Comparing Compiler and Library Performance in Material Science Applications on Edison

Jack Deslippe and Zhengji Zhao
NERSC
Lawrence Berkeley National Laboratory
Berkeley, CA USA
Email: jrdeslippe@lbl.gov

***Abstract*—Materials science and chemistry applications are expected to represent approximately one third of the computational workload on NERSC's Cray XC30 system, Edison. The performance of these applications can often depend sensitively on the compiler and compiler options used at build-time. For this reason, the NERSC user services group supplies users with optimized builds of the most commonly used materials science applications in order to ensure these cycles are used as efficiently as possible. The materials science and chemistry codes in question are written in Fortran, C, C++, or a combination of these languages, and use MPI or other message passing libraries, OpenMP, as well as linear algebra and FFT libraries. In this paper, we compare the performance of various material science and chemistry applications when built with the Cray, Intel and GNU compiler suites as well as linked against the MKL, LibSci and FFTW libraries. We compare the optimal compilers and libraries on Edison with those previously obtained on the NERSC Cray XE6 machine, Hopper.**

***Keywords*-component; formatting; style; styling;**

I. Introduction

Materials science and chemistry represent one of the largest science areas in NERSC's workload. Application codes from these fields typically consume approximately 1/3 of the compute cycles at NERSC every year. These codes are used in research projects that are at the core of many of the Department of Energy's energy science goals: for example, the development of next generation solar energy and carbon capture materials. In order to increase the productivity of our users and increase the scientific output of the center, NERSC supports a set of pre-compiled materials science and chemistry programs that are commonly used when conducting research in materials science and chemistry. This effort is designed to save time for the researchers (who can subsequently focus more time on science instead of code compilation) as well as to save compute cycles by ensuring that researchers are using an optimized version of each code.

Fully optimizing a given application's performance often requires a deep understand of the source, an accurate profile for a representative run and the ability to have changes to the source accepted upstream. However, in many cases, significant performance gains can be achieved by simply optimizing the code over the matrix of possible compilers, compiler options and libraries available on a given machine. In this paper, we explore the performance variability of six common materials science applications at NERSC with respect to the compilers and libraries available on Edison, NERSC's Cray XC30 [1].

NERSC currently supports compilers from three different vendors on the XC30 system, Edison: Intel, GNU and Cray. NERSC's Cray XE6 system, Hopper, additionally supports the PGI compilers. On Hopper, it has previously been shown that each of the above compilers produce distinct performance results, with variations as large as 20% [2]. Thus, it is of interest to the center to determine the optimal compiler choices for Edison early in the machine's life cycle. Additionally, materials science applications generally rely heavily on math libraries such as FFTW, BLAS, LAPACK and ScaLAPACK. NERSC provides several library options for these routines on Edison: FFTW2, FFTW3, LibSci and MKL. In this paper, we compare the performance of BerkeleyGW [3], Quantum ESPRESSO [4], VASP [5] LAMMPS [6], NAMD [7] and NWChem [8] with the compilers and libraries listed above.

II. Methods

All compilations, except those done on the XE6 machine Hopper for comparison purposes, were run on Edison, NERSC's Cray XC30 system.

Jobs were run out of directories on the local Lustre scratch filesystem, where all IO was performed. IO was additionally minimized through run-time options when available. For the purpose of checking reproducibility, all computations were performed at least twice, with the lowest runtime chosen in each case and subsequent runs performed if the runtimes differed in a significant way.

We test each application build at a range of MPI tasks and, when available, OpenMP threads to confirm the performance characteristics from a set of tests is sustained over different concurrencies.

There are three different compilers available to build applications on Edison: Intel, Cray and GNU. The Cray compiler was excluded from the NWChem tests because there were irresolvable error messages generated during

Compiler	Flags
GNU	-O3 -ffast-math
Cray	(default)
Intel	-fast -no-ipo

Table I
COMPILER OPTIONS USED FOR COMPILERS ON EDISON.

compilation or run time. The GNU compiler was excluded from the VASP test due also to irresolvable runtime errors.

We chose compiler flags for each of the above compilers based on the recommendation from previous NERSC studies of compiler optimization options on Edison [9]. The recommendations were based on performance studies across a benchmark suite covering many application areas. The used compiler flags are listed in Table I. The compiler optimizations considered for production builds are limited to those builds with resulting binaries that pass strict validity checks and can be used for scientific calculations. These builds do not necessarily represent the highest optimizations the compilers can reach.

When running hyrbrid MPI-OpenMP calculations (with BerkeleyGW and Quantum ESPRESSO) we distribute the MPI tasks evenly across the NUMA nodes (using the aprun "-S" option) and use the aprun options "-cc numa_node" and "-ss" which relax thread core binding to binding only to a NUMA node and restrict memory usage from crossing NUMA node boundaries.

Hyper-threading was not used in any of the benchmarks.

We list below the math libraries used in the optimization of the test applications.

A. FFTW (2 & 3)

FFTW [10] is a C subroutine library for computing the discrete Fourier transform in one or more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, i.e. the discrete cosine/sine transforms). FFTW3 may be built with a pthread library (libfftw3_threads) or an OpenMP threaded library (libfftw3_omp). The use of the threaded libraries requires minor modifications the applications source code if threaded FFTW calls are not already supported (e.g. in Quantum ESPRESSO). In this paper, we used FFTW3 3.3.0.1 unless otherwise specified.

