Evaluating and Optimizing the NERSC Workload on Knights Landing

Taylor Barnes, Brandon Cook, Jack Deslippe*, Douglas Doerfler, Brian Friesen, Yun (Helen) He, Thorsten Kurth, Tuomas Koskela, Mathieu Lobet, Tareq Malas, Leonid Oliker, Andrey Ovsyannikov, Abhinav Sarje, Jean-Luc Vay, Henri Vincenti, Samuel Williams
Lawrence Berkeley National Lab
*Corresponding Author

Pierre Carrier, Nathan Wichmann, Marcus Wagner
Cray Inc.

Paul Kent
Oak Ridge National Lab

Christopher Kerr
NCAR Consultant

John Dennis
National Center for Atmospheric Research

***Abstract*—NERSC has partnered with 20 representative application teams to evaluate performance on the Xeon-Phi Knights Landing architecture and develop an application-optimization strategy for the greater NERSC workload on the recently installed Cori system. In this article, we present early case studies and summarized results from a subset of the 20 applications highlighting the impact of important architecture differences between the Xeon-Phi and traditional Xeon processors. We summarize the status of the applications and describe the greater optimization strategy that has formed.**

I. Introduction

NERSC [1] is the production HPC facility for the U.S. DOE Office of Science. The center supports over 6000 users with more than 600 unique applications in a wide variety of science domains. HPC systems deployed at NERSC must support a diverse workload and broad user base but also need to satisfy an increasing demand for cycles from this community as spelled out in the requirements studies of the various science domains. [2] In addition to meeting the science demand, NERSC is transitioning to an energy efficient data center and requires high performing, energy efficient architectures. This transition has recently begun at NERSC with the installation of the Cori system, a Cray XC40 system powered by 9300+ Intel Xeon-Phi "Knights Landing" (KNL) processors added to an existing Cori phase 1 system powered by 1500+ Xeon (Haswell) processors.

In order to prepare user applications for Cori's KNL processors, NERSC has developed an "application-readiness" strategy involving both in-depth work with strategic application partners as well as an extensive training programming for the greater NERSC community.

The application-readiness effort is engaging 20 representative application teams (Table I) across significant domain areas and algorithmic paradigms to develop optimization case studies and relevant expertise that can then be used to develop an optimization strategy for general applications at NERSC. Towards this end, NERSC has established the NERSC Exascale Science Application Program (NESAP), a collaboration of NERSC staff along with experts at Cray and Intel, and the representative application teams with the goal of porting and optimizing the selected applications to the KNL architecture.

For the purposes of testing and optimizing performance on KNL with access to a small number of KNLs, application teams put together single-node runs that were representative of projected science problems on the full Cori system. In some cases, representative kernels were extracted to match the single-node work of the large science runs at scale.

In this article, we discuss the lessons learned from optimizing a number of these representative applications and kernels and discuss the impact of key KNL hardware features (including missing features). Finally, we will discuss how the application readiness process has informed the creation of an overall optimization strategy at NERSC targeting the broader user community.

II. Novel Features of Knights-Landing Relevant to Applications

For the last couple decades, NERSC has fielded massively parallel distributed memory systems powered by traditional x86 processors. The previously largest NERSC system, Edison, is powered by over 5500 dual Xeon (Ivy-Bridge) nodes with 12 cores per socket. The majority of CPU cycles on Edison are utilized in applications whose parallelism is expressed by MPI, with a growing minority of applications additionally expressing on-node parallelism via OpenMP. A small but growing number of applications are also written using a PGAS and/or task based programming model.

In comparison to Edison, the Cori KNL based system contains a number of important architectural differences that require attention from application developers. We list some of the most important hardware features below:

1. Many cores - The Xeon-Phi 7250 processors powering the Cori system contain 68 physical cores with 4 hardware threads per core, for a total of 272 hardware threads per processor. This is nearly 3 times the number of cores per node as Edison (24) and close to 6 times the number of hardware threads. These cores have an AVX frequency (1.2 GHz) that is half that of Edison's "Ivy-Bridge" processor (2.4 GHz).

TABLE I
NESAP TIER 1 AND 2 APPLICATIONS

Application	Science Area	Algorithm
Boxlib	Multiple	AMR
EBChombo	Multiple	AMR
CESM	Climate	Grid
ACME	Climate	Grid
MPAS-O	Ocean	Grid
Gromacs	Chemistry / Biology	MD
Meraculous	Genomics	Assembly
NWChem	Chemistry	PW DFT
PARSEC	Material Sci.	RS DFT
Quantum ESPRESSO	Material Sci.	PW DFT
BerkeleyGW	Material Sci.	MBPT
EMGeo	Geosciences	Krylov Solver
XGC1	Fusion	PIC
WARP	Accelerators	PIC
M3D	Fusion	CD/PIC
HACC	Astrophysics	N-Body
MILC	HEP	QCD
Chroma	Nuclear Physics	QCD
DWF	HEP	QCD
MFDN	Nuclear Physics	Sparse LA

2. Configurable on-chip interconnect - The cores are interconnected in a 2D mesh, with each mesh point comprising two cores (a tile) sharing an L2 cache. The mesh can be configured in a number of NUMA configurations. For this study, we concentrate on "quad mode", where the cores and DDR4 are logically contained in one NUMA domain, and the MCDRAM in a 2nd, but the distributed tag-directory for each memory controller is localized within four quadrants.

