectral element discretization. In Section 3 we describe the parallel GPU development, including parallel communication and partitioning strategies at exascale and present an illustration of high-performance kernels for GPU-based nodes. In Section 4 we provide extensive performance studies at scale, including weak- and strong-scale studies on all of Summit. We conclude with remarks and discussion in Section 5.

2. Formulations

We simulate thermal transport governed by the incompressible Navier–Stokes (NS) and energy equations,

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \frac{1}{Re} \nabla^2 \mathbf{u}, \quad (1)$$

$$\nabla \cdot \mathbf{u} = 0, \quad (2)$$

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \frac{1}{Pe} \nabla^2 T, \quad (3)$$

subject to appropriate velocity (\mathbf{u}), pressure (p), and temperature (T) initial conditions in Ω and boundary conditions on $\partial\Omega$. For typical applications, the Reynolds (Re) and Peclet (Pe) numbers

are large, implying that the flows are advection dominated. For high Re , in fact, the flows are fully turbulent, implying a need for highly accurate numerical discretizations in order to avoid numerical dispersion and dissipation [14]. (We note that NekRS currently supports conjugate heat transfer where T may be defined on a domain that is larger than Ω . In what follows, however, we omit further discussion of the energy equation (3).)

2.1. BDF Time Discretization

We begin with a backward difference (BDF k) approximation to $\frac{\partial \mathbf{u}}{\partial t}$ to derive an implicit Stokes substep for velocity and pressure at time level t^n ,

$$\frac{\beta_0}{\Delta t} \mathbf{u}^n = \frac{1}{\Delta t} \mathbf{u}^* - \nabla p^n + \frac{1}{Re} \nabla^2 \mathbf{u}^n, \quad (4)$$

$$\nabla \cdot \mathbf{u}^n = 0, \quad (5)$$

where \mathbf{u}^* accounts for quantities known from prior substeps and is computed in one of two ways. The standard (Courant- or CFL-limited) formulation is BDF k /EXT k ,

$$\mathbf{u}^* := - \sum_{j=1}^k (\beta_j \mathbf{u}^{n-j} + \Delta t \alpha_j \mathbf{u}^{n-j} \cdot \nabla \mathbf{u}^{n-j}), \quad (6)$$

where the β_j s are the k th-order BDF coefficients and the α_j s are k th-order extrapolation coefficients. An alternative formulation that avoids the CFL constraint on stepsize Δt is the semi-Lagrangian approach with

$$\mathbf{u}^* := - \sum_{j=1}^k \beta_j \tilde{\mathbf{u}}^{n-j}. \quad (7)$$

Here, each $\tilde{\mathbf{u}}^{n-j}(\mathbf{x})$ represents the value of $\mathbf{u}^{n-j}(\mathbf{x}^*)$, where \mathbf{x}^* is the foot of the characteristic that would be found by integrating the velocity field backward in time over $[t^n, t^{n-1}]$. In practice, the off-grid interpolation required for direct evaluation of $\mathbf{u}^{n-j}(\mathbf{x}^*)$ can be avoided by solving a hyperbolic advective subproblem on $[t^{n-j}, t^n]$,

$$\frac{\partial \mathbf{w}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{w} = 0, \quad (8)$$

with initial condition $\mathbf{w}(\mathbf{x}, t^{n-j}) = \mathbf{u}^{n-j}$ [15, 16]. We typically use the third-order ($k=3$) formulation for (6) with a Courant number of CFL=0.5. For (7), CFL=2–4 is most common, but we typically use only second-order in time ($k=2$) because of the relative expense of the hyperbolic substeps (8), which are fully dealiased [17].

2.2. Implicit Stokes Solve

The unsteady linear Stokes problem (4)–(5) is further decoupled via a fractional step method that treats the divergence-free and viscous terms as separate subproblems. A pressure-Poisson problem derives from taking the divergence of (4),

$$-\nabla \cdot (\nabla p^n) = -\frac{\nabla \cdot \mathbf{u}^*}{\Delta t} + \frac{1}{Re} \nabla \cdot (\nabla \times \omega), \quad (9)$$

where $\omega = \sum_{j=1}^k \alpha_j \nabla \times \mathbf{u}^{n-j}$ is the extrapolated vorticity, which serves to control divergence errors at the boundaries. Additional details on the boundary conditions for (9) can be found in [18, 19, 20, 21].

The final substep requires the solution of

$$-\frac{1}{Re} \nabla^2 \mathbf{u}^n + \frac{\beta_0}{\Delta t} \mathbf{u}^n = \frac{\mathbf{u}^{**}}{\Delta t}, \quad (10)$$

where \mathbf{u}^{**} is the divergence-free velocity

$$\mathbf{u}^{**} = \mathbf{u}^* - \Delta t \nabla p^n. \quad (11)$$

The advantage of (4)–(11) is that it decouples the NS equations into independent substeps, each of which can be efficiently treated by techniques tailored to the governing physics: hyperbolic substeps for advection, diagonally preconditioned conjugate gradient (PCG) iteration for the viscous Helmholtz problems, and multilevel PCG or GMRES for the pressure solve. Because it governs the fastest modes (i.e., the acoustic modes, which are infinitely fast in the incompressible model), the pressure-Poisson problem is intrinsically the stiffest substep. Isolating it from the other governing operators results in a fast algorithm because there is no need to evaluate viscous or advection operators with each iteration, which would be required of a fully implicit approach.

2.3. Spectral Element Discretization

To develop an efficient spatial discretization, we employ high-order spectral elements (SEs) [22] in which the solution, data, and test functions are represented as *locally structured* N th-order tensor product polynomials on a set of E *globally unstructured* curvilinear hexahedral brick elements. The approach yields two principal benefits. First, for smooth functions such as solutions to the incompressible NS equations, high-order polynomial expansions yield exponential convergence with approximation order, implying a significant reduction in the number of unknowns ($n \approx EN^3$) required to reach engineering tolerances. Second, the locally structured forms permit local lexicographical ordering with minimal indirect addressing and, crucially, the use of tensor-product sum factorization to yield low $O(n)$ storage costs and $O(nN)$

work complexities [23]. As we demonstrate, the leading order $O(nN)$ work terms can be cast as small dense matrix-matrix products (tensor contractions) with favorable $O(N)$ work-to-storage ratios (computational intensity) [24].

The equations for the SE basis coefficients are derived from a weighted residual formulation for each subproblem. For example, (6) and (10)–(11) become *Find* $\mathbf{u}^n, p^n \in X_b^N(\Omega) \times X^N(\Omega)$ *such that for all* $\mathbf{v}, q \in X_0^N \times X^N$,

$$(\mathbf{v}, \mathbf{u}^*) = -\sum_{j=1}^k (\beta_j (\mathbf{v}, \mathbf{u}^{n-j}) + \Delta t \alpha_j (\mathbf{v}, \mathbf{u}^{n-j} \cdot \nabla \mathbf{u}^{n-j})), \quad (12)$$

$$(\nabla q, \nabla p^n) = -\frac{1}{\Delta t} (q, \nabla \cdot \mathbf{u}^*) - \frac{1}{Re} (\nabla q, \nabla \times \omega) \quad (13)$$

$$\frac{1}{Re} (\nabla \mathbf{v}, \nabla \mathbf{u}^n) + \frac{\beta_0}{\Delta t} (\mathbf{v}, \mathbf{u}^n) = \frac{1}{\Delta t} (\mathbf{v}, \mathbf{u}^{**}). \quad (14)$$

Here, $(\mathbf{v}, \mathbf{u}) = \int_{\Omega} \mathbf{v} \cdot \mathbf{u} dV$ is the L^2 inner product on Ω ; X_b^N is the subset of X^N satisfying the Dirichlet conditions on $\partial\Omega$; X_0^N is the subset of X^N satisfying homogeneous Dirichlet conditions on $\partial\Omega$; and $X^N \subset H^1$ is the set of continuous N th-order spectral element basis functions described in [24]. H^1 is the usual Sobolev space of functions that are square integrable on Ω , whose derivatives are also square integrable.

The discrete systems of equations are derived by formally expanding the test and trial functions in terms of a basis $\{\phi_i\}$ for X^N . Consider the weak form of the Poisson equation, *Find* $u \in X_0^N$ *such that* $(\nabla v, \nabla u) = (v, f)$ *for all* $v \in X_0^N$, with

$$u(\mathbf{x}) = \sum_{j=1}^n u_j \phi_j(\mathbf{x}). \quad (15)$$

Inserting this expansion and taking $v = \phi_i$, we get

$$A_{ij} \underline{u} = \underline{b}_i, \quad (16)$$

with $\underline{u} = [u_1 \dots u_n]^T$ the vector of unknown basis coefficients and

$$A_{ij} := (\nabla \phi_i, \nabla \phi_j), \quad (17)$$

the symmetric positive definite (SPD) stiffness matrix associated with the Poisson operator. Elements of the data vector \underline{b} are computed by evaluating inner products $b_i := (\phi_i, f)$.

To derive fast matrix-free operator evaluations for iterative solution of (16), we introduce the local SE basis functions. To begin, we assume $\Omega = \cup_{e=1}^E \Omega^e$, where the non-overlapping subdomains (elements) Ω^e are images of the reference domain,

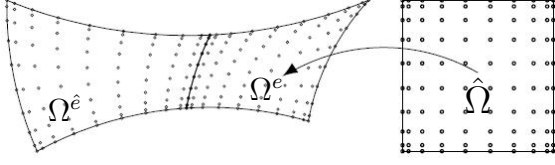


Figure 1: 9th-order SE mapping from canonical domain $\hat{\Omega}$ to physical subdomain $\Omega^e \subset \mathbb{R}^2$.

