Nek5000/RS Performance on Advanced GPU Architectures

Mathematics and Computer Science
About Argonne National Laboratory
Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory’s main facility is outside Chicago, at 9700 South Cass Avenue, Lemont, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY


Reports not in digital format may be purchased by the public from the

National Technical Information Service (NTIS):
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703) 605-6000
Fax: (703) 605-6900
Email: orders@ntis.gov

Reports not in digital format are available to DOE and DOE contractors from the

Office of Scientific and Technical Information (OSTI):
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: reports@osti.gov

Disclaimer
This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.
Nek5000/RS Performance on Advanced GPU Architectures

prepared by

Misun Min\(^1\), Yu-Hsiang Lan\(^{1,2}\), Paul Fischer\(^{1,2,3}\), Thilina Rathnayake\(^2\), John Holmen\(^4\)

\(^1\) Mathematics and Computer Science Division, Argonne National Laboratory
\(^2\) Department of Computer Science, University of Illinois Urbana-Champaign
\(^3\) Department of Mechanical Science & Engineering, University of Illinois Urbana-Champaign
\(^4\) Oak Ridge Leadership Computing Facility, Oak Ridge National Laboratory

September 30, 2022
# Contents

Executive Summary ................................................................. ii

1 Introduction ........................................................................... 1
   1.1 Performance Metrics ....................................................... 2
   1.2 Test Cases ......................................................................... 4

2 NekRS Performance on a Single GPU ........................................ 5

3 NekRS Performance on Frontier vs. Crusher ............................... 5

4 NekRS performance on Summit, ThetaGPU, Perlmutter, Polaris, Crusher and Frontier 11

5 Discussion ............................................................................... 15
   5.1 Performance on Summit V100 16GB vs. 32GB .................... 15
   5.2 Performance on Crusher with Rank Dependency .................... 16
   5.3 Performance on Polaris with GPUDirect vs. without GPUDirect 16
   5.4 Performance on Perlmutter with Slingshot 10 vs. Slingshot 11 . . 16

6 Conclusions .............................................................................. 22

Acknowledgments ........................................................................ 22

References .................................................................................. 23
Executive Summary

We demonstrate NekRS performance results on various advanced GPU architectures. NekRS is a GPU-accelerated version of Nek5000 that is targeting high performance on forthcoming exascale platforms. It is being developed in DOE’s Center of Efficient Exascale Discretizations (CEED), which is one of the co-design centers under the Exascale Computing Project (ECP). In this report, we consider Frontier, Crusher, Spock, Polaris, Perlmutter, ThetaGPU and Summit. Simulations are performed using ExaSMR’s 17×17 rod-bundle geometries with different problem sizes. The report focuses on strong scaling performance and analysis. Many of the results shown in this report are the outcome from participation in the ALCF GPU Hackathon on Polaris, which was held on 7/19/22, and 7/26–7/28/22. Members of the Nek5000/RS team included Misun Min, Yu-Hsiang Lan, Paul Fischer and Thilina Rathnayake. Mentors were Kris Rowe (ALCF) and Peng Wang (NVIDIA). Also presented in this report are results on Frontier, which were obtained in collaboration with John Holmen at OLCF.
1 Introduction

As part of its Exascale Computing Project, the U.S. Department of Energy leadership computing facilities will deploy platforms capable of reaching $> 1$ exaFLOPS ($10^{18}$ floating point operations per second) in 2022–2023. These highly parallel computers, and their pre-exascale predecessors, feature $\approx 10^3$–$10^4$ nodes, each equipped with powerful CPUs and anywhere from 4 to 8 accelerators (i.e., GPUs), which provide the bulk of the compute power. For reasons of efficiency, a favored programming model for these architectures is to assign a single process (i.e., MPI rank) to each GPU (or GPU processing unit, such as a GCD on the AMD MI250X or a tile on the Intel PVC) and execute across the GPUs using a private distributed-memory programming model. With $P = 10^3$–$10^5$ MPI ranks, this approach affords a significant amount of internode parallelism and contention-free bandwidth with no increase in memory-access latency, save for the relatively sparse internode communication that is handled by MPI.

This report describes performance results for the open source thermal-fluids simulation code, Nek5000/RS [1, 2], on several of the DOE’s recently-installed HPC platforms. Many of the results are the outcome from participation of ALCF GPU Hackathon on Polaris, which was held on 7/19/22, and 7/26–7/28/22, where four members of Nek5000/RS team (M. Min, Y. Lan, P. Fischer, T. Rathnayake), with Kris Rowe (ALCF) and Peng Wang (NVIDIA) as mentors, participated. The results on Frontier are obtained in collaboration with John Holmen at OLCF. NekRS is a GPU-oriented version of Nek5000 that was developed under DOE’s Center for Efficient Exascale Discretizations (CEED). The principal developers are Stefan Kerkemeier (K2/ANL), Malachi Phillips (UIUC), Thilina Rathnayake (UIUC), Misun Min (ANL), Yu-Hsiang Lan (UIUC/ANL), Paul Fischer (UIUC/ANL), Ananias Tomboulides (AUT/ANL), and Elia Merzari (PSU). For portability, all the GPU kernels are written in OCCA [3, 4], which was developed by Tim Warburton’s group.
at Virginia Tech. and Rice University. Many of the high performance kernels came out of the libParanumal library [5], also developed by Warburton and co-workers and supported by CEED.