B. MKL

MKL contains highly optimized, extensively threaded math routines for science, engineering, and financial applications. Core math functions include BLAS, LAPACK, ScaLAPACK, Sparse Solvers, Fast Fourier Transforms and more. MKL exposes FFT routines through a variety of interfaces, two of which are made to match the FFTW2 and FFTW3 interfaces. Thus, MKL's FFTW interfaces can be used as a drop in replacement for codes expecting to link against FFTW2 or FFTW3. In this paper, we used MKL from composer_xe_2013.1.117.

C. LibSci

The Cray scientific libraries package is a collection of numerical routines optimized for best performance on Cray systems. The package includes: BLAS, BLACS, LAPACK, ScaLAPACK, and more. In this paper, we used LibSci version 12.0.0.0.

III. APPLICATIONS

A. BerkeleyGW

Version 1.1 (Beta)

BerkeleyGW [3] is a Fortran 90 material science application for calculating the spectroscopic, or excited state, properties of materials starting with Density Function Theory (DFT) inputs from a variety of codes, including Quantum ESPRESSO [4]. We tested the epsilon executable, which computes the dielectric matrix from a set of input DFT orbitals. This executable typically represents the bottleneck in a spectroscopic computation. The epsilon executable is also ideal for testing library performance, since there are well-defined regions with computational tasks in which 3D FFT, BLAS (ZGEMM), and SCALAPACK library calls dominate the compute cycles.

We tested a pre-release of BerkeleyGW 1.1 which is a Hybrid MPI-OpenMP code relying both on explicit OpenMP do loops as well as threaded FFT and Linear Algebra libraries. We varied the number of MPI tasks, number of threads per MPI task, as well as the compiler and libraries across runs.

In the BerkeleyGW benchmark calculation, we consider the (8,0) single walled carbon nanotube (SWCNT) with an 80 Ry. wavefunction cutoff, 14 Ry. dielectric cutoff and 256 empty states. The nanotube is placed in a supercell of volume $1.1 \cdot 10^4 AU^3$ leading to a dielectric matrix of size 9770x9770.

When linking MKL libraries with the Cray compiler, the GNU MKL libs were used.

B. Quantum ESPRESSO

Version 5.0.2

Quantum Espresso [4] is a Fortran 90 material science program that performs electronic structure calculations within Desnsity Functional Theory (DFT) for materials modeling at the nanoscale level. Quantum ESPRESSO utilizes a plane wave (PW) basis set and supports norm-conserving and ultra-soft pseudopotentials as well as PAWs. Tests were run using the default Davidson diagonalization scheme.

FFTs are used extensively in Quantum ESPRESSO for the application of the Hamiltonian, H, to a wavefunction ψ. The default FFT library for Quantum ESPRESSO is FFTW3, with an option to use an internal FFTW2 version. Despite being, in principle, a hybrid MPI-OpenMP code, by

default Quantum ESPRESSO does not support the threaded FFTW3 libraries, and contains explicit OpenMP loops only around FFTs done with the internal FFTW2 version. We therefore made a small code modification allowing support for threaded FFTW3. We tested the FFTW3 and MKL FFT libraries. Threaded linear algebra calls do not require any code modifications. We tested the MKL and LibSci linear algebra libraries. Our tests were performed over 16 to 256 cores.

In the QE benchmark, we perform a self-consistent field (SCF) calculation on the (8,0) single walled-carbon nanotube (SWCNT) with an 100 Ry wave-function cutoff, 128 bands, 12 kpoints, in an $1.1 \cdot 10^4 AU^3$ unit cell. We vary the number of MPI tasks and OpenMP threads and use 4 k-point pools for each run.

When linking MKL libraries with the Cray compiler, the GNU MKL libs were used.

C. VASP

Version 5.3.3

VASP [5] is a DFT program that computes approximate solutions to the coupled electron Kohn-Sham equations for many-body systems. The code is written in Fortran 90 and MPI, and uses FFT and BLAS/Lapack linear algebra libraries. Plane waves basis sets are used to express electron characteristics such as electron wavefunctions, charge densities, and local potentials. Pseudopotentials and PAWs are used to describe the interactions between electrons and ions. The VASP benchmark was performed using two test cases provided by NERSC users. The first test system contains 154 atoms and used to test a commonly used diagonalization scheme. The other benchmark test is for a hybrid calculation for a system containing 105 atoms. We tested the VASP performance over GNU, Intel and Cray compilers, the MKL and LibSci BLAS/LAPACK libraries, and three FFT libraries options, FFTW, MKL and an internal FFTW library named "FURTH." We ran each benchmark test at different concurrencies. Our tests were designed to check if the performance sustains itself over the different concurrencies, computation types and system sizes. For VASP, we used an additional crayftn flag, "-0 ipa0", which disables inter-procedural analysis.

D. LAMMPS

Version 22Mar13

LAMMPS [6] is a C++ classical large-scale molecular dynamics code. It computes Newtons equations of motion for systems of particles in a liquid, solid, or gaseous state. Only the compiler itself was varied in the test, due to minimal dependence on libraries. Each compiler was tested using the three LAMMPS benchmark problems described below:

1) LJ: Atomic fluid, Lennard-Jones potential with 2.5 sigma cutoff (38 neighbors per atom), NVE integration.

2) Chain: Bead-spring polymer melt of 100-mer chains, FENE bonds and LJ pairwise interactions with a $2^{(1/6)}$ sigma cutoff (5 neighbors per atom), NVE integration.