$\mathbf{r} \in \hat{\Omega} = [-1, 1]^3$, given by

$$\begin{aligned} \mathbf{x}|_{\Omega^e} &= \mathbf{x}^e(r, s, t) \\ &= \sum_{k=0}^N \sum_{j=0}^N \sum_{i=0}^N \mathbf{x}_{ijk}^e h_i(r) h_j(s) h_k(t), \end{aligned} \quad (18)$$

as illustrated in Fig. 1. Here, $h_i(r)$ (s , or t) $\in \mathbb{P}_N$ is assumed to be a cardinal Lagrange polynomial, $h_i(\xi_j) = \delta_{ij}$, based on the Gauss–Lobatto–Legendre (GLL) quadrature points, $\xi_j \in [-1, 1]$, $j = 0, \dots, N$. This choice of points yields well-conditioned operators and allows for accurate pointwise quadrature with a diagonal mass matrix.

All functions in X^N have a form similar to (18). For example, the scalar $u(\mathbf{x})|_{\Omega^e} = u(\mathbf{x}^e(\mathbf{r})) =: u^e(\mathbf{r})$ is written in terms of the $(N+1)^3$ local basis coefficients $\underline{u}^e := \{u_{ijk}^e\}$,

$$u|_{\Omega^e} = \sum_{k=0}^N \sum_{j=0}^N \sum_{i=0}^N u_{ijk}^e h_i(r) h_j(s) h_k(t). \quad (19)$$

An important consequence of the GLL-based tensor-product Lagrange polynomial representation is that differentiation with respect to r , s , and t at quadrature points $\xi_{ijk} = (\xi_i, \xi_j, \xi_k)$ can be expressed as efficient tensor contractions. Let

$$\hat{D}_{ii} := \left. \frac{dh_i}{dr} \right|_{\xi_i} \quad (20)$$

be the one-dimensional differentiation matrix mapping from the nodal points to the (identical) quadrature points, and let \hat{I} be the $(N+1) \times (N+1)$ identity matrix. Let \underline{u}_r^e , \underline{u}_s^e , \underline{u}_t^e denote partial derivatives of $u^e(\mathbf{r})$ with respect to the coordinates $\mathbf{r} = (r, s, t) = (r_1, r_2, r_3)$ evaluated at the GLL points. Then

$$\underline{u}_r^e = D_1 \underline{u}^e := (\hat{I} \otimes \hat{I} \otimes \hat{D}) \underline{u}^e = \sum_i \hat{D}_{ii} u_{ijk}^e, \quad (21)$$

$$\underline{u}_s^e = D_2 \underline{u}^e := (\hat{I} \otimes \hat{D} \otimes \hat{I}) \underline{u}^e = \sum_j \hat{D}_{jj} u_{ijk}^e, \quad (22)$$

$$\underline{u}_t^e = D_3 \underline{u}^e := (\hat{D} \otimes \hat{I} \otimes \hat{I}) \underline{u}^e = \sum_k \hat{D}_{kk} u_{ijk}^e. \quad (23)$$

The chain rule is used to differentiate with respect to $\mathbf{x} = (x, y, z) = (x_1, x_2, x_3)$:

$$\nabla^e \underline{u}^e = \mathbf{D}^e \underline{u}^e = \left. \frac{\partial u^e}{\partial x_p^e} \right|_{\xi_{ijk}} = \sum_{q=1}^3 \left(\frac{\partial r_q}{\partial x_p^e} \frac{\partial u^e}{\partial r_q} \right)_{\xi_{ijk}}, \quad (24)$$

where derivatives with respect to r_q are computed by (21)–(23). The array of metrics $\frac{\partial r_q}{\partial x_p^e}$ is found by inverting (at each grid point, ξ_{ijk}) the 3×3 matrix, $\frac{\partial x_p^e}{\partial r_q} = D_q x_p^e$.

Note that evaluation of the full gradient (24) involves three tensor contractions, (21)–(23), each requiring $2(N+1)^4$ operations and $(N+1)^3$ memory references, followed by three pointwise contractions, $u_{x_p}^e = \sum_{q=1}^3 r_{q,x_p^e} u_{r_q}^e$, requiring $15(N+1)^3$ operations and $9(N+1)^3$ memory references (assuming that $u_{r_q}^e$ is cached from the ∇_r operation). For the full field, the gradient $(\underline{u}_x, \underline{u}_y, \underline{u}_z) \leftarrow \underline{u}$ thus requires $10E(N+1)^3 \approx 10n$ memory references and $6E(N+1)^4 + 15E(N+1)^3 \approx n(15+6N)$ operations. We note that the $O(N^4)$ work terms (21)–(23) are readily cast as dense matrix-matrix products [3, 24].

In addition to differentiation, the weighted residual formulation requires integration, which is effected through GLL quadrature. For any $u, v \in X^N$, we define the discrete inner products,

$$(v, u)_N := \sum_{e=1}^E (v, u)^e, \quad (25)$$

$$(v, u)^e := \sum_{k=0}^N \sum_{j=0}^N \sum_{i=0}^N v_{ijk}^e \rho_{ijk} \mathcal{J}_{ijk}^e u_{ijk}^e \quad (26)$$

$$= (\underline{v}^e)^T B^e \underline{u}^e. \quad (27)$$

Here, $\rho_{ijk} = \rho_i \rho_j \rho_k$ is the product of one-dimensional GLL quadrature weights; $\mathcal{J}^e(\mathbf{r}) = \left| \frac{\partial x_p^e}{\partial r_q} \right|$ is the Jacobian associated with the map \mathbf{x}^e from $\hat{\Omega}$ to Ω^e ; and $B^e = \text{diag}(\rho_{ijk} \mathcal{J}_{ijk}^e)$ is a local, diagonal mass matrix. For affine maps, $(v, u)^e \equiv \int_{\Omega^e} v u dV$ whenever the product vu is a polynomial of degree $2N-1$ or less. The high accuracy realized by the GLL quadrature is sufficient to ensure stability for the Poisson operator when $(\nabla \phi_i, \nabla \phi_j)_N$ replaces $(\nabla \phi_i, \nabla \phi_j)$ in (17) [25], but not for the advection operator, where the integrand is of degree $3N$ and thus requires higher-order integration [17].

Equipped with the basic calculus tools (21)–(27), we evaluate the bilinear form $(\nabla v, \nabla u)$ as follows,

$$\begin{aligned} (\nabla v, \nabla u) &= \sum_{e=1}^E (\nabla v^e, \nabla u^e)^e \\ &= \sum_{e=1}^E (\underline{v}^e)^T A^e \underline{u}^e = \underline{v}_L^T A_L \underline{u}_L, \end{aligned} \quad (28)$$

where $\underline{u}_L = [\underline{u}^1 \dots \underline{u}^E]$ is the collection of all local basis vectors and $A_L = \text{block-diag}(A^e)$ comprises the local stiffness matrices given in factored form

by

$$A^e = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}^T \begin{pmatrix} G_{11}^e & G_{12}^e & G_{13}^e \\ G_{12}^e & G_{22}^e & G_{23}^e \\ G_{13}^e & G_{23}^e & G_{33}^e \end{pmatrix} \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}. \quad (29)$$

Here, the six local geometric factors $G_{ij}^e = G_{ji}^e$ are diagonal matrices with one nontrivial entry for each gridpoint ξ_{ijk} ,

$$[G_{mm'}^e]_{ijk} = \left[\sum_{l=1}^3 \frac{\partial r_m}{\partial x_l^e} \frac{\partial r_{m'}}{\partial x_l^e} \right]_{ijk} B_{ijk}^e. \quad (30)$$

Evaluation of $A^e \underline{u}^e$ thus requires $7(N+1)^3$ memory references (six for $G_{mm'}^e$ and one for \underline{u}^e) and $12(N+1)^4 + 15(N+1)^3$ operations, per element. Aside from preconditioning, $A^e \underline{u}^e$, constitutes the principal work for the pressure-Poisson problem and for the viscous solves. In the case of the Jacobi-preconditioned viscous substeps, it is the only work term that scales as $O(nN)$. The other contributions in Jacobi PCG total to $\sim 12n$ floating-point operations.

To ensure interelement continuity ($u, v \in X^N \subset H^1$), one must constrain local basis coefficients at shared element interfaces to be equal. That is, for any given sets of coefficient indices (i, j, k, e) and $(\hat{i}, \hat{j}, \hat{k}, \hat{e})$,

$$\mathbf{x}_{ijk}^e = \mathbf{x}_{\hat{i}\hat{j}\hat{k}}^{\hat{e}} \longrightarrow u_{ijk}^e = u_{\hat{i}\hat{j}\hat{k}}^{\hat{e}}. \quad (31)$$

The statement (31) leads to the standard finite element procedures of matrix assembly and assembly of the load and residual vectors. Matrix-free algorithms that use iterative solvers require assembly only of vectors, since the matrices are never formed.