1.1 Performance Metrics

Of great interest to computational scientists is the speed that can be realized for a particular application for a given architecture. (Here, an application is a particular problem that uses Nek5000/RS, which is an application code.) For example, one frequently needs to estimate the number of node-hours and number of wall-clock hours that might be required for a large simulation campaign. A common metric, which is very much case-specific, is the number of degrees-of-freedom (dofs) per second that can be realized on a platform, or perhaps the number of dofs per second per accelerator (i.e., per MPI rank¹). In the sequel, we will assign MDOFS to the quantity, “millions of dofs per second per rank”. The case-specificity aspect of MDOFs is that one can realize a much larger MDOFS value for, say, linear solution of $Ax = b$ than would be possible for an incompressible Navier-Stokes solver.

Despite the large variance in MDOFS from one problem class to the next, it is nonetheless a worthy metric when making platform-to-platform comparisons. A related metric is the time-to-solution or, in the case of a time-stepping simulation code that could take an arbitrarily large number of steps, the time-per-step, $t_{\text{step}}$, which we measure in seconds. Even for a given code and architecture this latter quantity is subject to significant variability because some problems or computational meshes are more ill-conditioned than others, which leads to higher iteration counts in the linear solvers (e.g., in the pressure solve for an incompressible flow simulation) and hence a longer time-per-step.

MDOFS and $t_{\text{step}}$ are dependent, or output, parameters. For a fixed platform, code, and problem, users still have two independent parameters at their disposal, $n$, the problem size or number of dofs,² and $P$, the number of ranks (here, accelerator devices) to use. For a fixed problem size, $n$ (which is what a user typically has), there is only one variable, namely $P$. If a user is contemplating a simulation campaign, they will often be interested in predicting performance over a range of $n$. Under the given conditions, we will see that the most important quantity in predicting performance is

$$\frac{n}{P} = \text{the number of grid points per device}, \quad (1)$$

where we reiterate that we are assigning a single MPI rank to each device.

Two other dependent quantities of interest are parallel efficiency, $\eta_P$, and realized FLOPS, the measurable number of 64-bit floating point operations per second. We typically will report FLOPS per rank, which is more universal than aggregate FLOPS. Also, in cases where mixed precision is used (e.g., when 32-bit arithmetic is used in a preconditioner) we count the FP32 flops as a half-flop

¹We prefer dofs-per-second per rank because AMD’s MI250X has two compute units (GCDs) per GPU and Aurora’s Intel has two tiles per PVC—users view these as two processors.

²For fluid flow simulations, some authors set $n$ to be 4 times the number of grid points because there are typically 3 velocity components and one pressure unknown at each grid point. We prefer to take $n$ to be the number of grid points. The problems are generally large enough that we do not need to distinguish between interior points where the solution is unknown and surface points where boundary data is prescribed. We simply set $n$ to represent the union of these sets given that some quantities might have Neumann conditions on the boundary while others have Dirichlet.
A typical definition of (strong-scaling) parallel efficiency is

$$\eta_P := \frac{P_0 t_{\text{step}}(P_0)}{P t_{\text{step}}(P)} = \frac{\text{MDOFS}(P)}{\text{MDOFS}(P_0)},$$

(2)

where \( t_{\text{step}}(P) \) is the time-per-step when running a fixed problem of fixed size, \( n \), on \( P \) ranks and MDOFS\((P)\) is the corresponding number of mega-dofs per second per rank. Here, \( P_0 \) is the smallest number of ranks that is able to hold the problem. On some architectures, the amount of memory per GPU is relatively small, which prevents extensive strong-scaling studies. (From a user’s perspective, however, this is potentially a happy circumstance since there is “just enough” memory and not a lot of idle memory that incurs unnecessary capital and power overhead.)

An alternative definition of parallel efficiency is given by the relationship,

$$S_P = \eta_P P S_1.$$

(3)

Here, \( S_p \) is the speed, which could be measured in total (not per rank) FLOPS or MDOFS. This definition is equivalent to (2) when \( P_0 = 1 \). The utility of this definition is that one can consider it for either weak- (fixed \( n/P \)) or strong- (fixed \( n \)) scaling studies. If FLOPS are used, it’s relatively easy to get FLOPS on one rank for a smaller version of the application problem. (Although that might not be a useful starting point in the exascale era given that there is no exascale problem that comes anywhere close to fitting on one rank.)

What (3) tells us is that the speed on \( P \) ranks should be \( \approx P \) times the speed on 1 rank, if only we can sustain close to unity efficiency, \( \eta_P \approx 1 \). We remark that HPC users generally want to run as fast as possible, particularly for large campaigns, so they want \( P \gg 1 \). However, they also need to efficiently use their allocation, which implies \( \eta_P \approx 1 \). This latter condition places a constraint on time-to-solution that is generally stronger than the unconstrained “min-time-to-solution” result. In our studies, we will assume that the user is willing to run at 80% parallel efficiency, \( \eta_P = 0.8 \). Of course, other values are possible and a user can change \( P \) and, hence, \( \eta_P \) on a submission-by-submission basis for each case that they run in a given campaign. However, \( \eta_P = 0.8 \) is a reasonable starting point for analysis.