3) Rodo: Rhodospin protein in solvated lipid bilayer, CHARMM force field with a 10 Angstrom LJ cutoff (375 neighbors per atom), particle-particle particle-mesh (PPPM) for long-range Coulombics, NPT integration.

E. NWCHEM

Version 6.1.1

NWChem [8] is a chemistry application containing many physical approaches, such as MP2, CCSD and DFT, that is designed to be scalable and functional on high performance, parallel compute systems. It is a Fortran code, and its parallelization is implemented with the Global Array library. It can perform quantum mechanic functions, classical functions, hybrid functions, potential energy surface analysis, and electronic structure analysis.

Our NWChem benchmark is a coupled cluster test case from the standard NWChem distribution, cytosine_ccsd.nw. We tested the performance difference when using different compilers while linked to the 64 bit integer MKL library. We test performance when run with between 32 and 256 cores.

We used the armci-mpi library with GA-5.0 for our study.

We were unable to produce a successful NWChem executable with the Cray compilers.

F. NAMD

NAMD [7] is a C++ chemistry application that performs molecular dynamic simulations that compute atomic trajectories by solving equations of motion numerically using empirical force fields. The Particle Mesh Ewald algorithm provides a complete treatment of electrostatic and Van der Waals interactions. NAMD is parallelized through a communication library called Charm++. NAMD uses FFTW libraries. NAMD uses FFT libraries. However, since the runtime is not sensitive to the choice of the FFT libraries, we used FFTW2 single precision libraries in the computation. In addition, we were unable to produce a working executable with Cray compilers. Therefore our performance comparison was limited between Intel and GNU compilers. The Benchmark used for NAMD was the standard STMV 1,066,628-atom system.

IV. Results

A. BerkeleyGW

BerkeleyGW was tested with all of GNU, Intel and Cray compilers and FFTW, LibSci and MKL. A summary of the overall performance of the code on our example benchmark calculation is shown in Fig. 1.

As seen in the figure, the Intel Compiler + MKL library result is the best overall combination for both 1 and 4 OpenMP threads per MPI task. For a single thread, the MKL

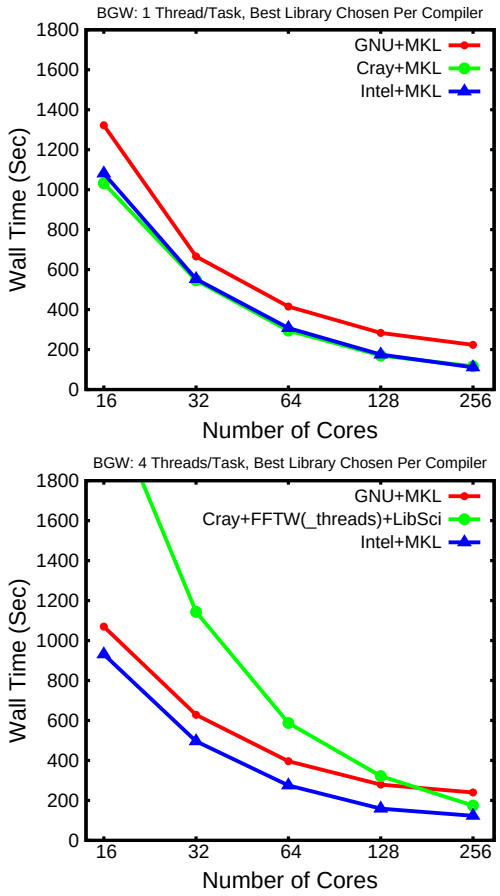

Figure 1. Summary of compiler performance for BerkeleyGW. See text for discussion and explanation of labels. The x-axis denotes the number of cores used via a combination of MPI tasks-OpenMP threads.

library is the best choice for each compiler. The combination of the Cray compiler + MKL actually outperforms the Intel + MKL combination when using a single thread. However, as we discuss more below, our Cray compiler + MKL build attempts yielded poor performance when multiple threads are used. The (_threads) label in the figure refers to the use of the provided libfftw3_threads library. The GNU compiler results have an approximately 100 second IO overhead when compared to the Intel and Cray results that we were not able eliminate - as shown in Fig. 2. This IO overhead occurred for all libraries used. This overhead is illustrated by the off-set present in Fig. 1. Thus, the Intel compiler + MKL combination is the only compiler and Library combination that performs well across the different number of MPI tasks and OpenMP threads that we considered.

In order to more fairly compare the FFT and Linear Algebra library performance it is useful to compare the library performance for a single compiler. We were only able to successfully compile and run our benchmark with all the available libraries for the case of the GNU compiler.