To implement (31), we introduce a *global-to-local map*, formally expressed as a sparse matrix-vector product, $\underline{u}_L = Q \underline{u}$, which takes global (uniquely defined) degrees of freedom u_l from the index set $l \in \{1, \dots, n\}$ to their (potentially multiply defined) local counterparts u_{ijk}^e . The continuity requirement leads to

$$\underline{v}_L^T A_L \underline{u}_L = \underline{v}^T Q^T A_L Q \underline{u} = \underline{v}^T A \underline{u}, \quad (32)$$

from which we conclude that the global stiffness matrix is $A = Q^T A_L Q$. We refer to A_L as the unassembled stiffness matrix and A as the assembled stiffness matrix.¹ With this factored form, a matrix-vector product can be evaluated

¹We typically denote $Q^T A_L Q =: \bar{A}$ as the *Neumann operator*, which is orthogonal to the constant vector, and $A = R \bar{A} R^T$ as the SPD stiffness matrix, where R is a restriction matrix that discards rows corresponding to Dirichlet data [24]. Application of R does not impact complexity so we do not discuss it further.

as $\underline{w} = Q^T A_L Q \underline{p}$, which allows parallel evaluation of the work-intensive step of applying A^e to basis coefficients in each element Ω^e . Application of the Boolean matrices Q and Q^T represents the communication-intensive phases of the process.² We note that Q and Q^T are the spectral element/finite element method equivalents of the finite difference “halo” exchange. Unlike finite differences, however, Q and Q^T have a *unit-depth stencil for all N , and the discretization is thus communication minimal*.

3. Parallel GPU Development

Our parallel approach to solving the incompressible NS equations follows the standard SPMD paradigm of partitioning the domain across P MPI ranks, each with its own private address space, and time advancing the equations in a cooperative fashion using iterative solvers to solve the elliptic subproblems for the velocity, temperature, and pressure. On each node, we run one MPI rank per GPU. All data resides on the device, with a copy back to the host only when needed (e.g., for I/O or analysis of turbulence statistics).

NekRS has been developed in close collaboration with the libParanumal project, [8, 9, 26, 27] which provides high-performance kernels for high-order methods on GPUs. The GPU kernels are written in the portable Open Concurrent Compute Abstraction (OCCA) library [10, 11, 28] to abstract between different parallel languages such as OpenCL, CUDA, and HIP. OCCA allows developers to implement the parallel kernel code in a slightly decorated C++ language, OKL. At runtime, the user can specify which parallel programming model to target, after which OCCA translates the OKL source code into the desired target language and Just-In-Time (JIT) compiles kernels for the user’s target hardware architecture. In the OKL language, parallel loops and variables in special memory spaces are described with simple attributes. For example, iterations of nested parallel for loops in the kernel are annotated with `@outer` and `@inner` to describe how they are to be mapped to a grid of work-item and work-groups in OpenCL or threads and thread-blocks in CUDA and HIP. All iterations that are annotated with `@outer` or `@inner` are assumed to be free of loop carried dependencies. We describe several of the kernels in detail below.

We note that experience with Mira [5] has established that the strong-scale limit for Nek5000

²In the case of nonconforming elements, Q is not Boolean but can be factored into a Boolean matrix times a local interpolation matrix [24].

is around $n/P = 2000\text{--}4000$ points per MPI rank (in -c32 mode), meaning using just 2 to 8 elements per rank for typical orders of $N=7$ to 11. By contrast, the strong-scale limit on modern GPUs such as the Nvidia V100 is around $n/P = 2\text{--}4$ million points per rank (i.e., per GPU), or about 4,000 to 8,000 elements [12]. Moreover, “hero” runs on Mira were at the level of about 15 million elements, which could easily be run on a million ranks. On Summit, NekRS routinely is run with 175 million elements and $N = 7$ ($n = 60B$). The quantitative differences have an impact on considerations such as communication hiding and the importance of internode latency. We discuss these issues further in Section 4.

3.1. Domain Partitioning

For the large runs that are now routine on Summit, we partition the domain using parallel recursive spectral bisection (parRSB) [29] in such a way that the number of elements on each processor differs by at most 1. The Fiedler vector for parRSB is computed by using either restarted Lanczos or inverse iteration with lean algebraic multigrid (AMG) [30] on the element-centered connectivity graph. Prior to running parRSB, we execute recursive coordinate bisection (parRCB) in order to organize the graph into reasonably connected subsets on each processor. Otherwise, the parRSB iterations can incur significant communication overhead because one or more processors may have each of its elements connected to different processors if the ordering is arbitrary (e.g., partitioned according to the original element numbering). Prepartitioning with parRCB can cut parRSB run times by a factor of 100. On GPU-based systems parRCB/RSB are run on the CPUs because the number of elements, E , in the SEM is typically three orders of magnitude smaller than the number of grid points, $n = EN^3$. On 8,100 cores of Summit, the partition time for $E = 60$ million is about 40 seconds using Lanczos with a parRCB preprocessing time of 0.8 seconds.

3.2. Parallel Communication

Time advancement of the discretized NS equations effectively amounts to executing a sequence of matrix-vector products. Because both the trial and test spaces (\mathbf{u}, p and \mathbf{v}, q) are continuous, these products involve a map of the form $\underline{w} = Q^T Z_L Q \underline{u}$, where $Z_L = \text{block-diag}(Z^e)$ represents some localized physics (e.g., advection or diffusion), Q^T reflects the continuity of the test functions, and Q the continuity of the trial functions. When recast as $\underline{w}_L = Q Q^T Z_L \underline{u}_L$, the parallelization is clear: each processor evaluates $\underline{\tilde{w}}^e = Z^e \underline{u}^e$,

$e = 1, \dots, E_p$, where E_p is the number of elements on rank p (With the private memory model, other discriminators are not required for Z^e or \underline{u}^e because their data is implicitly indexed by p .) This parallel work step is followed by the communication phase, $\underline{w}_L = Q Q^T \underline{\tilde{w}}_L$, which corresponds to an exchange and sum of shared interface values between adjacent elements. On Mira, $Q Q^T$ is almost 100% communication (latency) dominated in the strong-scale limit of one or two elements per rank. On GPUs there are thousands of elements per rank. A significant portion of the elements are thus interior to the local partition, and there is consequently an opportunity to overlap work and communication.

$Q Q^T$ is implemented in parallel by using the open-source communication library *gslib*, which supports multiple data types and associative/commutative operations (e.g., min, max, *, +) for scalar and vector fields, as well as one-sided operations Q and Q^T . The adjacency graph is prescribed by a simple interface: The user provides indices in a vector, \underline{g}_p of length $n_p = E_p(N+1)^3$ on each processor, $p = 0, \dots, P-1$. The indices correspond to global pointers, while their positions, $1, \dots, n_p$ correspond to local pointers. Passing \underline{g}_p to *gs_setup* returns a handle, *gsh*, which is then used when executing *gs_op(gsh, \underline{w}_L , +)* to produce $\underline{w}_L \leftarrow Q Q^T \underline{w}_L$. The user does not need to know which processor holds the adjacent elements or anything else about the shared indices. If a global pointer is unique in the set $\bigcup_p \{\underline{g}_p\}$, then the corresponding entry in \underline{w}_L will be unchanged. If one knows a priori that certain entries in \underline{g}_p are singletons, a 0 index may be supplied in \underline{g}_p , which saves work in the discovery phase of *gs_setup*. (We typically set all element-interior pointers 0.)

At setup time, *gslib* picks a communication strategy (pairwise, crystal router³ or all-reduce) that yields the lowest maximum time over a set of trials for the given adjacency graph. We have found this optimization to be particularly important ($10\times$) in the context of AMG for solution of distributed coarse-grid problems [4]. When the number of nonzeros per row is large, the pairwise exchange can require a large number of messages, whereas crystal router requires only $\log_2 P$ messages. The all-reduce approach has a nominal $\log_2 P$ cost; but with hardware support, the cost is effectively independent of P and (on Mira) bounded by $4\times$ the latency for short messages, even with $P > 1M$. *gs_setup* executes in $O(\log P)$ time and is quite fast. For example, with 3B gridpoints on a million ranks of Mira, the time (including setup and 10 trials

³Crystal router is a scalable generalized all-to-all [31].