The first analysis question we address is, For a fixed problem size \( n \), how many ranks can we use before \( \eta_P < 0.8 \)? An accompanying question is, What is the time per step at that value? The key to this first question is that parallel efficiency typically drops as the local amount of work, \( n/P \) tends towards zero. So, fixing \( \eta_P = 0.8 \) implies \( n/P \) is a fixed value (for a given fixed-sized problem with \( P \) varying). We denote this value as \( n_{0.8} \). It is the number of points per rank where the application realizes 80% efficiency, which is where we anticipate that users will typically run. With this definition, we can address the question of the expected \( t_{\text{step}} \) under these conditions. Assume that a given problem requires a certain amount of work, \( W \), that is measured in total number of floating-point operations (flops). Usually, \( W \sim Cn \), where \( C \) is an \( n \)-independent constant, which implies that, to leading order, the amount of work scales with \( n \). On \( P \) processors, we therefore can expect that,

$$t_{\text{step}} = \frac{W}{S_P} = \frac{C n}{\eta_P P S_1}.$$

(4)
We define \( t_{0.8} \) to be the value of \( t_{\text{step}} \) at 80% efficiency,

\[
    t_{0.8} = \frac{C \ n/P}{0.8 \ S_1} = \left( \frac{C}{0.8} \right) \frac{n_{0.8}}{S_1}.
\]  \( \tag{5} \)

We note that (4)–(5) is predicated on \( \eta_P \) being strongly dependent on \((n/P)\) with no direct \( P \) dependence. There are times when there is a weak \( P \) dependence, particularly for \( P > 10^4 \). In this case, one can simply modify the analysis to have a \( P \)-dependent \( n_{0.8} \).

We see from (5) that the time per step is governed by the speed on a single rank, \( S_1 \) (larger is better), and the amount of work on a single rank, \( n_{0.8} \) (smaller is better), where 80% efficiency is realized. If a new platform comes out with a \( 2 \times \) increase in \( S_1 \) but a \( 4 \times \) increase in \( n_{0.8} \), then the net time to solution \textit{increases by} \( 2 \times \). In HPC, it is the \textit{ratio}, \( n_{0.8}/S_1 \), that is critical to fast time to solution. Much of this analysis can be found in \([6, 7]\). Communication overhead on GPU-based architectures is discussed in \([8]\).

A typical use case for (5) is that a user knows \( n \), which is the number of gridpoints required to resolve a given simulation, and wants to know how many processors will be required to efficiently solve this problem and how long it will take to execute. They also know \( n_{0.8} \) from scaling studies of the type provided here. From that, they can determine

\[
    P_{0.8} = \frac{n}{n_{0.8}},
\]  \( \tag{6} \)

which is the maximum number of ranks that can be employed while sustaining 80% efficiency. Their time-per-step will be \( t_{0.8} \) and the total required node hours will be

\[
    \text{node hours} \approx \frac{P_{0.8}}{\text{ranks-per-node}} \times \frac{N_{\text{steps}} t_{0.8}}{3600 \text{ s/hour}},
\]  \( \tag{7} \)

where \( N_{\text{steps}} \) is the estimated number of timesteps.

### 1.2 Test Cases

In the following sections, we characterize these relevant parameters for NekRS across several of DOE’s pre-exascale and exascale platforms, including Frontier, Crusher, Spock, Polaris, Perlmutter, ThetaGPU and Summit. Simulations are performed using ExaSMR’s \( 17 \times 17 \) rod-bundle geometry, illustrated in Fig. 1. This geometry is periodic in the axial (vertical) flow direction, which allows us to weak-scale the problem by adding more layers of elements in the \( z \) direction. (The model problem is essentially homogeneous in \( z \).) Each case starts with a pseudo-turbulent initial condition so that the iterative solvers, which compute only the change in the solution on each step, are not working on void solutions. Most of the cases are run under precisely the same conditions of timestep size, iteration tolerances, and averaging procedures, which are provided case-by-case in the sequel.

We remark that the following performance summaries are for full Navier-Stokes solution times. We present a few plots that reflect work in salient kernels, such as the advection operator, which is largely communication-free, and the pressure-Poisson coarse-grid solve, which is highly communication-intensive. Detailed, kernel-by-kernel, break downs are presented in \([9, 10]\) and are available in every logfile generated by NekRS. Further, it is important to note that NekRS
Table 1: NekRS performance on various architectures using a single GPU.

<table>
<thead>
<tr>
<th>System</th>
<th>gpu</th>
<th>Device</th>
<th>API</th>
<th>(v_i)</th>
<th>(p_i)</th>
<th>(t_{step}) (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summit</td>
<td>1</td>
<td>16GB V100 GPU</td>
<td>CUDA</td>
<td>4</td>
<td>1</td>
<td>7.98e-02</td>
<td>1.00</td>
</tr>
<tr>
<td>Spock</td>
<td>1</td>
<td>32GB MI100 GPU</td>
<td>HIP</td>
<td>4</td>
<td>1</td>
<td>4.17e-02</td>
<td>0.84</td>
</tr>
<tr>
<td>Crusher</td>
<td>1</td>
<td>64GB MI250X (1 GCD)</td>
<td>HIP</td>
<td>4</td>
<td>1</td>
<td>6.02e-02</td>
<td>1.32</td>
</tr>
<tr>
<td>ThetaGPU</td>
<td>1</td>
<td>40GB A100 GPU</td>
<td>CUDA</td>
<td>4</td>
<td>1</td>
<td>6.78e-02</td>
<td>1.57</td>
</tr>
<tr>
<td>Perlmutter</td>
<td>1</td>
<td>40GB A100 GPU</td>
<td>CUDA</td>
<td>4</td>
<td>1</td>
<td>4.16e-02</td>
<td>1.62</td>
</tr>
<tr>
<td>Polaris</td>
<td>1</td>
<td>40GB A100 GPU</td>
<td>CUDA</td>
<td>4</td>
<td>1</td>
<td>4.31e-02</td>
<td>1.62</td>
</tr>
</tbody>
</table>

