VASP Performance on HPE Cray EX Based on NVIDIA A100 GPUs and AMD Milan CPUs

1st Zhengji Zhao
Lawrence Berkeley National Laboratory
Berkeley, USA
zzhao@lbl.gov

2nd Brian Austin
Lawrence Berkeley National Laboratory
Berkeley, USA
baustin@lbl.gov

3rd Stefan Maintz
NVIDIA
Zurich, Switzerland
smaintz@nvidia.com

4th Martijn Marsman
VASP Software GmbH and University of Vienna
Vienna, Austria
martijn.marsman@vasp.at

***Abstract*—NERSC's new supercomputer, Perlmutter, an HPE Cray EX system, has recently entered production. NERSC users are transitioning from a Cray XC40 system based on Intel Haswell and KNL processors to Perlmutter with NVIDIA A100 GPUs and AMD Milan CPUs with more on-node parallelism and NUMA domains. VASP, a widely-used materials science code that uses about 20% of NERSC's computing cycles, has been ported to GPUs using OpenACC. For applications to achieve optimal performance, features specific to Cray EX must be explored, including the build and runtime options. In this paper, we present a performance analysis of representative VASP workloads on Perlmutter, addressing practical questions concerning hundreds of VASP users: What types of VASP workloads are suitable to run on GPUs? What is the optimal number of GPU nodes to use for a given problem size? How many MPI processes should share a GPU? What Slingshot options improve VASP performance? Is it worthwhile to enable OpenMP threads when running on GPU nodes? How many threads per task perform best on Milan CPU nodes? What are the most effective ways to minimize charging and energy costs when running VASP jobs on Perlmutter? This paper will serve as a Cray EX performance guide for VASP users and others.**

***Index Terms*—VASP, OpenACC, NVIDIA A100, OpenMP + MPI, NCCL, AMD Milan, Slingshot, Performance**

I. Introduction

NERSC's new supercomputer, Perlmutter [1], an HPE Cray EX system, features NVIDIA A100 GPUs, AMD Milan CPUs, Slingshot interconnect, and an all-flash Lustre file system. Perlmutter has recently entered production. NERSC users are transitioning from a Cray XC40 system, Cori [2] with Intel Haswell and KNL processors, to Perlmutter, based on NVIDIA A100 GPUs and AMD Milan CPUs with more on-node parallelism and NUMA domains. VASP [3], a widely-used materials science code that consumes about 20% of computing cycles at NERSC (See Figure 1), has been ported to GPUs using OpenACC [4] [5] [6]. The OpenACC port of VASP has been highly optimized for NVIDIA GPUs and is capable of solving science problems that have not been possible on Cori (See Figure 2 for such an example), achieving more than 30x speedup over Cori for a single node performance (See Figure 3). For applications to achieve optimal performance, however, features specific to Cray EX must be explored, including the build and runtime options.

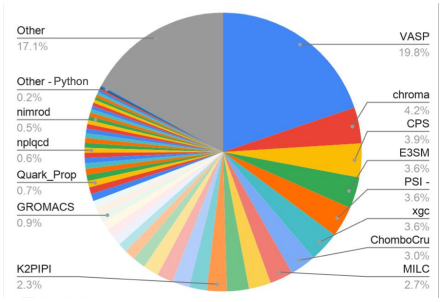


Fig. 1. NERSC machine time breakdown by applications in the allocation year 2018. VASP has been the top #1 code, spending about 20% of computing cycles each allocation year at NERSC.

In this paper, we present a performance analysis of representative VASP workloads on Perlmutter similar to [7], attempting to address practical questions concerning hundreds of VASP users on Perlmutter. We explored compiler and library options for VASP and studied the parallel scaling of VASP on both A100 GPU and Milan CPU nodes using a variety of VASP workloads, including its interaction with the internal VASP run parameters (e.g., use of NCCL). We also investigated how many MPI processes should share a GPU and if NVIDIA's Multi-Process Service (MPS) [8] can be used to accelerate further VASP workloads on GPUs. Furthermore, we investigated the the performance effect of SMT (Simultaneous Multi-Threading) and OpenMP threads on Milan CPUs, which offer an increased on-node parallelism and NUMA domains in comparison to Cori KNL. In addition, we investigated the GPU performance speedup over Milan CPUs for different problem sizes and computation types, and analyzed power usage of VASP jobs to understand what are the best ways to minimize machine time charging and energy

This work was supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231.

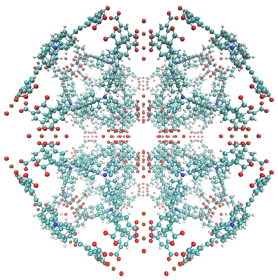


Fig. 2. This figure shows a large MOF structure that was sent to the Materials Project [9] by a user. It was too large for the Materials Project team to handle through their normal workflows on Cori. MOFs are composed of two major components: a metal ion or cluster of metal ions and an organic molecule called a linker. The system contained 3584 atoms and 13408 electrons. Here are the additional computational details: Functional: DFT; Algo: CG (BD+RMM); NBANDS= 9600; FFT grids: 378x378x378; 756x756x756; NPLWV: 54,010,152; KPOINTS: 1 1 1. The job completed 99 ionic steps in 32.8 hours using 32 nodes (128 GPUs) on Perlmutter.

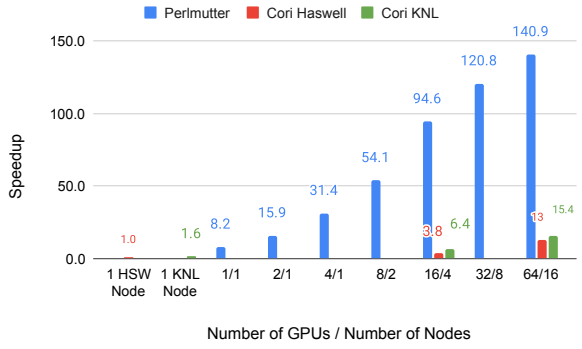


Fig. 3. VASP performance on Perlmutter GPUs with benchmark Si256_hse. This benchmark performs an HSE hybrid functional calculation on a 256-atom silicon supercell with a vacancy, one of the most commonly used workloads in VASP. The horizontal axis shows the number of GPUs, the number of nodes used on Perlmutter, and the number of nodes used for the reference runs on Cori Haswell and KNL. The vertical axis shows the speedup over a single Haswell node. The blue bars show the speedup of Perlmutter, and the red and green bars show the Haswell and KNL results, respectively. (Note that the horizontal axis shows both GPU and node counts, but only node counts apply to the Cori and KNL results.)

costs when running VASP jobs on Perlmutter. We selected seven benchmarks based on the recent VASP user survey at NERSC to cover the representative VASP workloads and to exercise different code paths.

This paper will serve as a Cray EX performance guide for VASP users and others. Additionally, the performance analysis presented in this paper provides feedback and input for computer vendors and others about the architecture's performance in a real-world scientific application with a large user base.

The rest of the paper is organized as follows: after the introduction, we will describe the experimental setup, and then present the performance results and analysis for both Perlmutter GPUs and CPUs, and discuss how to achieve charging and energy efficiency when running VASP jobs, and then conclude the paper with a summary.

II. System Configuration and Environment Setup

A. Perlmutter System Configuration

Perlmutter, based on the HPE Cray Shasta platform, is a heterogeneous system comprising both GPU-accelerated and CPU-only nodes. It consists of 1792 GPU accelerated nodes with one AMD EPYC 7763 processor (codename: Milan) and four NVIDIA A100 GPUs and 3072 CPU-only nodes with two AMD EPYC 7763 processors, interconnected with HPE Slingshot network. Among 1792 GPU accelerated nodes, 256 nodes have 80 GB HBM with a bandwidth of 2,039 GB/s per GPU, and the rest have 40 GB HBM with a bandwidth of 1,555 GB/s per GPU. Each Milan CPU processor has 64 cores (128 hardware threads) running at 2.45 GHz of base clock rate (maximum boost clock: up to 3.5 GHz) and 256 GB DDR4 memory with a bandwidth of 204.8 GB/s, thus there are 128 cores (256 hardware threads) and 512 GB memory per Milan CPU node on Perlmutter. Each Perlmutter CPU node has eight NUMA domains. In contrast, there are two NUMA domains for a Cori Haswell node and one for a KNL node. The Figure 4 illustrates Perlmutter GPU and CPU nodes. Each GPU-accelerated compute node is connected to four HPE Cray's proprietary Cassini NICs, while each CPU-only node is connected to one NIC. More information about the Perlmutter architecture is available at [1].