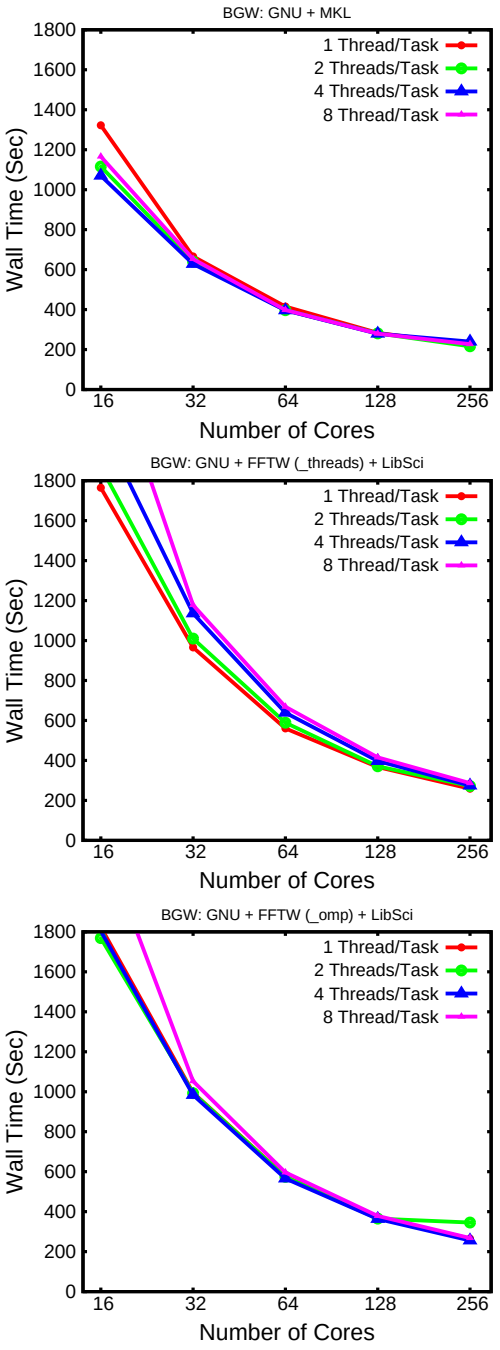

Figure 2. Wall-time vs cores for BerkeleyGW built with the GNU compiler with several different libraries. See text for discussion and an explanation of labels. The x-axis denotes the number of cores used via a combination of MPI tasks-OpenMP threads.

The results are summarized in Fig. 2.

It is seen in Fig. 2 that MKL consistently outperforms the FFTW + LibSci combinations. The label (_threads) in the figure refers to use of the provided libfftw3_threads library while (_omp) refers to the use of our manually

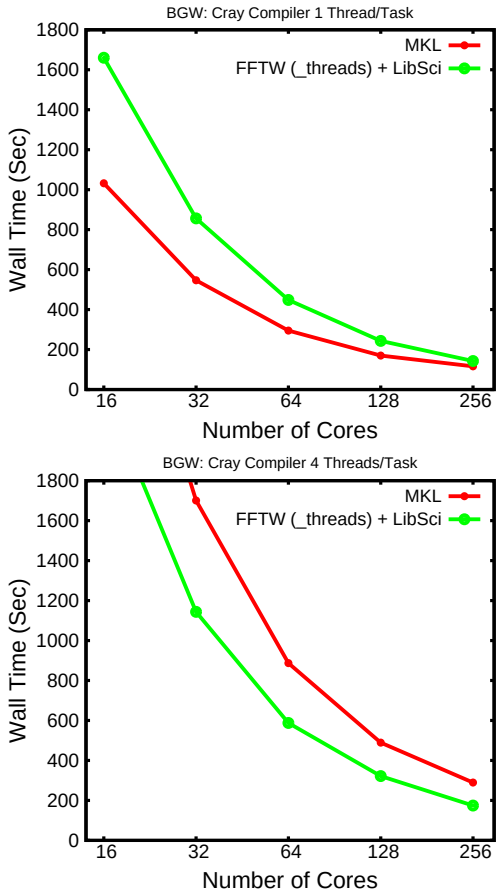

Figure 3. Wall-time vs cores for BerkeleyGW built with the Cray compiler with several different libraries. The x-axis denotes the number of cores used via a combination of MPI tasks-OpenMP threads.

compiled libfftw3_omp library. The plots show there is a walltime reduction for using multiple OpenMP threads per MPI Task (for a fixed number of cores) with the MKL build but a significant overhead when using the libfftw3_threads library. This could potentially be explained by the thread implementation in libfftw_threads library not performing well when combined with explicit OpenMP in the code, seemingly even when the threaded regions are distinct from each other. The use of the libfftw3_omp library, which was built for each compiler individually, and therefore contains the same OpenMP implementation as the linear algebra libraries and the rest of the code, appears to mitigate this conflict.

Figure 3 shows a summary of the BerkeleyGW results for the Cray compiler. In this case, we show results only for Cray's provided libfftw3_threads and for MKL.

From Fig. 3, we see that neither the threaded FFTW library, nor MKL, perform well with multiple threads with the Cray compiler in our tests. This might again be attributed to conflicts arising from multiple OpenMP implementations when other OpenMP regions exist in the code. The Cray compiler + MKL library combination proved to be particularly problematic. It should be noted that Intel does not provide a Cray compiler specific MKL library. For the numbers shown, we linked against the threaded MKL libraries intended for use with the GNU compiler. This may help explain the particularly poor performance in the Cray + MKL combination when multiple threads are used. One may use the sequential MKL libraries, but due to BerkeleyGW's heavy use of threaded libraries, the result is also poor. We additionally tried using MKL with libiomp5 (while removing the cce libomp.a from the build link line) without better success. We were unable to find an MKL library fully compatible with the Cray compiler or a suitable workaround, which limited the applicability of the Cray compiler for BerkeleyGW, since MKL outperforms FFTW and LibSci substantially. If a workaround could be found, the Cray compiler + MKL option could potentially outperform Intel+MKL.

We next consider the performance of the various libraries with the Intel compiler for BerkeleyGW. As of the time of writing, Cray has not released a version of LibSci compatible with the Intel compiler; so we limited our study to the comparison of the performance of BerkeleyGW using the MKL library for linear algebra while varying the FFT libraries.