supports multiple versions of all of its high-intensity kernels and communication utilities. These are determined at run time by running a small suite of tests for each invocation of the given utility. Thus, the performance is optimized under the loading conditions of that particular kernel for the particular platform for the particular application. An example of these outputs, along with the kernel-by-kernel breakdown, is presented in Section 5.

2 NekRS Performance on a Single GPU

Table 1 demonstrates NekRS performance for ExaSMR’s singlerod simulation on a single GPU. Simulations are performed for 500 steps and the average time-per-step, \(t_{step}\), is measured in seconds for the last 400 steps. For a given system, the speedup is the inverse ratio of \(t_{step}\) to that of Summit. \(v_i\) and \(p_i\) represent the average iteration counts per step of the velocity components and pressure. Timestepping is based Nek5000’s second-order characteristics method with one substep \([11, 12]\) and the timestep size is \(\Delta t = 1.2e-3\) (CFL=1.82). Pressure preconditioning is based on \(p\)-multigrid with CHEBYSHEV+ASM smoothing and hypre AMG for coarse grid solve \([9]\). Tolerances for pressure and velocity are 1e-4 and 1e-6, respectively. We note that this test case has been explored in the context of NekRS kernel and algorithm development on other architectures in earlier work, including \([13, 14, 15]\).

The single-device results of Table 1 show that, for the current version of NekRS, a single GCD of the MI250X on Crusher realizes a 1.32\(\times\) gain in Navier-Stokes solution performance over a single V100 on Summit. Similarly, the A100s are realizing \(\approx 1.6\)-fold gain over the V100.

3 NekRS Performance on Frontier vs. Crusher

In collaboration with OLCF, the Nek team performed scaling studies on Frontier using NekRS version 22.0. Simulations on Frontier were run by John Holmen at OLCF while those on Crusher were run by the Nek team. On Frontier, rocm/5.1.0 and cray-mpich/8.1.17 were used. On Crusher, simulations were performed with variation of versions such as rocm/5.1.0, rocm/5.2.0, cray-mpich/8.1.16 and cray-mpich/8.1.19. On Crusher, rocm/5.1.0 is 2%–5% faster than rocm/5.2.0. We observe that the performance on Frontier is better than that on Crusher.

We consider ExaSMR’s 17\(\times\)17 rod-bundle geometry and extend the domain in streamwise direction with 10, 17, and 170 layers, keeping the mesh density same, which correspond to 277
Table 2: Problem setup for strong/weak scaling studies.

<table>
<thead>
<tr>
<th>Strong Scaling Test Sets</th>
<th>E</th>
<th>n</th>
<th>rank, P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>277</td>
<td>95M</td>
<td>8–64</td>
</tr>
<tr>
<td>Case 2</td>
<td>470</td>
<td>161M</td>
<td>14–128</td>
</tr>
<tr>
<td>Case 3</td>
<td>470</td>
<td>1.6B</td>
<td>128–16320</td>
</tr>
</tbody>
</table>

Figure 2: Strong-scaling on Frontier and Crusher for $17 \times 17$ rod bundles with 10, 17 and 170 layers with total number of grid points of $n = 95M$, $161M$, $1.6B$. Average time-per-step vs. rank, $P$ (left) and average time-per-step vs. $n/P$ (right). Frontier is set with (cray-mpich/8.1.17, rocm/5.1.0) and Crusher with (cray-mpich/8.1.19, rocm/5.2.0).

Figure 2 compares the scaling performance of Frontier to that of Crusher. Simulations are performed for 2000 steps and the average time-per-step, $t_{\text{step}}$, is measured in seconds for the last 1000 steps. The third-order backward-difference formula (BDF3) combined with the third-order extrapolation (EXT3) [16] is used for timestepping and the timestep size is $\Delta t = 3.0e-04$ (CFL=0.82).

Figure 2, left, shows the classic strong scaling for the problem sizes of $n = 95M$, $161M$, and $1.6B$, demonstrating the average time-per-step vs. the number of MPI ranks, $P$. We run a single MPI rank per GCD and there are 8 GCDs per node. The dashed lines in skyblue represent ideal strong-scale profiles for each case. The solid lines in red are for Frontier and the solid lines in black are for Crusher. We observe that Frontier is consistently slightly faster than Crusher for these three problem sizes. For larger problem sizes and processor counts, the Frontier advantage is increased.

Figure 2, right, shows the average time-per-step vs. the number of points per MPI rank, $n/P$, where $n$ is the total number of grid points. $t_{\text{step}}$ based on $n/P$ is quite independent of the problem
size, \( n \). This is the metric illustrating that the strong-scaling performance is primarily a function of \( n/P \) and only weakly dependent on \( n \) or \( P \) individually, which is in accord with the extensive studies presented in [7]. Based on this metric, we can determine a reasonable number value of \( n/P \) for a given parallel efficiency and, from there, determine the number of MPI ranks required for a problem of size \( n \) to meet that expected efficiency. We provide more detailed performance behaviors depending on problem sizes in Figures 7–9.