Perlmutter runs HPE Cray OS (SLES15SP4) version 2.4 and Slurm 22.05.8.

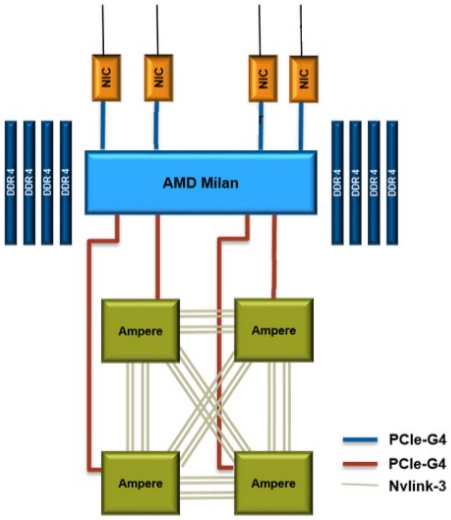
B. VASP

The Vienna Ab initio Simulation Package (VASP) [3] is a widely used materials science code that is highly ranked at NERSC and other supercomputing centers worldwide. The fundamental mathematical problem that VASP solves is a non-linear eigenvalue problem that has to be solved iteratively via self-consistent iteration cycles until a desired accuracy is achieved. The main task of VASP is to solve the following N eigenvalue equations

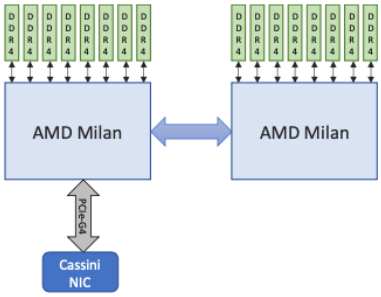
$$[-\tfrac{1}{2}\nabla^2 + V(\mathbf{r})]\Psi_i(\mathbf{r}) = \epsilon_i\Psi_i(\mathbf{r}), i = 1, 2, ..., N$$

for the "one-electron" orbitals, $\Psi(\mathbf{r})$, expanded in a plane wave basis, and energies, ϵ_i. N is referred to as the number of orbitals or bands. VASP implements efficient iterative matrix diagonalization techniques (as referenced in Table I, row Algo) and rich features that utilize various levels of approximations (as referenced in Table I, row Functional).

VASP is written mainly in Fortran 90 and heavily utilizes FFTs and linear algebra libraries. VASP is parallelized with MPI and OpenMP for multi-/many-core CPUs [10] [11] and MPI and OpenACC for GPUs [4] [5] [6]. VASP implements multiple levels of parallelism, and by default, it distributes orbitals (work and data) over MPI tasks. The GPU communications are optimized with NVIDIA's Collective Communications Library (NCCL) [12] alternatives to MPI.



(a) Perlmutter GPU Node



(b) Perlmutter CPU Node

Fig. 4. Perlmutter compute nodes. The upper panel illustrates a Perlmutter GPU node, and the lower one shows a Perlmutter CPU node.

The OpenACC port ensures that costly data transfers between the CPU and GPU memories are minimized by moving all important data structures to the GPUs as early as possible and keeping them there. Therefore, in addition to offloading the hot spots with libraries like cuFFT, cuBLAS, and cuSOLVER, all surrounding custom kernels had to be accelerated using OpenACC directives. Also, making all required communications GPU-centric using NCCL allows enqueueing communications in an OpenACC queue, just like a compute kernel, which helps avoid synchronization points. To address scaling issues due to Amdahl's law, the code sections surrounding the main challenge of optimizing the orbitals, such as updating the charge density and evaluating the density functionals, have also been brought to the GPUs. A few custom kernels benefit significantly from specialized implementations that are more GPU-friendly. For example, the CPU-optimized kernel calls a tiny GEMM in the inner loop, which would cause massive launch overheads on the GPU. A unified kernel offers heavily increased performance.

VASP has a single code base containing both the OpenACC port for GPUs and the OpenMP port for multi-/many-core CPUs. We used VASP 6.4.1 in our tests. We built the OpenMP port to run on Perlmutter's CPU-only nodes; and enabled both OpenACC and OpenMP for Perlmutter's GPU nodes. We selected NVIDIA's HPC SDK, which has the best OpenACC support, and used its optimized math libraries. Here is the software used in this study: NVIDIA HPC SDK 22.7 for NVIDIA compiler, CUDA 11.7, QD, cuBLAS, cuSOLVER, cuFFT libraries, and NCCL 2.15.5, Cray MPICH 8.1.25, MKL from Intel oneAPI 23.0.0 and its FFTW3 wrappers to FFT, and HDF5 1.12.2.

C. Benchmarks

In this study, we used seven test cases, denoted Si256_hse, B.hR105_hse, PdO4, PdO2, GaAsBi-64, CuC_vdw, and Si128_acfdtr. They were selected to represent the production workloads at NERSC and to exercise different code paths, including a variety of elements and problem sizes. For example, two HSE hybrid functional calculations with different atomic configurations and problem sizes are selected in the tests: Si256_hse is a 256-atom silicon supercell with a vacancy, and B.hR105_hse is a hexa-boron structure containing 105 atoms. The PdO4 and PdO2 are PdO slabs containing 348 and 174 atoms, respectively. They were selected to test the most commonly used code path, the DFT functional calculation using the RMM-DIIS iteration scheme. In addition, a ternary alloy structure, GaAsBi-64, was included to cover the metallic systems with the default iteration scheme, Block Davidson + RMM-DIIS algorithms. The CuC_vdw benchmark performs the Van Der Waals (VDW) functional calculation. Compared to the benchmarks used in the previous work [7], we added one more benchmark Si128_acfdtr, a 128-atom silicon supercell with a defect, to reflect the workload trend towards higher order methods based on the recent VASP user survey at NERSC.

The table I shows computational details about these benchmarks (See the VASP Wiki page [13] for more information about these tags).

D. Benchmark approach and runtime specifications

We have run various tests to answer the questions outlined in the abstract. Since the benchmarks are designed to run for a short time, in our tests, we disabled heavy I/O (LWAVE = .FALSE.). We started all runs from a given WAVECAR file so that different runs for the same benchmark always begin with the exact initial wavefunctions. We ran each test 5-10 times to avoid outliers and selected the best LOOP+ time, the dominant portion of the execution time in the production runs, where applicable. But for the ACFDTR tests, we used the elapsed time (total run time) reported by VASP. For the energy and power usage experiments, we modified the benchmarks slightly to run longer to mimic production runs (e.g., using a larger NELM than those reported in Table I).

VASP has three binaries; we used the standard vasp_std in the tests. In addition, we did a node-to-node performance

TABLE I
SEVEN VASP BENCHMARKS WERE CHOSEN TO COVER REPRESENTATIVE WORKLOADS AND TO EXERCISE DIFFERENT CODE PATHS.