Algorithm 1: Advection operator, 2D thread configuration

Output: Vector Qq , size $3 \times N_d \times N_p$, indexed as $Qq[\text{element}][x/y/z][ix][iy][iz]$
Inputs: Vector q , size $3 \times N_d \times N_p$, indexed as $q[\text{element}][x/y/z][ix][iy][iz]$,
differentiation matrix D^{1D} , size $N_q \times N_q$, volumetric geometric factors G_v , size $N_d \times N_p \times 12$

```

1 for  $e \in \{0, 1, \dots, N_d - 1\}$  do
2   Allocate  $sU, sV, sW, sD$  as  $(p+1) \cdot (p+1)$  shared memory arrays;
3   Allocate  $rU, rV, rW$  as  $(p+1)$  register arrays;
4   for  $k \in \{0, 1, \dots, p\}$  do
5     for each thread  $ij$  do
6       if  $k == 0$ , load  $D^{1D}$  into  $sD$ ;
7        $sU[j][i] = q[e][0][k][j][i]$ ,  $sV[j][i] = q[e][1][k][j][i]$ ;
8        $sW[j][i] = q[e][2][k][j][i]$ ;
9       if  $k == 0$  then
10         $rU[:] = q[e][0][:][j][i]$ ,  $rV[:] = q[e][1][:][j][i]$ ;
11         $rW[:] = q[e][2][:][j][i]$ ;
12      end
13    end
14    sync.threads();
15    for each thread  $ij$  do
16       $dudr = \sum_{n=0}^p sD[i][n] \cdot sU[j][n]$ ,  $dude = \sum_{n=0}^p sD[j][n] \cdot sU[n][i]$ ;
17       $dudt = \sum_{n=0}^p sD[k][n] \cdot rU[n]$ ,  $dvdz = \sum_{n=0}^p sD[i][n] \cdot sV[j][n]$ ;
18       $duds = \sum_{n=0}^p sD[j][n] \cdot sV[n]$ ,  $dvdz = \sum_{n=0}^p sD[k][n] \cdot rV[n]$ ;
19       $dudr = \sum_{n=0}^p sD[i][n] \cdot sW[j][n]$ ,  $duds = \sum_{n=0}^p sD[j][n] \cdot sW[n][i]$ ;
20       $dudt = \sum_{n=0}^p sD[k][n] \cdot rW[n]$ ;
21      Load  $drdx, drdy, drdz, dsdx, dsdy, dsdz, dtdx, dtdy, dtdz$  from  $G_v$ ;
22      // Apply chain rule;
23       $dudx = drdx \cdot dudr + dsdx \cdot dude + dtdx \cdot dudt$ ;
24       $dudy = drdy \cdot dudr + dsdy \cdot duds + dtdy \cdot dudt$ ;
25       $dudz = drdz \cdot dudr + dsdz \cdot duds + dtdz \cdot dudt$ ;
26       $dvdz = drdz \cdot dvdz + dsdz \cdot duds + dtdz \cdot dvdz$ ;
27       $dvdz = drdy \cdot dvdz + dsdy \cdot duds + dtdy \cdot dvdz$ ;
28       $dvdz = drdz \cdot dvdz + dsdz \cdot duds + dtdz \cdot dvdz$ ;
29       $dvdz = drdx \cdot dvdz + dsdx \cdot duds + dtdx \cdot dvdz$ ;
30       $dvdz = drdy \cdot dvdz + dsdy \cdot duds + dtdy \cdot dvdz$ ;
31       $dvdz = drdz \cdot dvdz + dsdz \cdot duds + dtdz \cdot dvdz$ ;
32       $u = q[e][0][k][j][i]$ ,  $v = q[e][1][k][j][i]$ ,  $w = q[e][2][k][j][i]$ ;
33       $Qq[e][0][k][j][i] = u \cdot dudx + v \cdot dudy + w \cdot dudz$ ;
34       $Qq[e][1][k][j][i] = u \cdot dvdz + v \cdot dvdz + w \cdot dvdz$ ;
35       $Qq[e][2][k][j][i] = u \cdot dudz + v \cdot dudz + w \cdot dudz$ ;
36    end
37    sync.threads();
38  end
39 end

```

Figure 2: Two-dimensional thread-block for advection.

each for pairwise and crystal router) is less than one second. For this example, the corresponding *gs.op* times are .000325 seconds for pairwise and .0046 seconds for the crystal router.

For device-based implementations of QQ^T , we extend the autotuning approach to test device-to-device transfers (i.e., GPU direct), transfers via the host with buffers packed on the host, and transfers via the host with buffers packed on the GPU. Since overlapped communication and computation is also supported, the determination of the fastest algorithm requires testing under overlapped run conditions, which is enabled through a call-back function that allows the setup to be tested in tandem with execution of the relevant kernel. The overlap works as follows. All elements with nonlocal adjacency connections are evaluated first (i.e., $\tilde{u}^{e_{nl}} = Z^{e_{nl}} u^{e_{nl}}$, where e_{nl} spans the set of elements having nonlocal connections). The nonlocal communication is initiated, and the remaining local products are evaluated. Incoming off-device contributions are then added to the result, \underline{w}_L .

3.3. High-Order Kernels

The $O(N)$ computational intensity of the spectral element method, coupled with minimal indirect addressing, provides significant performance opportunities on GPU architectures. Moreover, for vector- (e.g., velocity-) oriented operations, the $O(N^3)$ geometric factors associated with each element can be reused across each velocity component (e.g., when computing $\nabla \mathbf{u}$ on Ω^e). While the majority of NekRS simulations are run with $N=7$,

it is not uncommon to require $N=11$ – 15 for some applications, implying that the dealiased advection operators will be evaluated on N_q^3 quadrature points, with $N_q = 13$ – 23 . For these cases, a 2D thread structure must be used, as illustrated in Fig. 2. For relatively low N , operations may be organized into 3D thread structures and still be within shared-memory and thread-block limits, as illustrated in Fig. 3.

NVIDIA imposes a hard limit of 1,024 threads per thread block, meaning that a triply nested 3D thread structure of size $N_q \times N_q \times N_q$ mandates $N_q \leq 10$. Therefore, the maximum achievable polynomial order in the 3D thread structure advection operator, illustrated in Algorithm 2, is $N = 9$. The 2D thread structure of Algorithm 1, however, does not reach the 1,024 threads per threadblock restriction until $N_q = 32$. Further, using a 2D thread structure allows for the shared-memory usage to be better optimized. For example, in Algorithm 1, only four 2D shared-memory structures are needed. This approach is already available in libParanumal [8]. A small development building on this earlier work is the addition of using 2D shared-memory structures with a 1D register memory structure. Since each thread (i, j) has its own register data, array lookups of the form $A[*][j][i]$ can be reduced into one-dimensional register array lookups of the form $\mathbf{rA}[*]$, where \mathbf{rA} is a register array for thread (i, j) and $\mathbf{rA}[k] = A[k][j][i]$ corresponds to values of A along the k -index for a fixed (i, j) . A major advantage of this approach is reducing the number of (relatively) slow shared-memory loads by nearly a factor of three. In addition, this helps preserve the 48 kB of shared memory available on the NVIDIA V100, which can hold only 6,144 double-precision words. Because OCCA allows for runtime JIT-compilation of kernels, either the 2D or 3D thread structure kernel may be used for the case $N_q \leq 10$, based on which is more performant.

3.4. Poisson Solve Preconditioning

As noted above, the pressure-Poisson solve is intrinsically the stiffest substep in NS time advancement. It is easy to understand why this is so by considering the example of flow in a pipe. When flow is suddenly forced in at a given flow-rate on one end of the pipe, it must leave at the same rate at the far end of the pipe and must have the same mean value throughout the pipe—all by the end of the current timestep. A consequence of the divergence-free constraint is that the Poisson problem is intrinsically communication intensive—all processors must “know” about an inlet condition, which might be prescribed only on one processor. Fast scalable Poisson solvers are thus of paramount

Algorithm 2: Advection operator, 3D thread configuration

Output: Vector Cq , size $3 \times N_d \times N_p$, indexed as $Cq[element][x/y/z][ix][iy][iz]$,
Input: Vector q , size $3 \times N_d \times N_p$, indexed as $q[element][x/y/z][ix][iy][iz]$,
differentiation matrix $D^{1,D}$, size $N_q \times N_q$, volumetric geometric factors G_v , size $N_d \times N_p \times 12$

```

1 for  $e \in \{0, 1, \dots, N_d - 1\}$  do
2   Allocate  $sU, sV, sW, sD$  as  $(p+1) \cdot (p+1) \cdot (p+1)$  shared memory arrays;
3   for each threads  $i, j, k$  do
4     If  $k == 0$ , load  $D^{1,D}$  into  $sD$ ;
5      $sU[k][j][i] = q[e][0][k][j][i]$ ,  $sV[k][j][i] = q[e][1][k][j][i]$ ;
6      $sW[k][j][i] = q[e][2][k][j][i]$ ;
7   end
8   sync.threads();
9   for each thread  $i, j, k$  do
10     $dudx = \sum_{n=0}^p sD[i][n] \cdot sU[k][j][n]$ ,  $duds = \sum_{n=0}^p sD[j][n] \cdot sU[k][n][i]$ ;
11     $dudt = \sum_{n=0}^p sD[k][n] \cdot sU[n][j][i]$ ,  $dvdz = \sum_{n=0}^p sD[i][n] \cdot sV[k][j][n]$ ;
12     $dvds = \sum_{n=0}^p sD[j][n] \cdot sV[k][n][i]$ ,  $dvdv = \sum_{n=0}^p sD[k][n] \cdot sV[n][j][i]$ ;
13     $dwdz = \sum_{n=0}^p sD[i][n] \cdot sW[k][j][n]$ ,  $dwdv = \sum_{n=0}^p sD[j][n] \cdot sW[k][n][i]$ ;
14     $dwdt = \sum_{n=0}^p sD[k][n] \cdot sW[n][j][i]$ ;
15    Load  $drdx, drdy, drdz, dsdx, dsdy, dsdz, dtdx, dtdy, dtdz$  from  $G_v$ ;
16    // Apply chain rule;
17     $dudx = drdx \cdot dudx + dsdx \cdot duds + dtdx \cdot dudt$ ;
18     $dudy = drdy \cdot dudx + dsdy \cdot duds + dtdy \cdot dudt$ ;
19     $dudz = drdz \cdot dudx + dsdz \cdot duds + dtdz \cdot dudt$ ;
20     $dvdz = drdx \cdot dvdz + dsdx \cdot dvds + dtdx \cdot dvdv$ ;
21     $dvdv = drdy \cdot dvdz + dsdy \cdot dvds + dtdy \cdot dvdv$ ;
22     $dvdz = drdz \cdot dvdz + dsdz \cdot dvds + dtdz \cdot dvdv$ ;
23     $dwdx = drdx \cdot dwdx + dsdx \cdot dwdv + dtdx \cdot dwdt$ ;
24     $dwdy = drdy \cdot dwdx + dsdy \cdot dwdv + dtdy \cdot dwdt$ ;
25     $dwdz = drdz \cdot dwdx + dsdz \cdot dwdv + dtdz \cdot dwdt$ ;
26     $u = q[e][0][k][j][i]$ ,  $v = q[e][1][k][j][i]$ ,  $w = q[e][2][k][j][i]$ ;
27     $Cq[e][0][k][j][i] = u \cdot dudx + v \cdot dudy + w \cdot dudz$ ;
28     $Cq[e][1][k][j][i] = u \cdot dvdz + v \cdot dvdv + w \cdot dwdz$ ;
29     $Cq[e][2][k][j][i] = u \cdot dwdx + v \cdot dwdy + w \cdot dwdz$ ;
30  end
31 end

```

Figure 3: Three-dimensional thread-block for advection.

concern for incompressible simulations. Because the setup is amortized over tens of thousands or hundreds of thousands of timesteps, we care more about *solve* times than *setup* costs.