Figure 3, left and right, shows performance for a 17×17 rod bundle with 170 layers (\( n=1.6B \)). Here we extend our discussion to other NVIDIA-based GPU architectures such as Summit (V100) at OLCF, Perlmutter (A100) at NERSC, and Polaris (A100) at ALCF, and compare those to Frontier and Crusher. While we observe that Frontier is faster than Crusher in Figure 2, we see that Crusher is faster than Summit, but not quite as fast as the A100-based Perlmutter (NERSC) and Polaris (ALCF) platforms. We provide more detailed performance behavior as a function of \( P \) in Figures 7-9. Anomalous behavior for several of these architectures are also discussed in Section 5.

Finally, returning to the results for Frontier vs. Crusher, Figures 4-6 show detailed information for each problem size, including timings, parallel efficiency, DOFS, and runtime statistics for the advection kernel (\texttt{makef}) and for the coarse-grid-solve for the pressure Poisson problem. The bottom left plots in Figures 4-6 show that Frontier and Crusher deliver the same performance on the compute-intensive \texttt{makef} kernel, which evaluates the advection term for the Navier-Stokes equations. By contrast, the bottom right plots in Figures 4-6 show Crusher is a bit slower for the communication-intensive coarse-grid solve, which is part of the multigrid preconditioner for the pressure-Poisson problem. The solve is performed on the host CPUs using algebraic multigrid with \texttt{Hypre}. (These lower plots show the times for the full 2000 steps, not the time-per-step.)

For smaller problem sizes with \( n=95M \) and 161M, the second-row plots on the left and right in Figures 4–5 reveal that the 80% parallel efficiency is achieved with \( n/P=2.2M \) on Frontier and \( n/P=2.5M \) on Crusher. The same plots in Figure 6 demonstrate that Frontier realizes 80%
Figure 4: Strong-scaling on Frontier vs. Crusher for 17×17 rod bundle with 10 layers.
Figure 5: Strong-scaling on Frontier vs. Crusher for $17 \times 17$ rod bundle with 17 layers.
Figure 6: Strong-scaling on Frontier vs. Crusher for 17×17 rod bundle with 170 layers.
parallel efficiency at $n/P \approx 3M$ points per rank while Crusher at $n/P \approx 5$ million points per rank, which would normally be viewed as the strong-scale limit for NekRS simulations on these platforms. This somewhat disappointing increase in $n_0$ implies that NekRS is not weak-scaling well on Frontier/Crusher. As noted in [6], weak-scaling performance is generally affected by the coarse-grid solve, which is one of the few terms for which the cost grows with $P$ (in this case, as $O(\log P)$), instead scaling as $n/P$. Indeed, inspection of the lower right graph in Figures 5–6 shows that, for the same $n/P = 1.6M$, the total coarse-grid solve time (for 2000 steps) is 1s for $P = 1000$ and only 0.65s for $P = 100$, which corresponds precisely to the ratio $\log_2(1000)/\log_2(100)$. We note that Crusher suffers even more in the (host-driven) coarse-grid solve. By contrast, Crusher and Frontier have identical behavior on the communication-minimal and compute intensive nonlinear advection evaluation, as seen in the lower left graph in Figure 6.

The third-row left and right plots in Figures 4–6 show GDOF per step per $t_{\text{step}}$ per rank vs. time-per-step and GDOF per step per $t_{\text{step}}$ per rank vs. $n/P$, where GDOF is defined by the total number of degree of freedom, which is $4n$ (three velocity vectors and one pressure field). Frontier consistently shows faster performance than Crusher, particularly for the larger problem sizes (and hence, higher processor counts), which we can primarily attribute to the speed of the coarse grid solve on Crusher.

Figure 6 in particular shows the performance as $n/P$ gets as low as 0.1M, at which point the performance curve flattens and the device does not further speedup. There are two reasons for stalled performance: communication overhead and insufficient work on the device (even in the absence of communication), as shown in an earlier study for NekCEM on ORNL’s Titan, which feature NVIDIA K20X GPUs [17] and in the single-GPU studies on the NVIDIA V100 in the CEED benchmark paper [7].

4 NekRS performance on Summit, ThetaGPU, Perlmutter, Polaris, Crusher and Frontier

In this section, we continue with scaling studies on the 17×17 rod bundle simulations for the NVIDIA-based GPU platforms, Summit (V100), ThetaGPU (A100), Perlmutter (A100) and Polaris (A100), compared to the AMD MI-250X platforms, Frontier and Crusher. Recall that, in Figure 3, we showed the 170-layer case across the platforms of Frontier, Crusher, Summit, Perlmutter and Polaris. Here, we discuss the performance in more detail in Figures 7–9. In Figure 8, we include performance on ThetaGPU. We run one MPI rank per V100 or A100 on the NVIDIA-based nodes and one MPI rank per GCD on the AMD MI250X nodes.