	Si256_hse	B.hR105_hse	PdO4	PdO2	GaAsBi-64	CuC_vdw	Si128_acfdtr
Electrons (Ions)	1020 (255)	315 (105)	3288 (348)	1644 (174)	266 (64)	1064 (98)	512 (128)
Functional	HSE	HSE	DFT	DFT	DFT	VDW	ACFDT/RPA
Algo	CG (Damped)	CG (Damped)	RMM (VeryFast)	RMM (VeryFast)	BD+RMM (Fast)	RMM (VeryFast)	ACFDTR
NELM (NELMDL)	3 (0)	10 (5)	5 (3)	10 (4)	8 (0)	10 (5)	
NBANDS	640	256	2048	1024	192	640	23506 (NBANDSEXACT)
FFT grids	80x80x80 160x160x160	48x48x48 96x96x96	80x120x54 160x240x108	80x60x54 160x120x108	70x70x70 140x140x140	70x70x210 120x120x350	60x60x60 120x120x120
NPLWV	512000	110592	518400	259200	343000	1029000	216000
IRMAX	1579	1847	1445	1445	4177	3797	1340
IRDMAX	4998	2358	3515	3515	17249	50841	4249
LMDIM	18	8	18	18	18	18	6
KPOINTS (KPAR)	1 1 1 (1)	1 1 1 (1)	1 1 1 (1)	1 1 1 (1)	4 4 4 (2)	3 3 1 (1)	1 1 1 (1)

comparison when comparing VASP performance on the GPU and CPU nodes.

For most of the tests where possible, we used Slurm's CPU binding option (`-c` and `--cpu-bind`); but we didn't use its GPU binding (`--gpu-bind`) as it either slows down the code or causes the VASP jobs to hang. Instead, we used the following script, `gpu-bind.sh`, to set the `CUDA_VISIBLE_DEVICES` environment variable to a specific GPU for each process:

```
#!/bin/bash
export
    CUDA_VISIBLE_DEVICES=$((3-SLURM_LOCALID))
$*
```

This helps to address proper GPU affinities given that the NUMA nodes on the AMD Milan CPUs are numbered backward and rank 0, which gets bound by SLURM to NUMA node 0, has to use GPU 3.

The srun command is as follows for a single node run:

```
srun -n 4 -c32 --cpu-bind=cores gpu-bind.sh
    vasp_std
```

When running the hybrid OpenMP and MPI VASP on Perlmutter CPU nodes, we bind the threads to the CPUs using the OpenMP thread binding mechanism in addition to setting the process affinity:

```
export OMP_PROC_BIND=true
export OMP_PLACES=threads
```

However, when running VASP on GPUs, we didn't bind the threads to the CPUs as above because it slowed down the code significantly. This is explained by NCCL needing to spawn background threads that poll the communication status. NCCL inherits the core binding from its parent process, which will keep scheduling GPU kernels, so these two will compete for cycles on the same core, slowing down overall progress. An alternative to not binding the threads is to instruct NCCL to ignore the inherited CPU binding by using the following environment variable:

```
export NCCL_IGNORE_CPU_AFFINITY=1
```

Therefore, when running VASP on GPU nodes with OpenMP threads enabled (e.g., on a GPU node), the following affinity setting can be used:

```
export NCCL_IGNORE_CPU_AFFINITY=1
export OMP_NUM_THREADS=16 # or 8
export OMP_PROC_BIND=true
export OMP_PLACES=threads
srun -n 4 -c32 --cpu-bind=cores -G4
    ./gpu-bind.sh vasp_std
```

However, we did not see a visible difference when running VASP on GPU nodes with or without the OpenMP thread binding. While one must use the NCCL_IGNORE_CPU_AFFINITY=1 to avoid performance slowdown when binding the CPU processes and threads as above, the use of NCCL_IGNORE_CPU_AFFINITY=1 is always recommended to enable the best GPU performance for VASP.

III. VASP PERFORMANCE ON PERLMUTTER GPUS

In this section, we present the VASP performance results on Perlmutter GPU nodes and derive best practice tips based on the analysis of the results.

A. How many nodes/GPUs are optimal for VASP jobs running on Perlmutter GPU nodes?

Understanding the parallel scaling of applications is critical to run applications efficiently on HPC systems. While the optimal number of nodes and GPUs for a specific problem can be achieved only through case-by-case benchmarking, a parallel scaling study on the representative workloads could provide some guidance on how to select a good number of nodes and GPUs for a given problem.

Figure 5 shows the parallel efficiency of VASP on Perlmutter GPU nodes. The last benchmark, Si128_acfdtr, will be discussed separately in Section V. These benchmarks are not designed for heroic runs but to reflect users' day-to-day scientific runs. Si256_hse, PdO4, and CuC_vdw are relatively large

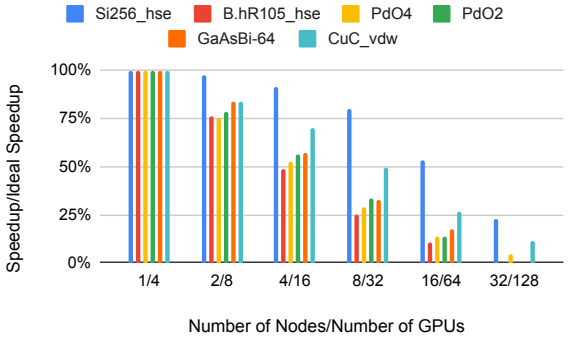


Fig. 5. Parallel efficiency of VASP on Perlmutter GPU nodes. The horizontal axis shows the number of nodes and GPUs, and the vertical axis shows the parallel efficiency of VASP (speedup/ideal parallel speedup). All tests ran with four MPI tasks per node (one task per GPU), each with one OpenMP thread.

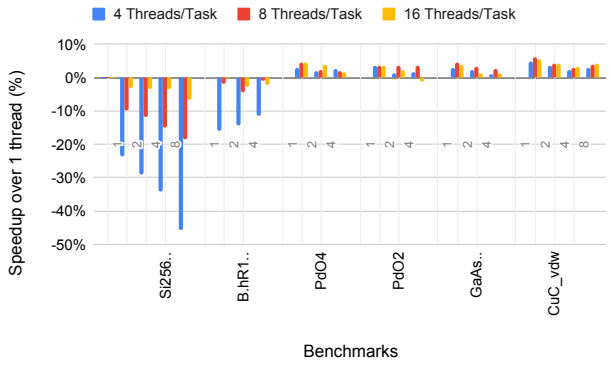


Fig. 6. OpenMP threading effect on VASP performance on Perlmutter GPU nodes. The horizontal axis shows the benchmarks, and the vertical axis shows the percentage speedup when using more than one thread per task, defined as [[Time(1-thread)-Time(multi-threads)]/Time(1-thread)]x100%. Thus, a positive number indicates a speedup and a negative number indicates a slowdown over one thread performance. For each benchmark, the results for several node counts are displayed.

among the six selected benchmarks. The remaining benchmarks, B.hR105_hse, PdO2, and GaAsBi-64, are relatively small, as indicated by the number of electrons and planewaves (NPLWV) in Table I. The parallel efficiency depends on the workloads and problem sizes. For smaller benchmarks, B.hR105_hse, PdO2, and GaAsBi-64, VASP hardly scales over two nodes, but for the larger benchmarks, Si256_hse, and CuC_vdw, VASP scales further out to more nodes. For example, Si256_hse (HSE workload) scales to eight GPU nodes with more than 75% of parallel efficiency. But PdO4 (DFT workloads), which is even more significant in size than Si256_hse, does not scale as well as Si256_hse, which performs a more expensive computation than PdO4.

Note that compared to running on CPUs, a significantly lower number of MPI processes are used for GPU runs. Therefore, increased MPI communication time is not the leading cause of the parallel efficiency drop when using more nodes. Instead, under-utilizing GPU resources is the leading cause, as there is simply insufficient work to saturate the increased GPU resources.

In practice, for the best interest of utilizing the compute resources efficiently, one may not run VASP at a parallel efficiency lower than 70%. Therefore, using one or two nodes would be sufficient for systems containing up to several hundred atoms (e.g., 300 atoms). For the HSE calculations, one may use more nodes; roughly distributing 100 bands per node would be a reasonable estimation.