For CPU-based applications, we have developed a p -multigrid strategy that uses an overlapping additive Schwarz method (ASM) as a smoother [32, 33]. The local solves are effected in $\approx 12E(N+3)^4$ operations by using tensor-contraction-based fast-diagonalization methods (FDMs) [24]. Typical multigrid schedules use approximation orders N , $N/2$, and $N=1$ at successively coarser levels. For modest-sized meshes (e.g., $E < 500K$), the $O(E)$ -sized coarse-grid problem is solved by using a fast direct solver that requires a minimal $2 \log_2 P$ message exchanges [34]. For larger problems on CPU platforms, the coarse-grid problem is solved by using communication-minimal implementations of algebraic multigrid. Here, the adaptability of *gslib* is essential because of the stencil growth in the lower levels of AMG. At each level, communication is effected by the fastest supported algorithm in *gslib*, resulting in a 5- to 10 \times improvement over simply using pairwise exchanges for stencil updates [4]. Our CPU implementation of ASM also uses an additive approach *between* levels, which means that there is only one proper matvec in A per PCG iteration. The idea is that each element of the Krylov subspace should be projected, rather than using cycles for unprojected iterations. On BG/Q, which has hardware support for all-reduce ($< 20\mu s$ for $P = 1M$ [5]), dot products for projection incur only a fraction of a percent of total run time. We use flexible PCG because weighting the ASM, which improves its smoothing properties, introduces a slight asymme-

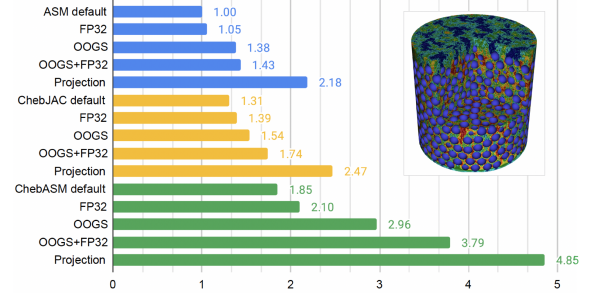


Figure 4: Successive gains in pressure-solve performance for ASM, CHEBY-JAC, and CHEBY-ASM p -multigrid smoothers as a function of implementation options for the 1568-pebble case ($E = 524K$, $N = 7$, characteristics) on 22 nodes of Summit ($n/P = 1.36M$, $P = 132$ V100s).

try in the preconditioner.

On GPUs, the situation is somewhat different. First, the arithmetic for the matvecs and local solves is very fast. Because of other strong-scaling limitations, there are enough elements per rank (typically, $E/P \sim 4000$ – 8000 on the NVIDIA V100 and AMD InstinctTM MI100) to effectively overlap communication with computation on the fine-grid matvecs. Moreover, dot products are not fast on GPUs. A better smoother than straight ASM is consequently more effective. We consider two strategies. The first smoother uses two Chebyshev-accelerated Jacobi (CHEBY-JAC) sweeps, with pre- and postsmoothing at the top two levels. Hypre or parAlmond [35, 36] is used for the $O(E)$ coarse-grid problem. (AMG cannot be applied directly to the dense $A\mathbf{u} = \mathbf{b}$ systems. It can, however, be applied to FEM-based surrogates for A to develop an alternative preconditioning strategy [23, 37, 38].) The second (CHEBY-ASM) applies Chebyshev acceleration to the FDM-based ASM smoother, with the coarse-grid solver unchanged.

The effectiveness of CHEBY-JAC is illustrated in Fig. 4, which contrasts with the baseline ASM results under a succession of algorithmic refinements, including switching to FP32⁴ for certain parts of the preconditioner and/or using an adaptive gather-scatter that chooses between different (device/host) buffer packing and pairwise exchange strategies, and incorporating projection-based initial guesses [39]. The test problem is turbulent flow through a cylinder with 1,568 spherical pebbles at $Re_D = 5000$, run on 22 nodes of Summit. The discretization consists of $E = 524K$ elements of order $N = 7$ ($n = 180M$), with $n/P = 1.36M$, (i.e., beyond the 80% strong-scale limit). Timestepping is based on two-stage 2nd-order characteristics with

⁴FP32 is used only for the smoothing steps. Indirect-addressing overheads result in minimal gains if FP32 is used for the unstructured coarse-grid solve.

NekRS Preconditioning Development, $n = 2,569,495,663$										
Timestepper	Smoother	GPU	E	N	n/GPU	Δt	CFL	v_i	p_i	t_{step} (s)
CHAR-BDF2	RAS	54	524386	7	3.33M	1.0e-03	4.05	4	112	1.64e+00
	ASM	54	524386	7	3.33M	1.0e-03	4.05	4	93	1.44e+00
	CHEBY-JAC	54	524386	7	3.33M	1.0e-03	4.05	4	32	1.04e+00
	CHEBY-RAS	54	524386	7	3.33M	1.0e-03	4.05	4	26	6.56e-01
	CHEBY-ASM	54	524386	7	3.33M	1.0e-03	4.05	4	16	5.03e-01
BDF3-EXT3	RAS	54	524386	7	3.33M	2.5e-04	1.06	2	51	7.54e-01
	ASM	54	524386	7	3.33M	2.5e-04	1.06	2	39	6.38e-01
	CHEBY-JAC	54	524386	7	3.33M	2.5e-04	1.06	2	15	4.96e-01
	CHEBY-RAS	54	524386	7	3.33M	2.5e-04	1.06	2	13	3.43e-01
	CHEBY-ASM	54	524386	7	3.33M	2.5e-04	1.06	2	8	2.59e-01

Table 1: NekRS preconditioner performance comparison on 54 GPUs of Summit for the case of Fig. 4. A restart file at convective time $t=20$ is used to provide a turbulent initial condition. Here, the Courant number (CFL), time per step in seconds (t_{step}), velocity iteration count (v_i), and pressure iteration count (p_i) are all averaged over 100 steps.

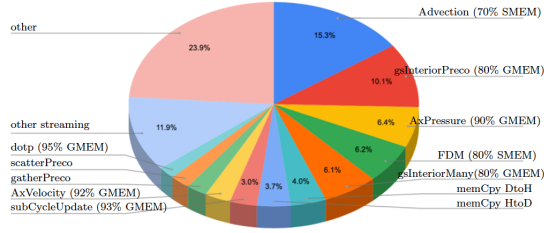


Figure 5: Timing breakdown for the case of Fig. 4.

CFL=4.

Figure 4 shows that the straight CHEBY-JAC strategy (yellow) is 30% faster than baseline ASM (blue). With the enhancements, CHEBY-JAC yields a $2.5\times$ gain over the baseline. The CHEBY-ASM case (green) shows similar improvements ($1.85\times$) in the default configuration but yields an overall $4.85\times$ speedup in the pressure solve when using the adaptive gather-scatter, FP32-based smoothing, and projection-based initial guesses.

In addition to ASM, we explored the potential of restricted additive Schwarz (RAS) [40], in which each subdomain (spectral element) retains its own data after the FDM solve, rather than exchanging and adding (with counting weight w [33]). These results, along with the others, are presented for the 1,568-pebble case on 9 nodes of Summit in Table 1. With $n/P > 3.3\text{M}$, this case is well above 80% parallel efficiency, but the overall relative performance is similar to the 22-node case of Fig. 4. This table also shows the advantage of the characteristics timestepping, which allows a $4\times$ gain in stepsize with only a $2\times$ increase in time per step.

Fig. 5 shows a performance breakdown of the key kernels for the CHEBY-ASM results of Fig. 4. Even with the 4.85 -fold reduction in the pressure solve time, about 25% of the wall-clock time is still directly attributable to the pressure step, as indicated by the Preco, FDM, and AxPressure kernels. The interior-element gather-scatter kernels

(10% of wall time) are limited by and sustain 80% of global memory bandwidth (GMEM). The AxPressure kernel (6.4%) sustains 90% GMEM, and the FDM is limited by and sustains 80% of shared-memory bandwidth (SMEM). For this particular case, the characteristics (8) accounts for about 18% of wall time and sustains 70% of SMEM.

4. Application Performance

In this section, we explore scalability and performance comparisons of NekRS for several applications on a variety of platforms.

4.1. Comparison of Summit and Mira

We begin with comparisons of NekRS on Summit and Nek5000 on Mira for the two configurations depicted in Fig. 6. The first is a 3.2M element LES of a spacer-grid configuration at $Re_D = 14,000$, as considered in [41]. The second is an 8.4M element DNS in a 5×5 rod bundle configuration at $Re_D = 19,000$ [42]. Both were run with approximation order $N = 7$ on 8,192 nodes of the IBM BG/Q, Mira, in -c32 mode ($P = 262144$ MPI ranks). For the NekRS runs, we used tolerances, timestep sizes, and other run parameters that were equivalent to the Mira-based Nek5000 simulations. All cases use dealiasing with $N_q = 12$. The only significant algorithmic difference is in the use of CHEBY-ASM for NekRS versus the default ASM pressure smoother used in Nek5000. The NekRS runs used restart files from the Nek5000 cases, and timings were compared over the same simulation steps. Table 2 summarizes the comparative data.