3. Wide vector units - Each core contains two 512-bit SIMD units (supporting the AVX512 ISA) with fused multiply-add (FMA) support enabling 32 double precision FLOPs per cycle, compared to Edison's 8 FLOPS per cycle.

4. On-package MCDRAM - The Knights Landing processor has 1 MB of L2 cache per tile (shared between two compute cores) but lacks a shared L3 cache. On the other hand, it contains 16 GB of on-package MCDRAM with a bandwidth of 450 GB/s as measured by STREAM (compared to 85 GB/s from the DDR4 memory subsystem). The MCDRAM can be configured as addressable memory, as a direct mapped cache or split between these configurations.

We discuss in the following sections a number of optimization case studies targeting the above hardware features and summarize key results across applications in the final section.

Unless otherwise noted, our KNL numbers were collected on single Xeon-Phi 7210 processors with a slightly lower AVX frequency of 1.1 GHz and 64 cores rather than 68 on the 7250 part on Cori. Our comparisons were made against single-node runs on the Cori Phase 1 system powered by two 16 core Xeon E5-2698 (Haswell) processors clocked at 2.3 GHz.

III. CASE STUDY 1 - QUANTUM ESPRESSO

Quantum ESPRESSO (QE) is a suite for performing plane-wave density functional theory (DFT) simulations [3], [4] of systems in materials science and chemistry [5]. We focus, here, on the performance of calculations employing hybrid functionals, which are often important for the accurate calculation of charge distributions [6], [7], band gaps [6], [8], [9], polarizabilities [10], and structural properties [8], [11]. Such calculations are dominated by the cost of computing the action of the exchange operator, $\hat{K}$, on the occupied electron orbitals, $\{\psi_i\}$ [12], [13], [14]:

$$(\hat{K}\psi_i)(\mathbf{r}) = -\sum_{j=1}^{n_{\text{occ}}} \psi_j(\mathbf{r}) \int \frac{\psi_j(\mathbf{r}')\psi_i(\mathbf{r}')}{|\mathbf{r}' - \mathbf{r}|} d\mathbf{r}'(\mathbf{r}) \qquad (1)$$

where n_{occ} is the number of occupied bands. Eq. 1 must be calculated for each occupied orbital, and thus requires the calculation of a total of n_{occ}^2 integrals. In QE, these integrals are calculated by subroutine `vexx`, through the use of FFTs, as shown in Algorithm 1. QE facilitates two primary approaches for parallelizing the evaluation of Eq. 1: (1) plane-wave parallelization, in which the work of computing each individual FFT is distributed across many processes, and (2) band parallelization, in which multiple FFTs are simultaneously computed by different sets of processes. In previous work [15] within the NESAP effort, we have expanded upon the existing band parallelization through the introduction of a "band pair" parallelization algorithm, along with other enhancements that enable optimal parallelization of the calculation of exact exchange without negatively impacting the parallelization of other regions of the code. These modifications have lead to significant improvements in the strong scaling efficiency of the code and increase the density of node level work.

Algorithm 1 Evaluation of Eq. 1

```
1:  procedure VEXX
2:      ...
3:      for i in 1:n_occ do
4:          ...
5:          c(g) = FFT[1/|r' − r|]
6:          for j in 1:n_occ do
7:              ρ_ij(r) = ψ_i(r)ψ_j(r)
8:              ρ_ij(g) = FFT[ρ_ij(r)]
9:              v_ij(g) = c(g)ρ_ij(g)
10:             v_ij(r) = FFT^{-1}[v_ij(g)]
11:             (Kψ_i)(r) += ψ_j(r)v_ij(r)
```

We focus here on our efforts to improve the single-node performance of Algorithm 1 on KNL, which is primarily governed by the efficiency of the hybrid MPI-OpenMP plane-wave parallelization implementation in QE. The many-core nature of the KNL architecture renders on-node OpenMP threading an appealing option for the optimal use of memory resources; however, efficiently scaling to large numbers of threads is challenging, as shown by the red curve in Fig. 1. The difficulty of obtaining good thread scaling in Algorithm 1 derives from the structure of the inner loop, which involves three "vector-multiplication"-like operations (used to compute $\rho_{ij}(\mathbf{r})$, $v_{ij}(\mathbf{g})$, and $(\hat{K}\psi_i)(\mathbf{r})$), interleaved by FFTs. Because of this structure, each vector-multiplication is computed within an individual OpenMP region, and several OpenMP fork and