B. Do additional OpenMP threads help VASP performance on GPUs?

Perlmutter GPU nodes have four NVIDIA A100 GPUs and 64 CPU cores (128 hardware threads) per node. If running four MPI tasks per node, which is the default for many applications running on Perlmutter, VASP would leave 60 CPU cores idle. Can the idling CPU resources be used to improve performance? The OpenACC port of VASP can run with OpenMP threads enabled. While most computation-intensive kernels are off-loaded to GPUs, enabling OpenMP threads may speed up the CPU portion of the code.

Figure 6 shows the OpenMP threading effect on VASP performance for GPU runs on Perlmutter. Since each benchmark can be run using a different number of nodes, the results running at several different node counts are displayed for each benchmark. One can see that enabling OpenMP threads significantly slows down the HSE workloads, especially when using fewer threads per task (see blue bars for Si256_hse and B.hR105_hse). However, OpenMP threads benefit other workloads, although not significantly, especially for one-node runs (up to 6%). In practice, enabling eight or 16 threads per task to utilize otherwise idling CPU resources for non-HSE workloads can get a small additional speedup.

The significant slowdown from using OpenMP threads for HSE workloads can be fixed when setting `NBLOCK_FOCK=NBANDS/Total_number_of_MPI_processes` in the INCAR file. `NBLOCK_FOCK`, the blocking factor in the Fock-exchange operator (an undocumented INCAR tag), is set to `2*OMP_NUM_THREADS` internally in VASP. Apparently, it benefits from a larger `NBLOCK_FOCK` than `2*OMP_NUM_THREADS`, especially for smaller `OMP_NUM_THREADS` values, to achieve optimal performance. When a proper `NBLOCK_FOCK` is used for HSE, the OpenMP threads also benefit the VASP performance similar to the rest of the workloads.

C. Does MPS help VASP performance?

NVIDIA's Multi-Process Service (MPS) allows multiple processes to run concurrently on the same GPU while avoiding the otherwise necessary plethora of context switches. MPS can increase performance when a single application process underutilizes the GPU resources.

Compared to CPU runs, VASP runs at a significantly reduced number of MPI processes per node on GPUs by default (4 vs. 128), running one MPI task per GPU. Does increasing the MPI parallelism per node by running multiple processes per GPU help VASP performance? In addition, VASP jobs

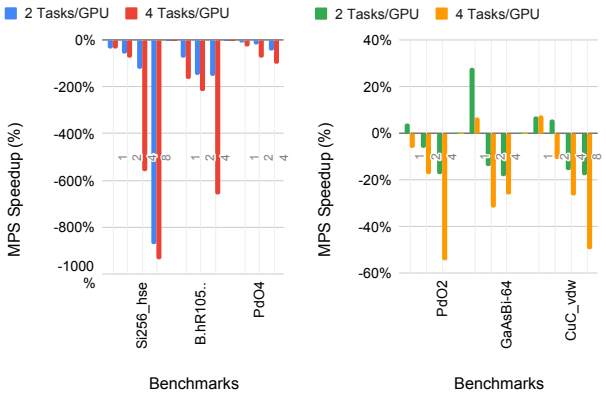


Fig. 7. MPS effect on VASP performance on Perlmutter GPU nodes. The horizontal axis shows the benchmarks, and the vertical axes, for both the left and right panels, show the percentage speedup when running multiple tasks per GPU via MPS over one task per GPU, defined as [[Time(1-task/GPU)-Time(multi-tasks/GPU via MPS)]/Time(1-task/GPU)]x100%. Thus, a positive number indicates a speedup and a negative number indicates a slowdown. The left vertical axis shows the results for benchmarks Si256_hse, B.hR105_hse, and PdO4; the right vertical axis shows the results for benchmarks PdO2, GaAsBi-64, and CuC_vdw. For each benchmark, the results for several different node counts are displayed.

must be large enough to get performance benefits on GPUs. Can MPS enable acceleration for additional VASP workloads on GPUs that are otherwise too small to run efficiently on GPUs?

Figure 7 shows the performance effect of MPS on VASP using the six selected benchmarks. The multi-task per GPU results via MPS are compared to the default run with NCCL enabled. As shown in the figure, MPS slows down VASP considerably (up to 9x!) with the benchmarks, Si256_hse, B.hR105_hse, PdO4, and PdO2, but has a slight benefit with GaAsBi-64 and CuC_vdw at node count one. Note that to use MPS, the NCCL must be disabled (with LUSENCCL=.FALSE.), which unfortunately slows down the code significantly (see Section III-D). VASP can automatically detect when GPUs are oversubscribed and disable NCCL. The benchmark GaAsBi-64, the smallest among the six selected benchmarks, uses k-point parallelism. So it is expected that MPS may help its performance at lower node counts, but the benefit was not significant ($\sim$3%) for this benchmark. But for an even smaller system with many k-points, a considerable performance benefit is expected by running multiple processes simultaneously per GPU. Therefore, MPS can enable better acceleration for small systems with many k-points on GPUs, which would otherwise not get performance benefits on GPUs.

For the smallest workloads where even one GPU cannot be saturated with one VASP job, Perlmutter provides the shared queue, which partitions a GPU into multiple segments using the NVIDIA A100 MIG feature [14]. The shared queue allows multiple users and jobs to share a GPU with complete job isolation securely. However, the shared queue was only recently made available on Perlmutter in April 2023.

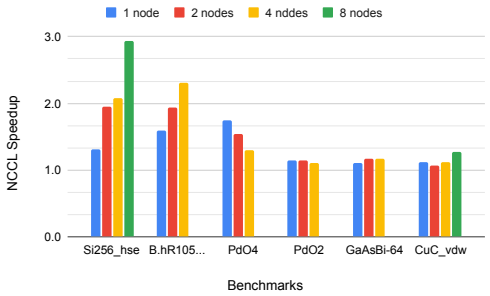


Fig. 8. NCCL effect on VASP performance on Perlmutter GPU nodes. The horizontal axis shows the benchmarks and the vertical axis shows the speedup from using NCCL, which is the ratio of the time without and with NCCL. NCCL is used in VASP by default, but it can be disabled by adding LUSENCCL=.FALSE. in the VASP INCAR file. The results at several different node counts are displayed for each benchmark.

D. Performance effect of NCCL

The NVIDIA Collective Communication Library (NCCL) implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and networking. VASP uses NCCL for GPU communications by default as an alternative to MPI where possible. This is an important optimization for the VASP OpenACC port for NVIDIA GPUs [15]. In this section, we explore how much performance gain VASP gets from using NCCL.

Figure 8 shows the NCCL effect on VASP performance, where the vertical axis shows the speedup from NCCL compared to the performance when NCCL is not used. Without exception, VASP speeds up when NCCL is active. The extent of the speedup is most significant with the HSE workloads (up to 3x). For other workloads, the speedup is about 1.1x - 1.25x in most cases.

One constraint of using NCCL is that processes that communicate through NCCL may not share a GPU, preventing jobs utilizing MPS from getting the NCCL performance benefit.

E. Does the number of NICs used per node affect VASP performance?

Perlmutter compute nodes employ the Slingshot 11 interconnect [16] fabric with HPE Cray's proprietary Cassini NICs (See Figure 4). While each CPU-only node is connected to one NIC, each GPU-accelerated node is connected to four NICs designed to provide sufficient inter-node bandwidth. How many NICs to include for each compute node is an important design choice for supercomputers. It would be interesting to study how the number of NICs per node affects VASP performance.