For the spacer-grid case, the number of points per rank on Mira $n/P = 4116$, which corresponds to a parallel efficiency of $\approx 80\%$ (cf. Fig. 1 in [5]). The corresponding time per step of $t_{step} = 0.68\text{s}$ is thus at a minimum for this efficiency. The average number of ASM-based pressure iterations per step

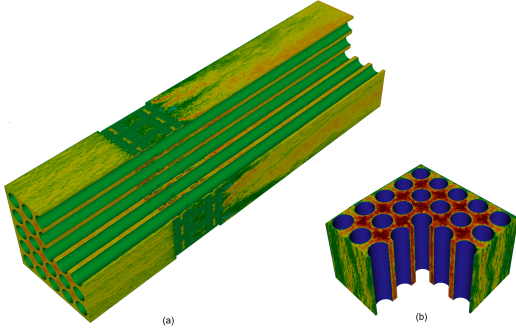


Figure 6: Turbulent velocity snapshots: (a) spacer-grid and (b) DNS 5×5 .

is 8.7, which constitutes about 27% of the wall-clock time. This case used characteristics-based timestepping with a single RK4 substep. On 98 nodes of Summit we have $t_{step} = 0.14s$ — $4.8 \times$ faster than Mira at comparable parallel efficiency. The breakdown of the Summit GPU wall-clock time in this case is roughly 22% for the advection term (8), 24% for the velocity (viscous) solve, and 52% for the pressure solve. (The coarse-grid solve time was 8% of t_{step} .)

For the DNS case, the number of points per rank on Mira $n/P = 10973$, and we can assume that 80% strong-scale limit timings would be realized at $P \approx 5,242,88$ ranks with $t_{step} \approx 0.35s$. In this case, the Summit scaling is not ideal—the best times are at $n/P = 3.9M$, which is significantly larger than what we typically see on Summit. Whether this anomalous behavior is attributable to system noise or to something peculiar about the partitioning is as yet unclear. Nonetheless, $t_{step} = .183s$ is about $2 \times$ faster than the strong-scale limit times on Mira.

4.2. Summit Scaling Performance

Here we consider simulations scaling out to all of Summit for the rod-bundle configurations of Fig. 7. Target geometries for small modular reactors consist of hundreds of 17×17 rod bundles, which total to tens of thousands of long communicating flow channels. For scalability tests, we consider two geometries: a long single 17×17 bundle and a “full-core” collection comprising 37 such bundles that are shorter in length. These cases use inflow-outflow boundary conditions with synthetic vortical flows as initial conditions. The pressure iterations are likely to be a bit higher under fully turbulent conditions, but the overall scaling results for production runs will be similar to what is presented here.

We measured the average wall time per step in seconds, t_{step} , using 101-200 steps for simulations with $Re_D = 5000$. The approximation order

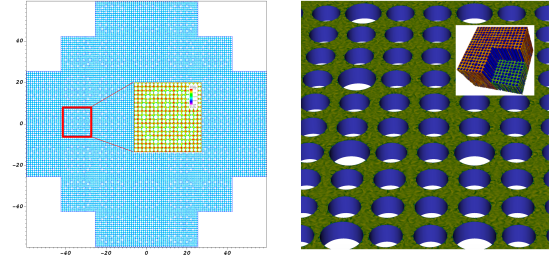


Figure 7: Full-core and 17×17 rod-bundle configurations.

is $N = 7$, and dealiasing is used with $N_q = 9$. We use projection in time, CHEBY+ASM, and flexible PCG for the pressure solves with tolerance $1.e-04$. The velocity solves use Jacobi-PCG with tolerance $1.e-06$. BDF3+EXT3 is used for timestepping with $\Delta t = 3.0e-04$, corresponding to $CFL=0.66$ for the full-core case and $CFL=0.54$ for 17×17 case. We also show the average velocity (v_i) and pressure (p_i) iteration counts over the same simulation interval. The geometries for the weak-scaling studies were generated by extruding layers of 2D elements in the axial flow direction. For strong scaling, we used $E = 175M$, totaling $n = 60$ billion grid points.

The scaling results are presented in Table 3. The pressure iteration counts, $p_i \sim 2$, are lower for these cases than for the pebble cases, which have $p_i \sim 8$ for the same timestepper and preconditioner. The geometric complexity of the rod bundles is relatively mild compared to the pebble beds. Moreover, the synthetic initial condition does not quickly transition to full turbulence. We expect more pressure iterations in the rod case (e.g., $p_i \sim 4-8$) once turbulent flow is established.

We observe that these cases exhibit excellent strong scaling to all of Summit, pointing to $n/P \approx 2.5M$ as the 80% efficiency level for the V100s. Note that, save for the anomalous DNS 5×5 case, this value of n/P is consistent with our previous results. As discussed in [12, 5], the leading indicator of parallel scalability for a given algorithm-architecture coupling is n/P , rather than the number of processing units, P .

The weak-scaling results are less straightforward to interpret. First, we note that they are conducted at $n/P = 2.1M$, which is beyond the strong-scale limit. The data is thus heavily influenced by communication overhead. Nonetheless, the Rod-1717 case exhibits reasonable weak scaling, with only an 18% drop in efficiency over a 53-fold increase in processor count. Weak scaling for the full-core case, however, drops to 54% with only a 17-fold increase in processor count. Part of the performance degradation stems from the very low time exhibited by the full core for

Spacer-Grid, Performance on Mira vs. Summit, $E = 3235953$, $N = 7$, $n = 1.10B$												
System	Code	Device	Node	Rank	R/N	E/R	n/Rank	$t_{step}(s)$	R	R_i	eff %	R**
Mira	Nek5000	CPU	8192	262144	32	12	4116	6.90e-01	1.00	1.00	100	1.00
Summit	Nek5000	CPU	38	1596	42	2027	695446	1.68e+01	1.00	1	100	0.06
			76	3192	42	1013	347723	0.50e+01	2.12	2	106	0.13
			152	6384	42	506	173861	0.23e+01	4.56	4	114	0.29
			304	12768	42	253	86930	0.11e+01	9.64	8	120	0.62
Summit	NekRS	CPU	38	1596	42	2027	695446	0.78e+01	1.00	1	100	0.08
			76	3192	42	1013	347723	0.38e+01	2.01	2	100	0.17
			152	6384	42	506	173861	0.20e+01	3.81	4	95	0.33
			304	12768	42	253	86930	0.11e+01	6.72	8	84	0.59
Summit	NekRS	GPU	38	228	6	14193	4.8M	2.75e-01	1.00	1.00	100	2.46
			60	360	6	8988	3.0M	1.92e-01	1.42	1.57	90	3.52
			76	456	6	7096	2.4M	1.64e-01	1.67	2.00	84	4.14
			98	588	6	5503	1.8M	1.39e-01	1.97	2.57	77	4.87

DNS 5×5 , Performance on Mira vs. Summit, $E = 8387008$, $N = 7$, $n = 2.87B$												
System	Code	Device	Node	Rank	R/N	E/R	n/Rank	$t_{step}(s)$	R	R_i	eff %	R**
Mira	Nek5000	CPU	8192	262144	32	32	10973	7.00e-01	1.00	1.00	100	1.00
Summit	Nek5000	CPU	175	7350	42	1141	391393	3.97e+00	1.00	1.00	100	0.17
			1152	48384	42	173	59456	9.51e-01	4.17	6.58	63	0.73
			2304	96768	42	87	29728	7.30e-01	5.43	13.16	41	0.95
Summit	NekRS	GPU	87	522	6	16067	5.5M	2.30e-01	1.00	1.00	100	3.04
			120	720	6	11648	3.9M	1.83e-01	1.25	1.37	91	3.80
			160	960	6	8736	2.9M	1.49e-01	1.53	1.83	84	4.68
			220	1320	6	6353	2.1M	1.27e-01	1.80	2.52	71	5.48

Table 2: Performance on Mira (Nek5000) vs. Summit CPU (Nek5000) and GPU (NekRS). Timings are in seconds for the wall time per step, t_{step} . R**, the ratio of t_{step} of 8192 nodes on Mira to all others on Summit CPU and GPUs. (top) **Spacer-Grid**: 400 timesteps over simulation time interval [138.0431, 138.0671] with $\Delta t = 6.00e-05$ (CFL=1.74) at $Re_D = 14000$. CHAR=T (1 substep). (bottom) **DNS 5×5** : 400 timesteps over simulation time interval [59.71, 59.78] with $\Delta t = 1.9e-04$ (CFL=0.32) at $Re_D = 19000$. BDF2+EXT2.

$P = 1626$ V100s, which at .066 s is substantially below the best time of .086 s for the Rod-1717 case. By contrast, the weak-scale time for Full-Core is 20% higher than Rod-1717 for $P = 27648$. Here, the discrepancy is closely correlated with the maximum number of neighbors that any processor (GPU) is connected to in the QQ^T graph, which is indicated by n_{gh} in the last column of Table 3. The full-core geometry is a relatively flat graph, and it appears that the partitioning for $P = 271$ resulted in a 2D decomposition, with a maximum of 9 neighbors. When this mesh is partitioned further by RSB, it results in neighbor counts that are twice those of the Rod-1717 case. These results, coupled with the high latency of GPUs (as indicated by the large $n_{0.8}$ values), suggests that partitioning with the aim of minimizing the number of neighbors, rather than the data volume, might be beneficial in this context.

4.3. Performance on Other GPU Architectures

Here we contrast our baseline Summit performance with recently deployed NVIDIA A100 and AMD MI100 node architectures.