From Fig. 4, we see that the code with MKL FFTs generally outperforms the code with FFTW. This is particularly evident with 4 threads, where, once again, the provided libfftw3_threads library performs poorly. While the manually compiled libfftw3_omp library does not gain significant overhead with the use of threads, it is still generally outperformed by the MKL library.

In order to provide a clearer picture of library performance within BerkeleyGW, Fig 5 shows the wall-time spent in FFT routines and ZGEMM for the various libraries with a single OpenMP thread per MPI task. The results shown are from BerkeleyGW compiled with the GNU compiler. However, the library timings are expected to be insensitive to the compiler. Because the performance of libfftw3_threads and libfftw_omp is nearly identical for a single thread, the FFTW curves have been combined.

The MKL FFTW interface outperform the FFTW3 interface across all core counts and the MKL ZGEMM outperforms the LibSci ZGEMM across all core counts. The latter difference is very significant leading to a reduction in ZGEMM walltime by nearly a factor of 2.

We are able to reproduce the MKL ZGEMM advantage over LibSci in a standalone ZGEMM test code. The same advantage is not present in a standalone DGEMM example, however, where the difference between the two libraries is within a few percent. This suggest that the ZGEMM performance gap is something Cray will likely close in a future release of LibSci.

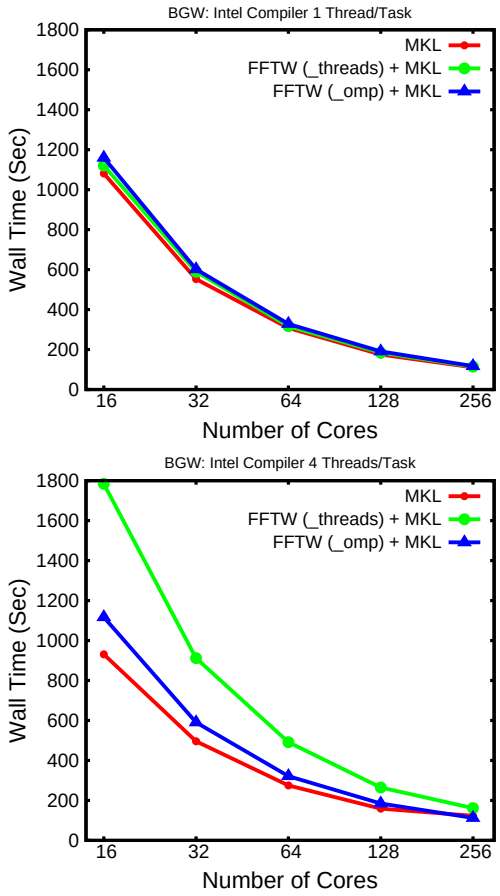

Figure 4. Wall-time vs cores for BerkeleyGW built with the Intel compiler with several different libraries. The x-axis denotes the number of cores used via a combination of MPI tasks-OpenMP threads.

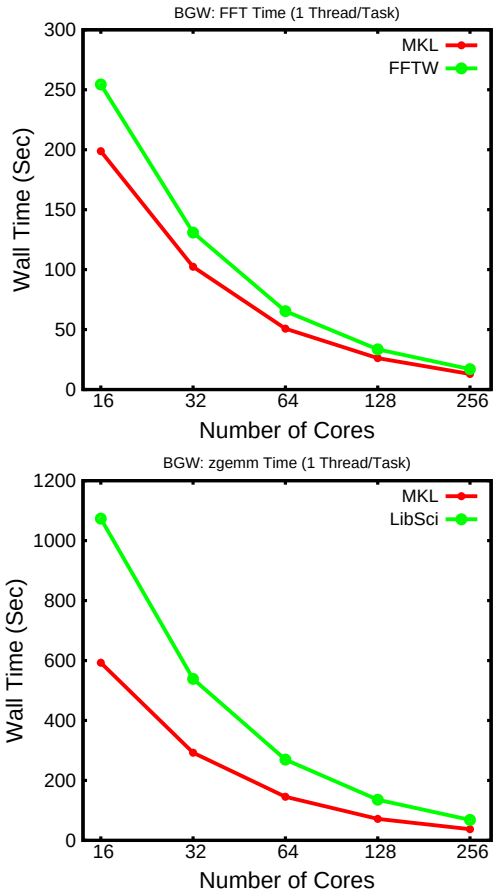

Figure 5. FFT and ZGEMM library performance within BerkeleyGW when compiled with the GNU compiler. Runs were computed with a single OpenMP thread per MPI-task.

B. Quantum ESPRESSO

Quantum ESPRESSO was tested with all of GNU, Intel and Cray compilers and FFTW, LibSci and MKL. A summary of the overall performance of the code on our example benchmark calculation is shown in Fig. 6.

As seen in the figure, the Intel Compiler + MKL library and GNU Compiler + MKL library result in the best overall combinations for both 1 and 4 OpenMP threads per MPI task. The combination of the Cray compiler + MKL slightly outperforms the other combinations when using a single thread. However, as was the case in BerkeleyGW, the Cray compiler + MKL again yielded less performance when multiple threads were used, again potentially attributable to conflicting OpenMP implementations (see this discussion in the BerkeleyGW section). As above, the (_threads) label refers to the use of the provided libfftw3_threads.

We note that our benchmark example is dominated by the FFT step. In order to more fairly compare the library performance (in this case predominately FFT) it is useful to compare the library performance for a single compiler. The results for the GNU compiler are summarized in Fig. 7.