By default, when multiple NICs are available on the node, Cray MPICH will attempt to use them all [17]. However, it provides options for applications to use a desired number of NICs per node and a desired mapping for the MPI ranks to these NICs at runtime. We first investigated the performance effect of different mappings via the environment variable, MPICH_OFI_NIC_MAPPING, which specifies the precise rank-to-NIC mapping on the node. But we

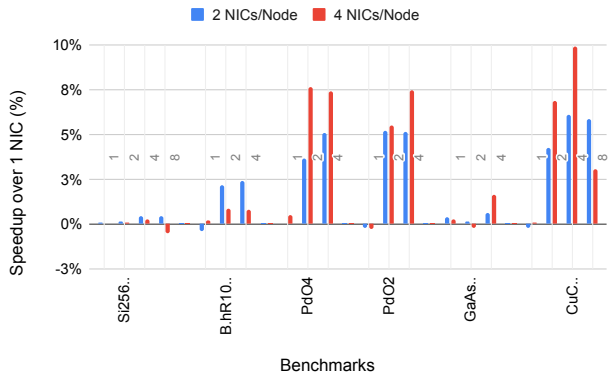


Fig. 9. This figure shows the performance effect of the number of NICs per node on Perlmutter GPUs. The horizontal axis shows the benchmarks, and the vertical axis shows the speedup when multiple NICs are used compared to running with one NIC per node. The speedup is defined as [Time(multi-NICs)-Time(1-NIC)]/Time(1-NIC). The blue and red bars show the results for the number of NICs per node two and four, respectively. The results at several different node counts are displayed for each benchmark.

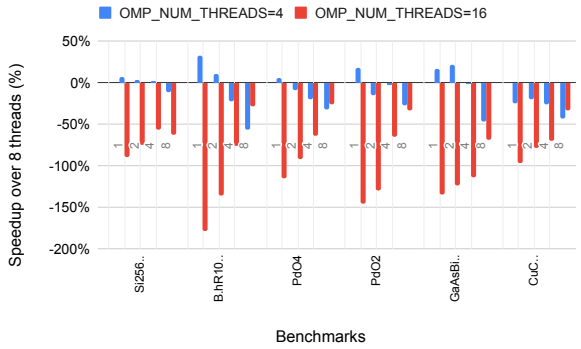


Fig. 10. This figure shows the hybrid OpenMP+MPI VASP thread performance on Perlmutter Milan CPU nodes. The horizontal axis shows the benchmarks, and the vertical axis shows the percentage speedup over the eight threads per task performance when four (blue bars) and 16 (red bars) threads per task are used, defined as [[Time(8 Threads)-Time (4 or 16 threads)]/time(8 Threads)]x100%. Thus positive numbers show a speedup; negative numbers show a slowdown. For each benchmark the results for several different node counts, 1, 2, 4, and 8, are displayed as these benchmarks can be run at a range of node counts in practice.

didn't observe any visible difference in VASP performance on GPUs. Next, we tested if the number of NICs used per node affects VASP performance via the environment variable, MPICH_OFI_NUM_NICS, which specifies the number of NICs the job can use on a per-node basis. We set the environment variable MPICH_OFI_NIC_VERBOSE=2 to confirm the desired number of NICs and rank-to-NIC mapping are indeed honored.

Figure 9 shows the performance effect of the number of NICs on VASP performance on Perlmutter GPUs. The figure shows the performance speedup when using more NICs over the one NIC per node performance. VASP performance improves when using more NICs per node but by an insignificant amount. The speedup is within 10% at best. And for Si256_hse, the number of NICs has a negligible effect on VASP performance. Since the choice of NICs would affect the inter-node performance, as one can see for the single-node runs, there is no difference between the choice of different numbers of NICs for all benchmarks. Note that the selected benchmarks are run on a small number of nodes, 1-8, and that could be the reason why the number of NICs choice didn't affect VASP performance by a significant amount. In addition, NCCL was disabled in these tests; otherwise, the jobs hung.

In practice, VASP can run with the default setting, i.e., four NICs per node, and users don't have to worry about how to map them on the specific NICs on the node.

IV. VASP Performance on Milan CPUs

Perlmutter has GPU-accelerated and CPU-only nodes consisting of two AMD Milan CPUs. While this paper focuses on VASP performance on Perlmutter GPUs, some new features specific to AMD Milan CPUs are worth exploring. For example, compared to Intel Haswell and KNL nodes on Cori, Perlmutter Milan CPU nodes have more CPU cores and NUMA domains and deploy a different high-speed network - HPE Slingshot interconnect. This section will address how these features affect VASP performance on Perlmutter Milan CPU nodes.

A. Optimal threads per task

On Perlmutter CPU nodes, the hybrid OpenMP+MPI VASP port is used. To run the hybrid OpenMP+MPI VASP, the first question to address is how many threads per task are optimal. Since Perlmutter CPU nodes have eight NUMA domains, starting with eight threads per task is natural. Figure 10 shows the VASP thread performance on Perlmutter CPU nodes for the six selected benchmarks. The figure shows the best performance achieved with eight OpenMP threads most of the time, consistent with what we have seen with VASP thread performance on Cori Haswell and KNL nodes [7]. But a few benchmarks outperform eight threads when using four threads per task at node count one or two (see the blue bars above the horizontal axis). This is not surprising, showing the performance trade-off between the MPI and OpenMP parallelisms in the code. Increasing MPI tasks by reducing threads per task results in better performance at a smaller node count where more MPI tasks are appreciated. In contrast, when a benchmark is run with more nodes than its optimal node count, more OpenMP threads per task could help VASP to scale further to more nodes.

In practice, using eight threads per task is a safe choice most of the time; however, for small node counts, increasing the total number of MPI tasks by using four threads per task or less may help performance.

B. AMD Simultaneous Multithreading (SMT)

AMD Simultaneous Multithreading (SMT) is equivalent to the Hyper-Threading on Intel processors. Perlmutter Milan CPU nodes have two hardware threads per core, so each node can run up to 256 MPI processes or threads at total capacity.

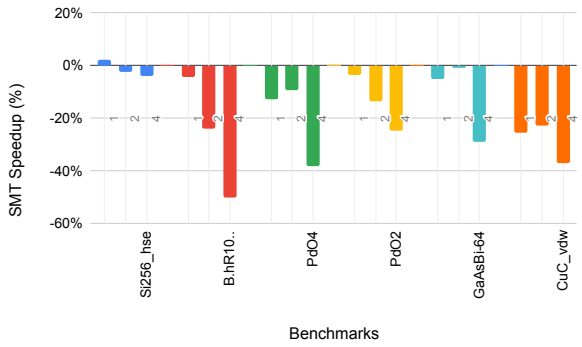


Fig. 11. This figure shows the percentage speedup of VASP when using SMT. The percentage speedup is defined as [[Time(noSMT)-Time(SMT)]/Time(noSMT)]x100% The horizontal axis shows the six selected benchmarks. The results of running with 1, 2, and 4 nodes are displayed for each benchmark. All runs used eight threads per task. SMT runs use twice as many MPI tasks as those without SMT at each node count.

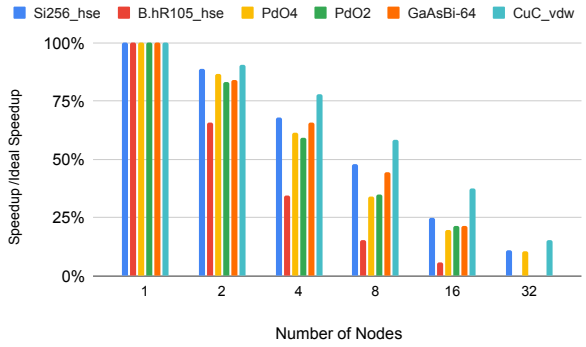


Fig. 12. Parallel efficiency of VASP on Perlmutter CPUs. The horizontal axis shows the number of nodes, and the vertical axis shows the parallel efficiency of VASP (speedup/ideal parallel speedup). All tests ran with 16 MPI tasks per node, each with eight OpenMP threads.

A previous study on Hyper-Threading [18], [7] concluded that Hyper-Threading does not help VASP performance on Intel Ivy Bridge, Haswell, and KNL processors. However, we still investigated if VASP gains any performance from using AMD SMT, as they are additional compute resources on the Milan CPU nodes not to leave any performance on the table.