The first case is the NekRS turbulent pipe flow example with a synthetic initial condition,

$Re_D = DU/\nu = 19,000$, $E = 6840$, $N = 7$, and $n = n/P = 2,346,120$ (which is near the strong-scale performance limit). We use characteristics with two RK4 substeps, dealiasing with $N_q = 10$, and timestep size $\Delta t = .006D/U$, where U is the mean velocity and D is the diameter. The pressure solve uses projection in time, CHEBY-ASM smoothing for flexible CG, and a tolerance of $1.e-04$. The Jacobi-PCG tolerance for velocity is $1.e-06$. Timings are in seconds for the averaged-walltime per step, t_{step} , using steps 101–200.

Table 4 provides preliminary single-GPU results for AMD GPUs against the Summit baseline. The AMD MI60 and MI100 results were obtained on the HPE *Tulip* platform while the NVIDIA A100 runs were done on ALCF’s *Theta-GPU*. Performance on a single CPU core and multiple cores on Summit IBM Power9 is also presented. The AMD interface is provided by OCCA’s HIP backend. To produce optimized code, hand-tuning is still required for good performance each of the devices.

In Table 4 we see that Summit is slightly faster than the Tulip V100, which might be expected given that Summit uses NVLink vs. the PCI-E interconnect on Tulip. The NVIDIA A100 clearly is outperforming the V100 by $1.5\times$, which is in line

NekRS Strong Scaling on Full Summit, $N = 7$, $n = 59B$ (full-core) and $n = 60B$ (rod-1717)												
Case	Node	GPU	E	E/GPU	n/GPU	v_i	p_i	$t_{\text{step}}(s)$	R	R_{ideal}	P_{eff}	ngh
Full-Core	1810	10860	174233000	16044	5.5M	3	2	2.17e-01	1.00	1.00	100	41
	2715	16290	174233000	10696	3.6M	3	2	1.39e-01	1.55	1.50	103	26
	3620	21720	174233000	8021	2.7M	3	2	1.18e-01	1.84	2.00	92	32
	4525	27150	174233000	6417	2.2M	3	2	1.22e-01	1.76	2.50	70	47
	4608	27648	174233000	6301	2.1M	3	2	1.21e-01	1.79	2.54	70	40
Rod1717	1810	10860	175618000	16171	5.5M	3	2	1.855e-01	1.00	1.00	100	25
	2536	15216	175618000	11542	3.9M	3	2	1.517e-01	1.22	1.40	87	25
	3620	21720	175618000	8085	2.7M	3	2	1.120e-01	1.65	2.00	82	26
	4180	25080	175618000	7002	2.4M	3	2	1.128e-01	1.64	2.30	71	28
	4608	27648	175618000	6351	2.1M	3	2	1.038e-01	1.78	2.54	70	29

NekRS Weak Scaling on Full Summit, $N = 7$												
Case	Node	GPU	E	E/GPU	n/GPU	v_i	p_i	$t_{\text{step}}(s)$	R	R_{ideal}	P_{eff}	ngh
Full-Core	271	1626	10249000	6303	2.1M	3	2	6.58e-02	1.00	1.00	100	9
	813	4878	30747000	6303	2.1M	3	2	9.68e-02	0.67	1.00	67.9	12
	1626	9756	61494000	6303	2.1M	3	2	1.05e-01	0.62	1.00	62.5	44
	3253	19518	122988000	6301	2.1M	3	2	1.18e-01	0.55	1.00	55.8	56
	4608	27648	174233000	6301	2.1M	3	2	1.21e-01	0.54	1.00	54.0	40
Rod-1717	87	522	3324000	6367	2.1M	3	2	8.57e-02	1.00	1.00	100	25
	320	1920	12188000	6347	2.1M	3	2	8.67e-02	0.98	1.00	98.7	25
	800	4800	30470000	6347	2.1M	3	2	9.11e-02	0.94	1.00	94.0	25
	1600	9600	60940000	6347	2.1M	3	2	9.33e-02	0.91	1.00	91.8	27
	3200	19200	121880000	6347	2.1M	3	2	9.71e-02	0.88	1.00	88.2	25
	4608	27648	175618000	6351	2.1M	3	2	1.03e-01	0.82	1.00	82.5	29

Table 3: Summit strong and weak scalings for full-core and 17×17 rod bundle geometries.

with the improved memory bandwidth of the A100. The (early) MI100 and MI60 GPUs are delivering 85 and 60% of Summit’s V100, respectively. We also observe that Summit’s single V100 is comparable to 336 CPU cores on 8 nodes, while it is only 248× faster than a single CPU. The implication is that the parallel efficiency for NekRS on Summit’s Power9 CPUs is 74% with $n/P = 6615$.

In our second comparative study we consider multi-GPU, single-node performance for the Summit V100 vs. ThetaGPU A100s for the atmospheric boundary layer (ABL) example of Fig. 8 (left). The domain is doubly-periodic ($400\text{m} \times 400\text{m} \times 400\text{m}$) with $E = 32768$ spectral elements of order $N = 7$ (i.e., $n=11.2\text{M}$). A geostrophic wind speed of 8 m/s and reference potential temperature of 263.5K are prescribed with a no-slip condition at the lower wall. A restart file at convective time $t=1710$ is used to provide a turbulent initial condition, corresponding to the physical convective time of 6 hours. Single-node scaling shows the 80% strong-scale limit to be 1.8M points/GPU for both the V100 and A100, with the A100 running at .055 s/step and 1.55 times faster than the V100. We remark that the low strong-scale limit of $n/P = 1.8\text{M}$ for these single-node studies is more likely due to the low number of neighbors (at most 6 or 8 for the V100 or A100, respectively), rather than a high internode communication cost in the multinode cases.

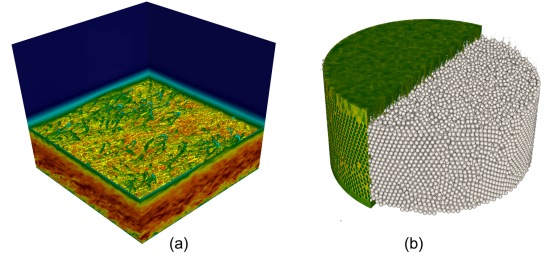


Figure 8: Turbulence in stratified ABL and 44257-pebble configurations.

We close with a final example illustrating the potential of GPU-based simulations of turbulence in HPC settings. The example, shown on the right in Fig. 8, is a 44,257-pebble configuration, which is a prototype for pebble-bed reactors that will ultimately hold hundreds of thousands of spherical pebbles. This example has 13M elements of order $N = 7$ ($n = 4.5\text{B}$). The all-hex mesh was developed from an initial Voronoi tessellation of the sphere centers, with each Voronoi facet tessellated into quadrilaterals that are then projected onto the sphere surfaces in order to sweep out a hexadral volume. Edge collapse, mesh refinement, and mesh smoothing tools ensure a high-quality mesh for the fluid flow in the void space. Timestepping is based on 2nd-order characteristics with a single RK4 subcycle, $N_q = 11$, and $\Delta t = 3.e-4$ (CFL=4). The average number of velocity iterations (tol=1.e-6)

is 3, and the average number of pressure iterations is 18 ($\text{tol}=1.e-4$). On 1,788 V100s ($n/P = 2.5M$), $t_{\text{step}}=.54$ s. The timing breakdown is 10% for advection (8), 6% for the velocity solve (10), and 84% for the pressure solve. The pressure solve is broken down into percentage of total simulation time: 16% for the coarse-grid solve and 56% for the remainder of the preconditioner. While the time per step is higher than for the other cases, these simulations strong-scale well; and the target configuration of 300,000 pebbles, which will require about 30B grid points, is well within the current performance envelope on Summit.

5. Conclusions

We developed an C++/OCCA-based open-source Navier–Stokes solver for GPUs that leverages prior scaling development in Nek5000 and high-performance kernels developed in libParanumal. We discuss its performance and scalability on leadership computing platforms. The solver is based on 2nd- or 3rd-order timesplitting of the incompressible Navier–Stokes equations with an exponentially convergent spectral-element-based discretization in space. We demonstrate weak- and strong-scaling up to 27,648 V100 GPUs on OLCF’s Summit system for reactor geometries with problem sizes of more than 175M spectral elements ($n = 60B$ gridpoints). Performance results show that NekRS sustains 80–90% of the realizable (bandwidth-limited) peak and that 80% parallel efficiency on Summit is realized for local problem sizes of $n/P \approx 2.5M$, where P is the number of GPUs employed for the simulation. Preliminary timing data for NVIDIA A100s and AMD MI100s are also presented.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

This research is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation’s exascale computing imperative.

The research 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. This

research also 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 DE-AC05-00OR22725. Support was also given by the Frontier Center of Excellence.