Figures 7–9 show the same metrics as for the Crusher-Frontier plots of Figures 4–6. It is important to point out that these strong-scaling plots start from a high level of performance. NekRS currently leverages extensive tuning of several key FP64 and FP32 kernels in libParanumal, including the standard spectral element Laplacian matrix-vector product, local tensor-product solves using fast diagonalization, and dealiased evaluation of the advection operator on a finer set of quadrature points. These kernels are sustaining up to 3 TFLOPS FP64 and 5–8 TFLOPS FP32, per GPU or GCD. At the strong-scale limit, with MPI overhead, NekRS is sustaining $\approx$ 1 TFLOPS per rank (i.e., per A100 or GCD) for the full Navier-Stokes solution.

Vector reductions also exhibit $O(\log P)$ growth, but these can be mitigated by hardware support [6].
Figure 7: Strong-scaling on various GPU architectures for $17 \times 17$ rod bundle with 10 layers.
Figure 8: Strong-scaling on various GPU architectures for $17 \times 17$ rod bundle with 17 layers.
Figure 9: Strong-scaling on various GPU architectures for $17 \times 17$ rod bundle with 170 layers.
An important figure of merit is $n_{0.8}$, which is the value of $n/P$ at which the simulation realizes 80% parallel efficiency. From the second row, right, we see that $n_{0.8} = 2.5M$ for Perlmutter/Crusher and 2M for Polaris (GPUDirect in green dashed lines) For Polaris without GPUDirect (magenta solid line) we find $n_{0.8} = 4.5M$. The plot on row 3, left, indicates that a remarkably small $t_{step}$ value of 0.015 seconds per step is realizable on Polaris, albeit at 25% efficiency.

The plots on the last row, left, of Figures 7–9 show that the time in the advection update strong-scales quite well, as would be expected. The curves for the single GCD and A100 collapse to nearly the same performance while the older V100 technology of Summit is about 1.5× slower. In the absence of communication, this kernel is sustaining 3–4 TFLOPS FP64 on these newer architectures, although the graphs here do include the communication overhead. By contrast, the last row, right, shows the performance for the communication-intensive coarse grid solve, which is performed using Hypre on the host CPUs. Here, both Crusher and Summit show relatively poor performance at small values of $n/P$ or large values of $P$. Also, in Figure 7 lower right we see that Polaris without GPUDirect exhibits some level of system noise.

We also observe in Figure 8 that ThetaGPU (dashed blue lines) is about 1.3–1.5× slower than Perlmutter and Polaris (GPUDirect). From the lower left curve, we see that the ThetaGPU performance is lower than the other A100 platforms even for this work-intensive operation. Moreover, in terms of parallel efficiency, it falls off faster than Polaris whether Polaris is or is not using GPUDirect communication.

5 Discussion

In this section we discuss a variety of “anomalous” behaviors encountered in these studies. By anomalous we mean that they are adverse behaviors that either appear or disappear with software and hardware updates. One could argue that these are passing phenomena not worthy of reporting. However, users are actively using these machines and it’s important that everyone understand potential pitfalls in system performance that might directly impact their own timing studies or production runtimes.

The behaviors described here include the performance of the large-memory (32GB vs 16GB) nodes on Summit, the use of a nonmultiple of 8 ranks on Crusher, the influence of GPU-direct communication on Polaris, and the upgrade from Slingshot 10 to Slingshot 11 on Perlmutter.

5.1 Performance on Summit V100 16GB vs. 32GB

Most of the 4608 nodes on Summit have 16GB of device memory, which limits how small one can take $P_0$ in the efficiency definition (2). There are, however, a few nodes that have 32GB, which allows one to fit more points onto each V100. Unfortunately, as seen in Figures 7–9, the Summit 32GB curves are seen to perform about 10% slower than their 16GB counterparts. The last row of graphs in Figure 8 are particularly revealing: One can see from the makef plot that the V100s perform at the same rate for both the 16GB and 32GB nodes, but that the host-based coarse grid solve costs differ by almost 1.5×, which indicates either an excessive device-host communication overhead or some type of inter-host communication slow down.
5.2 Performance on Crusher with Rank Dependency

In our timing studies, we typically do not require to fully occupy each node. This happens, for example, if one wants to make a device-to-device comparison between Summit, with six V100 per node, and Crusher, which has 8 GCDs per node. Our initial plots of Crusher timing data appeared to be very erratic in its dependence on $P$. Closer inspection, however, revealed that the performance for $\text{mod}(P,8) \neq 0$ was about $2 \times$ slower than for the case of $\text{mod}(P,8) = 0$. Figures 10–12 show early timings taken on Crusher with the results sorted onto two curves corresponding to multiples and nonmultiples of 8. Unlike the Summit behavior of the preceding section, this is clearly a device issue and not a host issue, as can be seen in the last row in each of Figures 10–12. This anomaly has been resolved and has not appeared as an issue on Frontier.

5.3 Performance on Polaris with GPUDirect vs. without GPUDirect

When we started the ALCF Polaris Hackathon in late July, 2022, Polaris did not support GPUDirect. However, the ALCF team managed to get it in place by midweek and we were able to get timings with and without GPUDirect. The results are shown in Figures 10–12.

As expected, communication overhead is more significant without GPUDirect and consequently the no GPUDirect curves on Polaris show relatively poor performance as the amount of local work, $n/P$, is reduced. For example, in row 2, right, of Figure 11 the $n_{0.8}$ for Polaris with GPUDirect is 2.5M, whereas it is 4.5M without GPUDirect. In the last row of the same figure we see that neither makef nor the coarse-grid solve are influence by the presence or absence of GPUDirect. For makef, communication does not matter. The coarse-grid solve is communication intensive, but all of that originates from host. We can also see in the results of Figure 12 that the no GPUDirect results are relatively noisy.