Figure 11 shows the performance effect of SMT on the hybrid OpenMP+MPI VASP on Perlmutter CPUs. As seen with Hypter-Threading on Intel processors, SMT does not help VASP performance on AMD Milan CPUs. For the large HSE benchmark, Si256_hse, there is a slight (<2%) performance benefit from using SMT at node count one. However, the benchmark scales to multiple nodes (see next section); therefore, it would run with a node count that is larger than one to shorten the time to solution. Thus, the extra hardware threads on each core can be safely ignored.

C. Parallel efficiency of VASP on Perlmutter CPU nodes

As described in Section III-A, understanding the parallel scaling of applications is critical to run the applications efficiently on HPC systems. Figure 12 shows the parallel efficiency of the VASP for the six selected benchmarks. Due to the increased parallelism on the single node, the parallel efficiency drops quickly when the number of nodes increases. Again, the parallel efficiency depends on the workloads and problem sizes. For the larger HSE benchmark, Si256_hse (blue bars), VASP scales to four nodes with 68% parallel efficiency, while for the smaller benchmark, B.hR105_hse (red bars), VASP does not scale over two nodes. Roughly distributing 200 bands per node is a reasonable estimate for large HSE workloads. Similarly, VASP scales to four nodes for CuC_vdw, suggesting about 200 bands per node. VASP does not scale over two nodes with the DFT workloads (PdO4, PdO2, and GaAsBi-64), regardless of their sizes.

In practice, for the best interest of utilizing the compute resources efficiently, one may not run VASP at a parallel efficiency lower than 70%. Therefore, using one or two nodes would be sufficient for systems containing up to several hundred atoms. This evaluation applies to the system containing one k-point group. For the systems comprising many k-points running multiple k-point groups simultaneously (with KPAR), the number of nodes to use can be multiplied by the number of k-point groups.

V. ACFDTR Workloads

The ACFDTR algorithm for Random Phase Approximation (RPA) (Algo=ACFDTR) has been ported to GPUs recently and is available in VASP 6.4.0 and up. Running ACFDTR workloads on GPUs significantly shortens the time to solution. In addition, it can be done in an all-in-one mode conveniently instead of the previous multi-step approach. However, compared to the workloads discussed, it is more compute-intensive and uses a significantly larger memory per task. Furthermore, increasing the number of nodes does not help reduce the memory footprint per task in the current implementation (v6.4.1). The exact diagonalization step required in this algorithm implementation is yet to be distributed over multiple GPUs, while the hybrid OpenMP+MPI code has already implemented the distributed diagonalization over CPUs. An effort to remove this bottleneck is underway. In this section, we will compare the VASP performance on Perlmutter GPUs and CPUs, and derive best practice tips from the performance observed.

Figure 13 shows the strong scaling of VASP with the Si128_acfdrt benchmark on Perlmutter GPU nodes. VASP does not scale over multiple GPU nodes with this workload. At node count two, the parallel efficiency is already dropped to 65%. When using more nodes, the run time does not reduce but increases. Figure 14 shows the strong scaling of VASP (hybrid OpenMP+MPI) when running benchmark Si128_acfdtr on Perlmutter CPU nodes. VASP scales better on CPUs than GPUs, where we do not distribute the exact diagonalization step. The parallel efficiency at node count four is 70%, which can be used to reduce the runtime effectively. Figure 15 shows the VASP GPU speedup over CPUs with Si128_acfdtr. Regardless of the poor parallel scaling, running

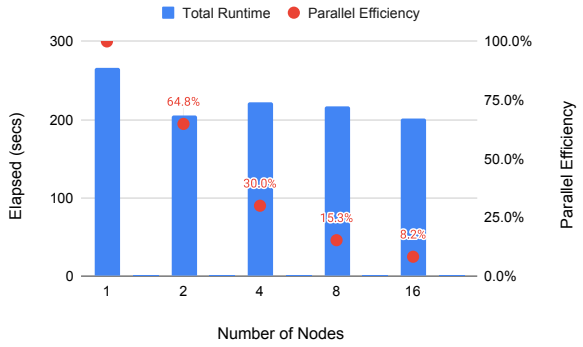


Fig. 13. VASP strong scaling performance on Perlmutter GPU nodes with Si128_acfdtr. The horizontal axis shows the number of nodes, the left vertical axis shows the total runtime of VASP (Elapsed Time), and the right vertical axis shows the parallel efficiency of VASP (speedup/ideal parallel speedup). All tests ran with four MPI tasks per node (one task per GPU), each with one OpenMP thread. Note the multi-node jobs were run on the 80 GB GPU nodes because the two-node job ran out of memory.

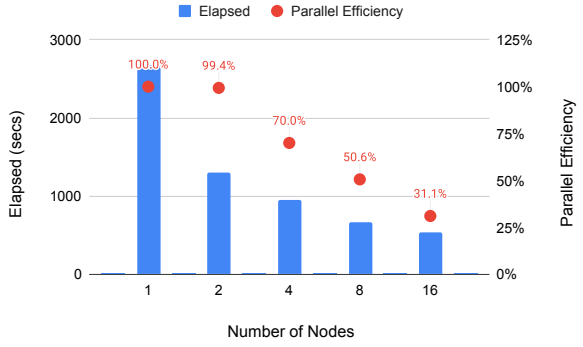


Fig. 14. VASP strong scaling performance on Perlmutter CPU nodes with Si128_acfdtr. The horizontal axis shows the number of nodes, the left vertical axis shows the total runtime of VASP (Elapsed Time), and the right vertical axis shows parallel efficiency (speedup/ideal parallel speedup). All tests ran with 16 MPI tasks per node, each with eight OpenMP threads.

this benchmark on GPUs can get up to 10x speedup on a single node performance; furthermore, a single GPU node outperforms four CPU nodes by a factor of 3.6.

In summary, the compute-intensive ACFDTR algorithm has been ported to GPUs, significantly reducing the time to solution. While the hybrid OpenMP+MPI code scales with multiple CPU nodes, the GPU port of ACFDTR hardly scales to multiple nodes yet. However, the GPU port is significantly faster than the CPU port; therefore, running ACFDTR workloads on GPUs is recommended. The ACFDTR GPU implementation is memory intensive. Consider using the 80 GB GPU nodes for systems requiring significantly more memory. The memory bottleneck will be removed to enable larger system computation in the future. Our previous attempt at running Algo=ACFDTR for a system containing 250 silicon atoms with one defect failed due to insufficient memory, even on Perlmutter's 80GB memory GPU nodes.

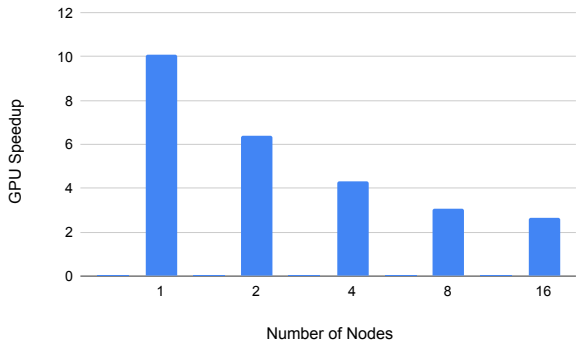


Fig. 15. VASP GPU speedup for benchmark Si128_acfdtr. The horizontal axis shows the number of nodes, and the vertical axis shows the GPU speedup of VASP (Time(CPU)/Time(GPU)).

VI. Energy and Charging Efficiency

A. Charging efficiency

At NERSC, machine usage is charged by node hours [19]. Due to limited allocations, users are concerned about charging in addition to shortening the time to solution. In general, no MPI code can reach 100% parallel efficiency due to the serial portion of the code (Amdahl's law) and various overheads, e.g., data communication and movements, I/O, etc. Therefore, using more nodes usually incurs more charging when running parallel applications.