References

- [1] Nek: Open source, highly scalable and portable spectral element code, <http://nek5000.mcs.anl.gov> (2020).
- [2] P. Fischer, E. Rønquist, D. Dewey, A. Patera, Spectral element methods: Algorithms and architectures, in: Proc. of the First Int. Conf. on Domain Decomposition Methods for Partial Differential Equations, SIAM, 1988, pp. 173–197.
- [3] H. Tufo, P. Fischer, Terascale spectral element algorithms and implementations, in: Proc. of the ACM/IEEE SC99 Conf. on High Performance Networking and Computing, Gordon Bell Prize, IEEE Computer Soc., CDROM, 1999.
- [4] P. Fischer, J. Lottes, W. Pointer, A. Siegel, Petascale algorithms for reactor hydrodynamics, J. Phys. Conf. Series 125 (2008) 012076.
- [5] P. Fischer, K. Heisey, M. Min, Scaling limits for PDE-based simulation (invited), in: 22nd AIAA Computational Fluid Dynamics Conference, AIAA Aviation, AIAA 2015-3049, 2015.
- [6] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, P. Fischer, Openacc acceleration of the nek5000 spectral element code, Int. J. of High Perf. Comp. Appl. 1094342015576846.
- [7] M. Otten, J. Gong, A. Mametjanov, A. Vose, J. Levesque, P. Fischer, M. Min, An MPI/OpenACC implementation of a high order electromagnetics solver with GPUDirect communication, Int. J. High Perf. Comput. Appl.
- [8] K. Świrydowicz, N. Chalmers, A. Karakus, T. Warburton, Acceleration of tensor-product operations for high-order finite element methods, Int. J. of High Performance Comput. App. 33 (4) (2019) 735–757.
- [9] A. Karakus, N. Chalmers, K. Świrydowicz, T. Warburton, A gpu accelerated discontinuous galerkin incompressible flow solver, J. Comp. Phys. 390 (2019) 380–404.
- [10] D. Medina, Okl: a unified language for parallel architectures, Ph.D. thesis, Rice University (2015).
- [11] D. S. Medina, A. St-Cyr, T. Warburton, OCCA: A unified approach to multi-threading languages, preprint arXiv:1403.0968.
- [12] P. Fischer, M. Min, T. Rathnayake, S. Dutta, T. Kolev, V. Dobrev, J.-S. Camier, M. Kronbichler, T. Warburton, K. Świrydowicz, J. Brown, Scalability of high-performance PDE solvers, IJHPCA 34, 5 (2020) 562–586.
- [13] gslib: Gather-scatter library (2020). <http://github.com/Nek5000/gslib>
- [14] H. Kreiss, J. Oliger, Comparison of accurate methods for the integration of hyperbolic problems, Tellus 24 (1972) 199–215.
- [15] Y. Maday, A. Patera, E. Rønquist, An operator-integration-factor splitting method for time-dependent problems: Application to incompressible fluid flow, J. Sci. Comput. 5 (1990) 263–292.
- [16] S. Patel, P. Fischer, M. Min, A. Tomboulides, A characteristic-based, spectral element method for moving-domain problems, Under Review.
- [17] J. Malm, P. Schlatter, P. Fischer, D. Henningson, Stabilization of the spectral-element method in convection

NekRS single GPU performance for turbulent pipe flow simulation, $n = 2, 222, 640$									
system	device	API	rank	node	E	N	n/rank	$t_{\text{step}}(s)$	R
OLCF Summit	NVIDIA V100	CUDA	1	1	6480	7	2.22M	8.51e-02	1.00e+00
ALCF Theta-GPU	NVIDIA A100	CUDA	1	1	6480	7	2.22M	5.59e-02	1.52e+00
HPE Tulip	NVIDIA V100	CUDA	1	1	6480	7	2.22M	8.85e-02	9.61e-01
HPE Tulip	AMD MI100	HIP	1	1	6480	7	2.22M	9.96e-02	8.54e-01
HPE Tulip	AMD MI60	HIP	1	1	6480	7	2.22M	1.41e-01	6.03e-01
OLCF Summit	IBM Power9	C	1	1	6480	7	2.22M	1.99e+01	4.27e-03
OLCF Summit	IBM Power9	C	336	8	6480	7	6615	8.02e-02	1.06e+00

Table 4: NekRS baseline performance on a single GPU of HPE Tulip AMD InstinctTM MI100, AMD Radeon InstinctTM MI60, and Nvidia V100 PCIe and ALCF/Theta-GPU Nvidia A100 SXM2, compared to OLCF/Summit Nvidia V100 SXM2, for turbulent pipe flow simulation with $Re = 19,000$, $E = 6840$, and $N = 7$.

NekRS on a Single Node for Atmospheric Boundary Layer Model, $E = 32768$, $N = 7$, $n = 11, 239, 424$										
gpu	E/gpu	n/gpu	V100	R	$P_{\text{eff}}(\%)$	A100	R	$P_{\text{eff}}(\%)$	R_{ideal}	$R_{V/A}$
2	16384	5.6M	2.050e-01	1.00	100	1.341e-01	1.00	100	1.00	1.52
3	10923	3.7M	1.464e-01	1.40	93.3	9.544e-02	1.40	93.7	1.50	1.53
4	8192	2.8M	1.171e-01	1.75	87.5	7.485e-02	1.79	89.6	2.00	1.56
5	6553	2.2M	9.898e-02	2.07	82.8	6.371e-02	2.10	84.2	2.50	1.55
6	5461	1.8M	8.575e-02	2.39	79.7	5.519e-02	2.43	81.0	3.00	1.55
7	4681	1.6M	-	-	-	5.080e-02	2.64	75.4	3.50	-
8	4096	1.4M	-	-	-	4.545e-02	2.95	73.8	4.00	-

Table 5: Atmospheric boundary layer baseline performance on Summit V100 SXM2 and ThetaGPU A100 SXM4.

- dominated flows by recovery of skew symmetry, J. Sci. Comp. 57 (2013) 254–277.
- [18] S. Orszag, M. Israeli, M. Deville, Boundary conditions for incompressible flows., J. Sci. Comp. 1 (1986) 75–111.
- [19] A. Tomboulides, M. Israeli, G. Karniadakis, Efficient removal of boundary-divergence errors in time-splitting methods, J. Sci. Comput. 4 (1989) 291–308.
- [20] A. Tomboulides, J. Lee, S. Orszag, Numerical simulation of low Mach number reactive flows, J. of Sci. Comp. 12 (June 1997) 139–167.
- [21] J. Guermond, P. Mineev, J. Shen, An overview of projection methods for incompressible flows, Comput. Methods Appl. Mech. Engrg. 195 (2006) 6011–6045.
- [22] A. Patera, A spectral element method for fluid dynamics : laminar flow in a channel expansion, J. Comput. Phys. 54 (1984) 468–488.
- [23] S. Orszag, Spectral methods for problems in complex geometry, J. Comput. Phys. 37 (1980) 70–92.
- [24] M. Deville, P. Fischer, E. Mund, High-order methods for incompressible fluid flow, Cambridge University Press, Cambridge, 2002 (500 pages).
- [25] E. Rønquist, A. Patera, A Legendre spectral element method for the Stefan problem, Int. J. Numer. Meth. Eng. 24 (1987) 2273–2299.
- [26] N. Chalmers, A. Karakus, A. P. Austin, K. Swirydowicz, T. Warburton, [libParanumal](http://github.com/libParanumal/libparanumal) (2020). <http://github.com/libParanumal/libparanumal>
- [27] N. Chalmers, T. Warburton, [streamParanumal](http://github.com/paranumal/streamparanumal). <http://github.com/paranumal/streamparanumal>
- [28] OCCA: Lightweight performance portability library, <http://libocca.org> (2020).
- [29] A. Pothen, H. Simon, K. Liou, Partitioning sparse matrices with eigenvectors of graphs, SIAM J. Matrix Anal. Appl. 11 (1990) 430–452.
- [30] O. E. Livne, A. Brandt, Lean algebraic multigrid (lamg): Fast graph Laplacian linear solver (2012). <http://arxiv.org/abs/1108.1310> arXiv:1108.1310.
- [31] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, D. W. Walker, Solving Problems on Concurrent Processors, Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [32] P. Fischer, J. Lottes, Hybrid Schwarz-multigrid methods for the spectral element method: Extensions to Navier-Stokes, in: R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Widlund, J. Xu (Eds.), Domain Decomposition Methods in Science and Engineering Series, Springer, Berlin, 2004.
- [33] J. W. Lottes, P. F. Fischer, Hybrid multigrid/Schwarz algorithms for the spectral element method, J. Sci. Comput. 24 (2005) 45–78.
- [34] H. Tufo, P. Fischer, Fast parallel direct solvers for coarse-grid problems, J. Parallel Distrib. Comput. 61 (2001) 151–177.
- [35] R. Gandham, K. Esler, YongpengZhang, A gpu accelerated aggregation algebraic multigrid method, Comp. & Math. with App. 68 (2014) 1151–1160.
- [36] J. F. Remacle, R. Gandham, T. Warburton, Gpu accelerated spectral finite elements on all-hex meshes 324 (2016) 246–257.
- [37] C. Canuto, P. Gervasio, A. Quarteroni, Finite-element preconditioning of G-NI spectral methods, SIAM J. Sci. Comput. 31 (2010) 4422–44251.
- [38] P. Bello-Maldonado, P. Fischer, Scalable low-order finite element preconditioners for high-order spectral element Poisson solvers, SIAM J. Sci. Comput. 41 (2019) S2–S18.
- [39] P. Fischer, Projection techniques for iterative solution of $Ax = b$ with successive right-hand sides, Comput. Methods Appl. Mech. Engrg. 163 (1998) 193–204.
- [40] X. chuan Cai, M. Sarkis, A restricted additive Schwarz preconditioner for general sparse linear systems, SIAM J. Sci. Comput. 21 (1999) 792–797.
- [41] G. Busco, E. Merzari, Y. A. Hassan, Invariant analysis of the Reynolds stress tensor for a nuclear fuel assembly with spacer grid and split type vanes., Int. J. of Heat and Fluid Flow 77 (2019) 144–156.
- [42] A. Kraus, E. Merzari, T. Norddine, O. Marin, S. Benhamadouche, Direct numerical simulation of fluid flow in a 5x5 square rod bundle using Nek5000, arXiv preprint arXiv:2007.00630.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory (“Argonne”). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan. <http://energy.gov/downloads/doe-public-access-plan>.

ECP Disclaimer: This research is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation’s exascale computing imperative.