We remark that the number of unknowns in the coarse grid on each rank is roughly equal to the number of spectral elements on that rank. For $p = 7$ and $n/P = 2M$, we have $E = 2M/(343) \approx 5800$ elements per rank, so the local coarse-grid problem size is 5800. If we were running on all $P = 27648$ V100s of Summit, the total coarse grid size would be $P \times 5800 = 160M$.

5.4 Performance on Perlmutter with Slingshot 10 vs. Slingshot 11

Finally, one other discovery during the Polaris workshop was a sudden change in the behavior of Perlmutter at NERSC. Polaris and Perlmutter have very similar architectures, so it was expected that they would have similar performance as is indeed evident in, for example, in Figure 8. During this timeframe, however, the Perlmutter interconnect was being upgraded from Slingshot 10 (SS10) to Slingshot 11 (SS11). The performance started to vary radically with $P$, albeit in a highly repeatable fashion.

The strong-scaling plot in Figure 13, top-left, illustrates the problem. For 48 ranks, the Navier-Stokes runtime with SS11 jumps by a factor of 3 over that with SS10 and over Polaris, which is still running SS10. These are repeatable results evidenced by several timings over a period of several weeks. Of course, the last row in the figure reveals that this anomaly is neither a GPU issue, since makef timings are essentially identical, nor a host-to-host communication issue, since the coarse-grid times are close to the same.
Figure 10: Strong-scaling on Crusher for 17×17 rod bundle with 10 layers.
Figure 11: Strong-scaling on Crusher for $17 \times 17$ rod bundle with 17 layers.
Figure 12: Strong-scaling on Crusher for $17 \times 17$ rod bundle with 170 layers.
Figure 13: Strong-scaling on Perlmutter with Slingshot 10 and 11.
Inspection of the NekRS logfiles shows that the increase is all focused in one section of the code, namely the non-local Schwarz-based smoother in the $p$-multigrid preconditioner for the pressure Poisson problem. That section of code is running $10 \times$ slower than its SS10 counterpart! Below, we show the logfile content for the two simulations with $P = 48$ (which is the slowest case).

SS10 logfile:

<table>
<thead>
<tr>
<th>name</th>
<th>time</th>
<th>% calls</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup</td>
<td>3.82904e+01s</td>
<td>0.38</td>
<td>1</td>
</tr>
<tr>
<td>loadKernels</td>
<td>1.03634e+01s</td>
<td>0.27</td>
<td>1</td>
</tr>
<tr>
<td>udfExecuteStep</td>
<td>4.79398e-03s</td>
<td>0.00</td>
<td>2001</td>
</tr>
<tr>
<td>elapsedStepSum</td>
<td>6.13724e+01s</td>
<td>0.62</td>
<td></td>
</tr>
<tr>
<td>solve</td>
<td>6.12031e+01s</td>
<td>0.61</td>
<td>9880</td>
</tr>
<tr>
<td>min</td>
<td>2.31879e-02s</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>5.1687e-02s</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>flop/s</td>
<td>3.36729e+13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SS11 logfile:

<table>
<thead>
<tr>
<th>name</th>
<th>time</th>
<th>% calls</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup</td>
<td>3.98696e+01s</td>
<td>0.16</td>
<td>1</td>
</tr>
<tr>
<td>loadKernels</td>
<td>8.86541e+00s</td>
<td>0.22</td>
<td>1</td>
</tr>
<tr>
<td>udfExecuteStep</td>
<td>4.79946e-03s</td>
<td>0.00</td>
<td>2001</td>
</tr>
<tr>
<td>elapsedStepSum</td>
<td>2.06042e+02s</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>solve</td>
<td>2.05867e+02s</td>
<td>0.84</td>
<td></td>
</tr>
<tr>
<td>min</td>
<td>5.50540e-02s</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>max</td>
<td>3.32500e-01s</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>flop/s</td>
<td>1.00108e+13</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We also note that SS11 delivers higher bandwidth than SS10 in all of the set-up tests done by NekRS when it is making a runtime selection of the best communication algorithm for each subproblem. We list the logfile output for those below.

SS10 logfile:

<table>
<thead>
<tr>
<th>name</th>
<th>time</th>
<th>% calls</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup</td>
<td>5.59237e+00s</td>
<td>0.09</td>
<td>2000</td>
</tr>
<tr>
<td>loadKernels</td>
<td>3.99699e-02s</td>
<td>0.01</td>
<td>2000</td>
</tr>
<tr>
<td>udfProperties</td>
<td>4.82886e-03s</td>
<td>0.00</td>
<td>2001</td>
</tr>
<tr>
<td>velocitySolve</td>
<td>1.73346e+01s</td>
<td>0.28</td>
<td>2000</td>
</tr>
<tr>
<td>rhs</td>
<td>2.93962e+00s</td>
<td>0.13</td>
<td>2470</td>
</tr>
<tr>
<td>pressureSolve</td>
<td>4.20520e+01s</td>
<td>0.56</td>
<td>2470</td>
</tr>
<tr>
<td>rhs</td>
<td>4.69203e+00s</td>
<td>0.14</td>
<td>2470</td>
</tr>
<tr>
<td>preconditioner</td>
<td>2.26178e+01s</td>
<td>0.66</td>
<td>2470</td>
</tr>
<tr>
<td>pMG smoother</td>
<td>1.51609e+01s</td>
<td>0.67</td>
<td>2470</td>
</tr>
<tr>
<td>coarse grid</td>
<td>5.35568e+00s</td>
<td>0.24</td>
<td>2470</td>
</tr>
<tr>
<td>initial guess</td>
<td>3.18958e+00s</td>
<td>0.09</td>
<td>2000</td>
</tr>
</tbody>
</table>