Figure 16 shows the charging (node hours) for the benchmarks Si256_hse and PdO4 when running on Perlmutter GPU nodes. As shown in Section III-A, VASP scales well to eight nodes at 75% parallel efficiency with Si256_hse performing an HSE calculation but does not scale well beyond two nodes with PdO4 performing a DFT calculation. Therefore, the charging quickly increases when running with an increased number of nodes for PdO4, while it increases slowly with Si256_hse. So, to reduce charging, one must avoid running VASP outside the parallel scaling region and use fewer nodes where possible. However, in practice, more nodes are appreciated for various reasons, e.g., to get the results sooner, to accommodate memory requirements, etc.

Users have options to run jobs either on Perlmutter CPU or GPU nodes. Figure 17 shows the charging comparison when running the benchmarks on Perlmutter CPUs and GPUs. The number of nodes, MPI processes, and OpenMP threads for each benchmark was selected to allow CPU and GPU jobs to run at their optimal setting. Since the charge factor is the same for the Perlmutter CPU and GPU nodes, Figure 17 also shows the GPU speedup over CPU runs. Running on GPUs can significantly reduce the charging. Therefore, VASP should be run on GPUs to reduce the charging whenever possible.

B. Energy efficiency

In the past, the application performance was measured solely by its time to solution. However, power is becoming an increasingly limiting factor in supercomputing now. Therefore,

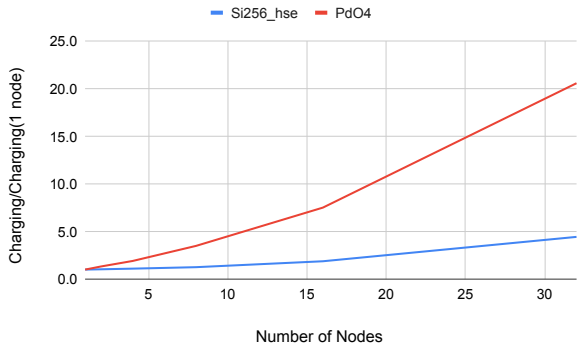


Fig. 16. VASP charging when running on Perlmutter GPU nodes. The horizontal axis shows the number of GPU nodes, and the vertical axis shows charging, i.e., the node hours, for Si256_hse (blue) and PdO4(red) in ratio to the single node charging when running on Perlmutter GPU nodes.

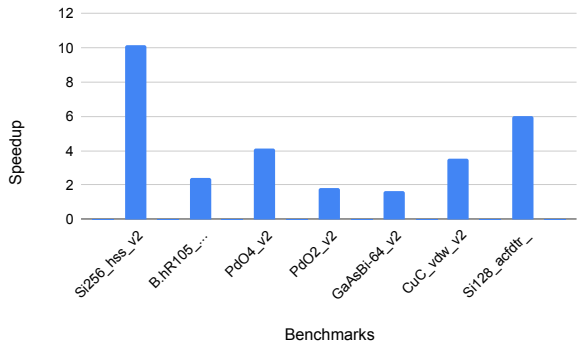


Fig. 17. Charging comparison between running on Perlmutter CPU nodes and GPUs. The horizontal axis shows the benchmarks, and the vertical axis shows the charging ratio when running on CPUs to GPUs. The number of nodes, MPI processes, and OpenMP threads used to run these benchmarks are shown in Table II.

the performance and scale of future high-performance computing systems will be determined by how efficiently they manage their power budgets [20]. As the top #1 code at NERSC VASP consumes more than 20% of computing cycles at NERSC, it is essential to investigate the most power-efficient way to run VASP jobs on Perlmutter.

Figure 18 shows the total energy comparison for VASP jobs run on Perlmutter GPU and CPU nodes. One can see that running VASP jobs on GPU nodes consumes significantly less energy than running them CPU-only. The energy saving from running them on GPUs is most significant for large jobs, e.g., Si_hse, PdO4, and CuC_vdw, as well as Si128_acftdr, the compute-intensive ACFDTR workload. The GPU speedup line (red) shows that the energy saving is consistent with the runtime saving. Figure 18 indicates that running VASP on Perlmutter GPU nodes gets the results faster and, at the very same time, saves energy.

C. Energy usage on Perlmutter and Cori

With the HPC community moving to more power-efficient supercomputing, it would be interesting to compare the power

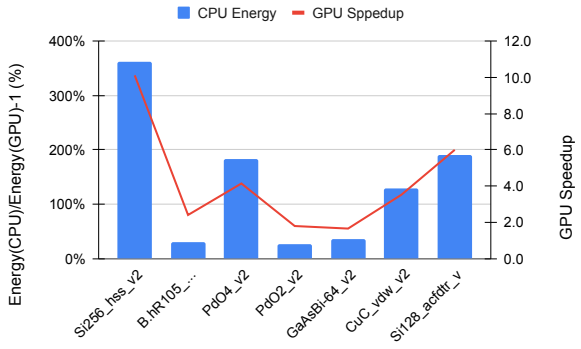


Fig. 18. Energy saving when running VASP jobs on Perlmutter GPUs over CPUs. The horizontal axis shows the benchmarks, and the vertical axis shows the energy saving when running on GPU nodes in ratio to the energy when running on GPU nodes. The number of nodes, MPI processes, and OpenMP threads used to run these benchmarks are shown in Table II. The energy usage data were collected from Slurm's accounting logs (sacct -X -o consumedenergyraw) and confirmed their consistency with the OMNI [21] power usage data at NERSC [21] for job samples using the power analysis scripts in [22].

usage of VASP jobs on Perlmutter and Cori, our previous flagship computer, to be retired by the end of May this year.

Figure 19 shows the energy usage of VASP jobs on Perlmutter and Cori. VASP consumes the least energy on Perlmutter GPU nodes (blue bars) and the most on Cori Haswell nodes (green bars). The energy saving for the larger benchmarks, Si256_hse, PdO4, CuC_vdw, and Si128_acfdtr, is greater than the rest smaller benchmarks. The energy usage on Cori KNL is larger than on Perlmutter CPU nodes with a couple of exceptions, but the differences are within 10-20%. This figure demonstrates the apparent move towards more energy-efficient systems at NERSC.

Figure 20 shows the average per-node power of the VASP jobs on Perlmutter and Cori. VASP uses significantly more power on GPUs than on CPUs. Among the CPUs, VASP uses the least power on KNL and the most on Perlmutter CPUs, consistent with their base clock rates: Milan CPU runs at 2.45 GHz (Max Boost Clock: 3.5 GHz), Intel KNL runs at 1.4 GHz (Max Turbo Frequency: 1.6 GHz), and Haswell runs at 2.3 GHz (Max Turbo Frequency: 3.6 GHz). Yet as shown in Figure 19, VASP uses the least energy on GPUs, indicating the energy saving on GPUs is from the substantial reduction in time to solution.

TABLE II
THIS TABLE SHOWS THE NUMBER OF NODES, MPI TASKS, AND OPENMP THREADS USED FOR THE BENCHMARK RUNS IN FIGURE 19, AND 20.

	#nodes	A100 GPUs		Milan CPUs		Haswell (KNL)	
		MPI	OMP	MPI	OMP	MPI	OMP
Si256_hse	4	16	1	64	8	16 (32)	8
B.hR105_hse	1	4	1	16	8	4 (8)	8
PdO4	2	8	1	32	8	8 (16)	8
PdO2	2	8	1	32	8	8 (16)	8
GaAsBi-64	2	8	1	32	8	8 (16)	8
CuC_vdw	4	16	1	64	8	16 (32)	8
Si128_acfdtr	2	8	1	32	8	8 (16)	8

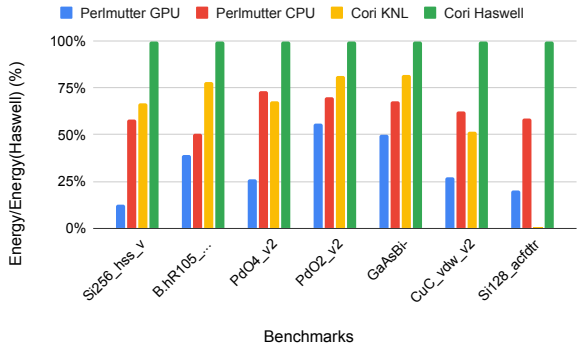


Fig. 19. Total energy usage comparison when running VASP on Perlmutter and Cori. The horizontal axis shows the benchmarks slightly modified to run longer to mimic production runs. The vertical axis shows the energy used by VASP benchmark jobs on Perlmutter GPUs (blue bars), CPUs (red bars), Cori KNL (yellow bars), and Cori Haswell(green bars) in ratio to the Cori Haswell usage. The number of nodes, MPI tasks, and OpenMP threads used for each benchmark are listed in Table II.