It is seen in Fig. 7 that MKL consistently outperforms the FFTW + LibSci library combinations (in this case dominated by FFTW time). The label (_threads) in the figure again refers to use of the provided libfftw3_threads library while (_omp) refers to the use of our manually compiled libfftw3_omp library. The plots show there is significant overhead when using the libfftw3_threads library. Once again, this is potentially attributed to that the thread implementation in the libfftw_threads library not performing well when combined with explicit OpenMP in the code, even when the threaded regions are distinct from each other. The use of libfftw3_omp library, which is built with the same OpenMP implementation as the linear algebra libraries and the rest of the code, mitigates this conflict.

Like BerkeleyGW, therefore, in Quantum ESPRESSO, MKL outperforms competing libraries. Our inability to find an MKL library fully suitable for the Cray compiler (and the corresponding poor multi-threaded performance in Cray+MKL builds using the GNU MKL libs) makes this

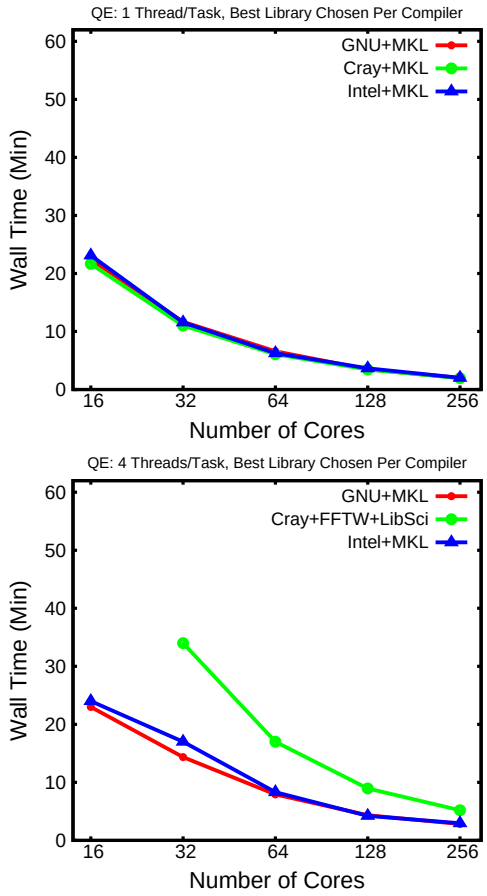

Figure 6. Summary of compiler and library performance for Quantum ESPRESSO. See text for discussion and explanation of labels. The x-axis denotes the number of cores used via a combination of MPI tasks-OpenMP threads.

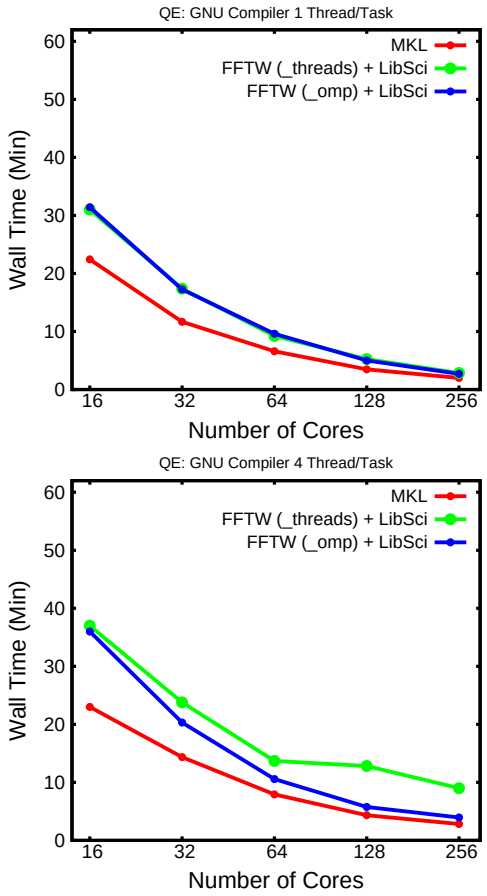

Figure 7. Wall-time vs cores for Quantum ESPRESSO built with the GNU compiler with several different libraries. See text for discussion and an explanation of labels. The x-axis denotes the number of cores used via a combination of MPI tasks-OpenMP threads.

combination un-optimal for Quantum ESPRESSO. Both the combination of Intel + MKL and GNU + MKL perform optimally in this case. Once again, if a suitable workaround could be found for multi-threaded Cray + MKL builds, the Cray compiler would be an optimal choice as well.

C. VASP

VASP was tested with the Intel and Cray compilers as well as FFTW, MKL FFTs and an internal FFT library denoted ”FURTH”. A summary of the overall performance of the code on our example benchmark calculations is shown in Fig. 8. The internal ”FURTH” fft library performed the worst, and is excluded from the figure for simplicity.

The figure shows different combinations of compiler, linear algebra library and FFT library. As in the case for BerkeleyGW and Quantum ESPRESSO above, we see that the best compiler and library combination was the Intel compiler with MKL used for both linear algebra and FFTs. Once again, MKL proved to provide the most performant FFT libs. In the case of VASP, we were unable to achieve a working executable with the Cray+MKL linear algebra combination.

D. LAMMPS

LAMMPS was tested with all of GNU, Intel and Cray compilers. A summary of the overall performance of the code on our example benchmark calculations is shown in Fig. 9. Since the LAMMPs examples described above do not make significant use of math libraries (the Rodo example does utilize FFTW, but the fraction of time spent in FFTW is a small fraction of the total runtime), we did not perform an extensive analysis of library performance in LAMMPS.