SS11 logfile:

<table>
<thead>
<tr>
<th>name</th>
<th>time</th>
<th>% calls</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup</td>
<td>5.57576e+00s</td>
<td>0.03</td>
<td>2000</td>
</tr>
<tr>
<td>loadKernels</td>
<td>3.99699e-02s</td>
<td>0.01</td>
<td>2000</td>
</tr>
<tr>
<td>udfProperties</td>
<td>4.82886e-03s</td>
<td>0.00</td>
<td>2001</td>
</tr>
<tr>
<td>velocitySolve</td>
<td>1.73346e+01s</td>
<td>0.28</td>
<td>2000</td>
</tr>
<tr>
<td>rhs</td>
<td>2.93962e+00s</td>
<td>0.13</td>
<td>2470</td>
</tr>
<tr>
<td>pressureSolve</td>
<td>4.20520e+01s</td>
<td>0.56</td>
<td>2470</td>
</tr>
<tr>
<td>rhs</td>
<td>4.69203e+00s</td>
<td>0.14</td>
<td>2470</td>
</tr>
<tr>
<td>preconditioner</td>
<td>1.67813e+02s</td>
<td>0.94</td>
<td>2470</td>
</tr>
<tr>
<td>pMG smoother</td>
<td>1.49446e+02s</td>
<td>0.89</td>
<td>2470</td>
</tr>
<tr>
<td>coarse grid</td>
<td>5.53950e+00s</td>
<td>0.03</td>
<td>2470</td>
</tr>
<tr>
<td>initial guess</td>
<td>3.20173e+00s</td>
<td>0.02</td>
<td>2000</td>
</tr>
</tbody>
</table>

For completeness, we also include the kernel performance numbers as reported in the NekRS logfiles:
In the table above, kv reflects the particular kernel version chosen out of the suite of available kernels in NekRS for the particular operation. We see that the 64-bit Ax kernels (the matrix-vector product with the Laplace operator for spectral element order \( N \)) realize \( \approx 2 \) TFLOPS per device, while their 32-bit counterparts realize 3–4 TFLOPS. (32-bit arithmetic is used in the preconditioner only.) The fast-diagonalization method (fdm) implements local tensor-product solves of the form \( \mathbf{z} = \mathbf{S} \mathbf{\Lambda}^{-1} \mathbf{S}^T \mathbf{r} \), where \( \mathbf{S} = (\mathbf{S}_t \otimes \mathbf{S}_s \otimes \mathbf{S}_r) \) is the orthogonal matrix of eigenvectors for the overlapped Poisson problem in local \((r-s-t)\) coordinates in the reference element, \( \hat{\Omega} = [-1,1]^3 \) [16]. This is a fast operation and can be seen to sustain \( > 7 \) GFLOPS (FP32). Note that, as expected, the kernel performance, which does not include any MPI overhead, is not dependent on the Slingshot version. NekRS also reports the total observed GFLOPS, which is seen to by 701 GFLOPS/rank for SS10 and 208 GFLOPS/rank for SS11. We note that, for this particular case, \( P = 48 \) and \( n = 96790120 \), for which \( n/P = 2.01M \), which is close to (but below) the value of \( n_{0.8} \). We can anticipate that the saturated floating-point performance for SS10 would thus be about 701/0.8 = 876 GFLOPS.

At present we do not know why the \( p \)-multigrid smoother time is an order-of-magnitude larger on SS11 than SS10. That question is under investigation but has been hampered by downtime of Perlmutter.

## 6 Conclusions

This study explores the performance of a highly-tuned incompressible flow code, NekRS, on current-generation HPC architectures featuring accelerator-based nodes. The principal accelerators under consideration are the NVIDIA V100, NVIDIA A100, and MI250X with eight GCDs per node. We found that the raw performance of a single GCD is about 85% of a single A100. We also found that the AMD-based Crusher platform had slower host-based communication than its Polaris/Perlmutter counterpart, as witnessed by the relatively poor performance of the Hypre-based coarse-grid solver, which runs on the host. This situation is significantly improved on Frontier.

What is critical to end-user performance is the ratio \( n_{0.8}/P \), which governs the time-to-solution (5). Despite the pessimistic results in [7], where it appeared that GPU-based platforms might be 3\times slower than ANL’s IBM BG/Q, Mira, we are finding that NekRS is in 3\times faster than Nek5000 on Mira. This gain can be attributed to careful attention to reducing \( n_{0.8} \), to improved pressure preconditioners tuned specifically for accelerator-based nodes, to the use of 32-bit precision in the preconditioners, and to extensive use of overlapped communication and computation.
Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357 and by the Exascale Computing Project (17-SC-20-SC). The research used resources at 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.

References