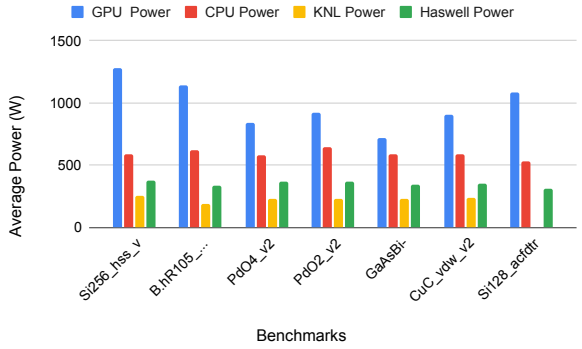


Fig. 20. VASP's average power usage per node on Perlmutter and Cori. The horizontal axis shows the benchmarks slightly modified to run longer to mimic production runs. The vertical axis shows the average power used per node by VASP benchmark jobs on Perlmutter A100 GPUs (blue bars), Milan CPUs (red bars), Cori KNL (yellow bars), and Haswell (green bars). The number of nodes, MPI tasks, and OpenMP threads used for each benchmark are listed in Table II.

VII. Summary

In this paper, we have studied VASP performance on Perlmutter GPU and CPU nodes using seven benchmarks representing the VASP production workloads at NERSC. We measured strong scaling, investigated the performance impact of OpenMP threads, MPS, NCCL, and the number of NICs per node on the VASP performance, and derived the best practice tips for users from these results. Here is the summary of the best practice tips for users:

- One or two GPU or CPU nodes would be sufficient for systems containing up to several hundred atoms (e.g., 300 atoms) on Perlmutter. The benchmarks studied in this paper, which contain representative workloads, problem sizes, and elements, can provide a good reference when selecting the number of nodes for VASP jobs. For example, for large HSE calculations, roughly distributing 100 and 200 bands per GPU and CPU node, respectively, would be a reasonable estimate.
- Enabling eight or 16 threads per task to utilize otherwise idling CPU resources for non-HSE workloads can get a small additional speedup.
- Running one task per GPU gives optimal performance, but MPS may accelerate small systems with many k-points on GPUs.
- NCCL greatly improves VASP performance. It should be used whenever possible, especially for HSE workloads. When using NCCL, one must run one task per GPU.
- For hybrid OpenMP+MPI VASP, using eight threads per task is a safe choice most of the time; however, for small node counts, increasing the total number of MPI tasks by using four threads per task or less may help VASP performance.
- SMT does not help VASP performance on AMD Milan CPUs.
- Algo=ACFDTR is now ported to GPUs, bringing a substantial speedup over running on CPUs. ACFDTR workloads are memory intensive; consider using the 80 GB memory GPU nodes.
- Running VASP on GPUs gets the results faster, reduces charging, and saves energy.

Acknowledgment

The authors would like to thank the VASP users at NERSC who participated in the VASP usage survey in 2023, from which the representative workloads were derived. In addition, they thank Sridutt Bhalachandra at NERSC for sharing his power analysis script; Nicholas Wright and the members of the Advanced Technologies Group at NERSC for providing valuable input. This work used the resources of the National Energy Scientific Computing Center (NERSC) at the Lawrence Berkeley National Laboratory.

References

[1] Perlmutter, a HPE Cray EX system, https://docs.nersc.gov/systems/perlmutter/architecture/
[2] Cori, a Cray XC40 system, https://www.nersc.gov/systems/cori/
[3] G. Kresse and J. furthmüller, "Efficiency of *ab initio* total energy calculations for metals and semiconductors using a plane-wave basis set", Comput. Mater. Sci. **6**, 15 (1996); G. Kresse and J. Furthmüller, "Efficient iterative schemes for total-energy calculations using a plane-wave basis set", Phys. Rev. B **54**, 11169 (1996).
[4] Martijn Marsman, Stefan Maintz, Alexey Romanenko, Markus Wetzstein, and Georg Kresse, Porting VASP to GPU using OpenACC: exploiting the asynchronous execution model, OpenACC Annual Meeting, Aug. 31st 2020, https://www.openacc.org/sites/default/files/inline-images/events/F2F20%20presentations/BoF_VASP_OpenACC_2020%20(1).pdf
[5] Stefan Maintz, and Markus Wetzstein, Strategies to Accelerate VASP with GPUs Using OpenACC, Available Online: https://cug.org/proceedings/cug2018_proceedings/includes/files/pap153s2-file1.pdf
[6] VASP Wiki for OpenACC port, https://www.vasp.at/wiki/index.php/OpenACC_GPU_port_of_VASP
[7] Z. Zhao, M. Marsman, F. Wende, and J. Kim, Performance of Hybrid MPI/OpenMP VASP on Cray XC40 Based on Intel Knights Landing Many Integrated Core Architecture, Available Online: https://cug.org/proceedings/cug2017_proceedings/includes/files/pap134s2-file1.pdf
[8] NVIDIA Multi-Process Service (MPS), https://docs.nvidia.com/deploy/mps/index.html

[9] Materials Project, https://materialsproject.org/
[10] F Wende, M Marsman, Z Zhao, J Kim, Porting VASP from MPI to MPI+ OpenMP [SIMD], Scaling OpenMP for Exascale Performance and Portability - 13th International Workshop on OpenMP, IWOMP 2017, Stony Brook, NY, USA, September 20-22, 2017, Proceedings.
[11] F Wende, M Marsman, J Kim, F Vasilev, Z Zhao, T Steinke, OpenMP in VASP: Threading and SIMD, International Journal of Quantum Chemistry 119 (12), e25851
[12] NVIDIA Collective Communication Library (NCCL), https://developer.nvidia.com/nccl
[13] VASP INCAR tags, https://www.vasp.at/wiki/index.php/Category:INCAR_tag
[14] Multi-Instance GPU (MIG), https://docs.nvidia.com/datacenter/tesla/mig-user-guide/
[15] Stefan Maintz, Alexey Romanenko, Harry Petty, Jonathan Lefman and Chris Porter, Scaling VASP with NVIDIA Magnum IO, https://developer.nvidia.com/blog/scaling-vasp-with-nvidia-magnum-io/
[16] HPE Slighshot 11, https://www.hpe.com/us/en/compute/hpc/slingshot-interconnect.html
[17] Cray MPICH man page for Slinghot options, ”man intro_mpi” on HPE Cray EX.
[18] Zhengji Zhao, Nicholas J. Wright and Katie Antypas, “Effects of Hyper-Threading on the NERSC workload on Edison”, Cray User Group meeting, May 6-9, 2013, Napa Valley, CA.
[19] NERSC QOS Limits and Charges, https://docs.nersc.gov/jobs/policy/#qos-limits-and-charges
[20] Sridutt Bhalachandra, Brian Austin, Nicholas J. Wright, ”Understanding power variation and its implications on performance optimization on the Cori supercomputer”, SC21 PMBS workshop, Saint Luis, MO, USA
[21] OMNI, NERSC data collection infrastructure for the performance monitoring data
[22] Sridutt Bjalachandra, Perlmutter OMNI Analysis, https://gitlab.com/NERSC/perlmutter-omni-analysis