In all the example cases, the Intel compiler had the best performance, closely followed by the GNU compiler.

E. NWCHEM

NWCHEM was tested with the GNU, Intel compilers, compiled with the long integer MKL library. A summary of the overall performance of the code on our example benchmark calculations is shown in Fig. 10.

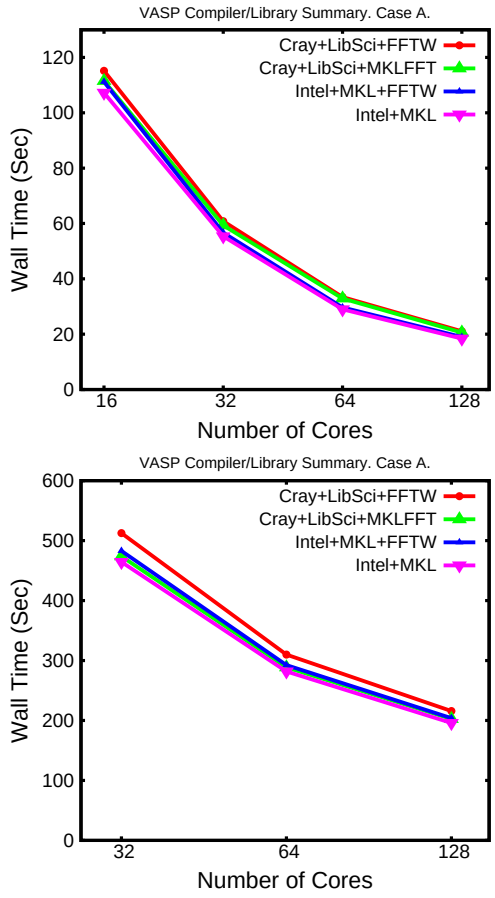

Figure 8. Summary of compiler and library performance for VASP examples. See text for discussion and explanation of labels.

We were unable to produce a NWChem executable using the Cray compilers. The figure shows that the Intel compiler has a small performance advantage over the GNU compiler for NWChem.

F. NAMD

NAMD was tested with the Intel and GNU compilers as well as FFTW. A summary of the overall performance of the code on our example benchmark calculations is shown in Fig. 11.

From the figure, we once again see that the Intel compiler is again provided this highest performing compilation. The NAMD is benchmark is overall less sensitive to FFT libraries, so FFT tests are not shown in the figure for simplicity.

V. Comparisons to Hopper

In Fig. 12, we compare the best performance (per core) of BerkeleyGW and Quantum ESPRESSO on the Cray XE6 system, Hopper, to the XC30 system, Edison. As with the above results, no hyper-threading has been used on Edison.

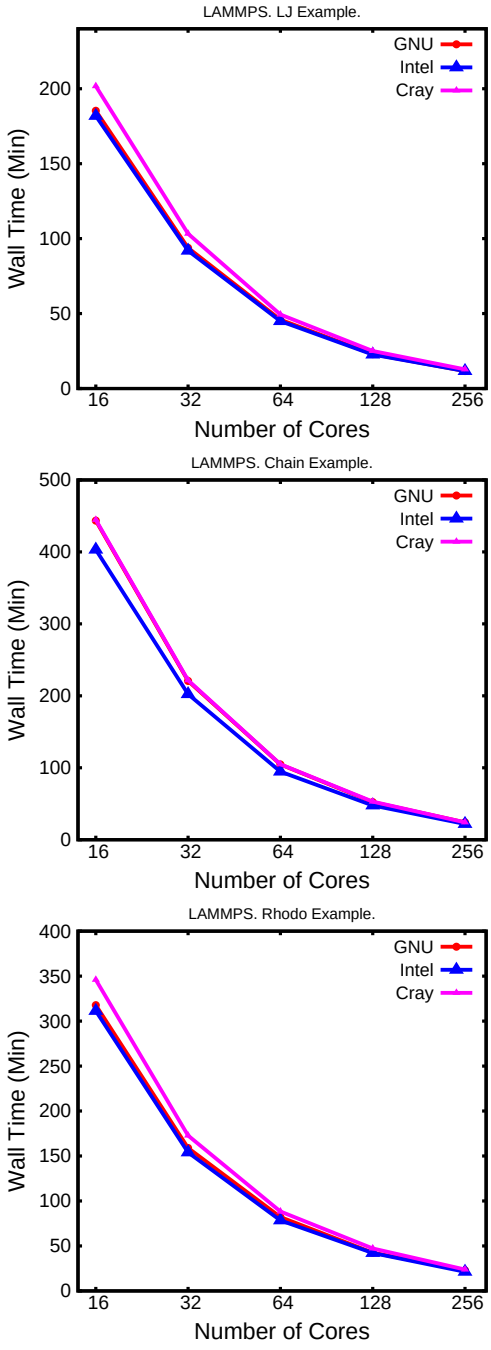

Figure 9. Summary of compiler and library performance for LAMMPS LJ, Chain and Rodo example. See text for discussion and explanation of labels.

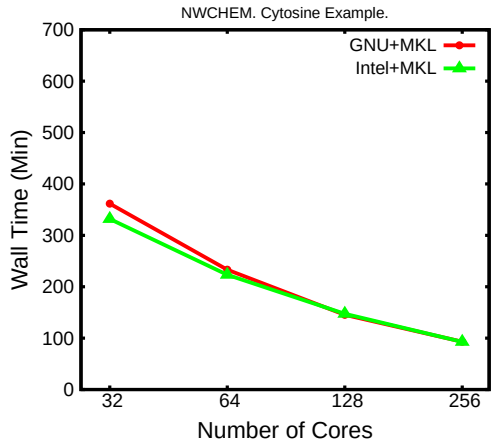

Figure 10. Summary of compiler and library performance for the NWCHEM example. See text for discussion and explanation of labels.

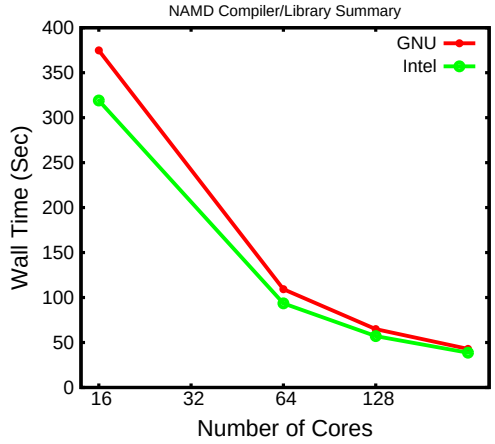

Figure 11. Summary of compiler and library performance for the NAMD example. See text for discussion and explanation of labels.

Edison significantly outperforms Hopper on a core by core comparison.

VI. Conclusion

We tested the performance of the 6 top material science and chemistry codes at NERSC on the Edison Cray XC30 system as function of the compiler and libraries used at build time. We found small but significant performance differences from different compilers for all codes. Even more significantly, we discovered large variations in performance (particularly in multithreaded cases) based on FFT and linear algebra library used. In particular, we found that MKL outperforms FFTW (both the provided libfftw3_threads and the custom built libfftw3_omp) and outperforms LibSci. Intel was the best overall compiler for the studied codes. This is in large part due to library support and compilation success rate. The Cray compiler coupled with MKL provided

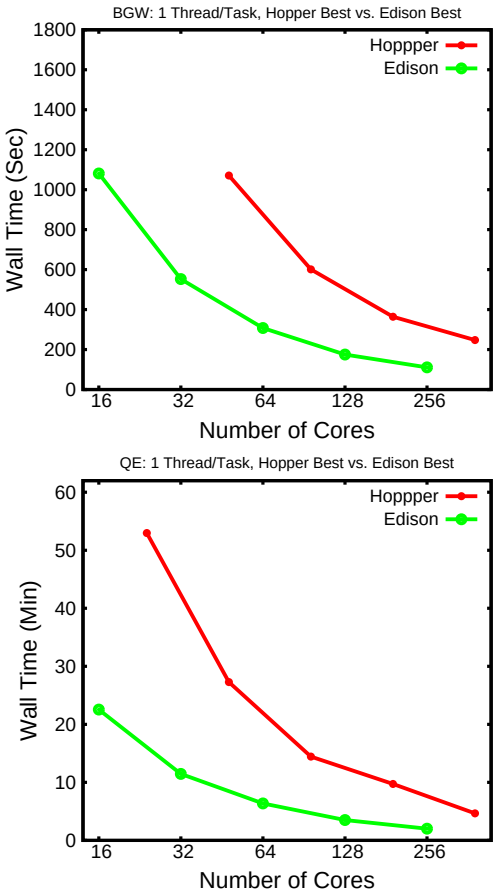

Figure 12. Walltime comparison of the application benchmarks between Hopper and Edison.

optimal sequential performance, but we failed to achieve performant builds with the threaded MKL library. If a suitable work-around could be found, the Cray compiler could yield optimal or near optimal builds for many of the codes.

Acknowledgment

The authors would like to thank Michael Stewart and Brian Austin at NERSC for their useful advice in completing this work.

Both authors were supported by the ASCR Office in the DOE, Office of Science, under contract number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy.

Support for BerkeleyGW profiling and OpenMP optimization provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences. Grant

Number DE-FG02-12ER46878 and under contract number DE-AC02-05CH11231.

REFERENCES

[1] http://www.nersc.gov/users/computational-systems/edison/

[2] Bowling, Megan, Zhengi Zhao, and Jack Deslippe. ”The Effects of Compiler Optimizations on Materials Science and Chemistry Applications at NERSC.” Optimization 4: 2.

[3] Deslippe, Jack, Georgy Samsonidze, David A. Strubbe, Manish Jain, Marvin L. Cohen, and Steven G. Louie. ”BerkeleyGW: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures.” Computer Physics Communications 183, no. 6 (2012): 1269-1289.

[4] Giannozzi, Paolo, Stefano Baroni, Nicola Bonini, Matteo Calandra, Roberto Car, Carlo Cavazzoni, Davide Ceresoli et al. ”QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials.” Journal of Physics: Condensed Matter 21, no. 39 (2009): 395502.

[5] G. Kresse and J. Furthmller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. Phys. Rev. B, 54:11169, 1996. http://www.vasp.at/

[6] http://lammps.sandia.gov/

[7] http://www.ks.uiuc.edu/Research/namd/

[8] http://www.nwchem-sw.org/

[9] http://www.nersc.gov/users/computational-systems/edison/performance-and-optimization/compiler-comparisons/

[10] http://www.fftw.org

[11] http://software.intel.com/en-us/intel-mkl