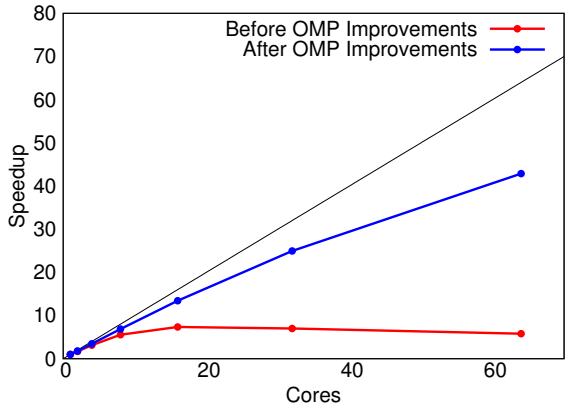

Fig. 1. Improvements to the thread scaling of Algorithm 1 on KNL. Each calculation is performed on system of 64 water molecules using a single KNL node.

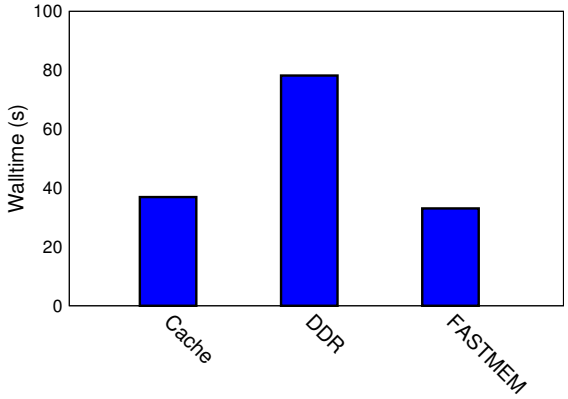

Fig. 2. Performance of Algorithm 1 on a single KNL node using several different memory modes. From left to right: cache mode, flat mode using only DDR, and flat mode with explicit MCDRAM use via `FASTMEM` directives.

join operations are required for each iteration of the inner loop. We reduce the amount of OpenMP overhead by changing the loop structure such that each OpenMP region encompasses a loop over bands. In the case of the calculation of $\rho_{ij}(\mathbf{r})$, this corresponds, in simplified terms, to:

```
!$omp parallel do ...
DO j=1, n_occ
   DO ir = 1, nr
      rho(ir,j)=psi(ir,i)*psi(ir,j)
   ENDDO
ENDDO
!$omp end parallel do
```

One might be tempted to add a `collapse(2)` clause to the OpenMP directive above. However, we noticed that this reduced the performance by nearly 2x. The explanation is that vector loads/stores are replaced by gather/scatter operations because the compiler can no longer guarantee stride 1 access.

In addition, we bundle the computation of the FFTs such that multiple FFTs are performed by each library call. This reduces the number of OpenMP fork and join operations and gives a factor $\sim$3.3$\times$ speedup compared to sequential execution of single multi-threaded FFTs - which may be additionally attributed to the ability of the library to optimize cache-reuse among the many FFTs. These modifications lead to substantial improvements in the threading efficiency of QE, as shown in Fig. 1

An additional concern when running on KNL is the optimal choice of memory mode. As shown in Fig. 2, running cache mode is approximately a factor of two faster than running exclusively out of DDR in flat mode - though variability between runs was observed in cache mode attributed to the fact that the working set size greatly exceeded 16GB leading to potential cache-conflicts in the direct-mapped MCDRAM cache. We used `FASTMEM` directives to place the arrays containing ψ_i, ρ_{ij}, v_{ij}, and $(\hat{K}\psi_i)(\mathbf{r})$ into MCDRAM. This approach significantly reduces the walltimes associated with the vector multiplication operations in Algorithm vexx, and

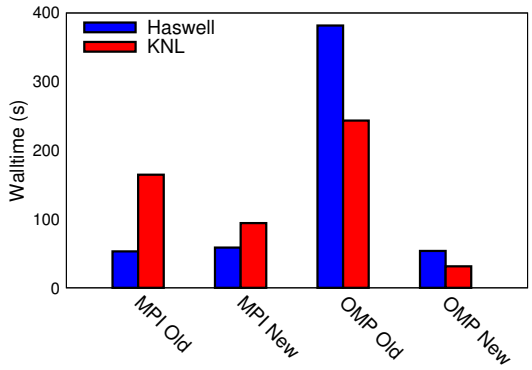

Fig. 3. Single-node performance of Algorithm 1 on KNL and Haswell. The MPI results correspond to running in pure MPI mode, while the OMP results correspond to running with 1 MPI task and 16 OpenMP threads per 16 cores. The "Old" results are obtained using a version of QE immediately prior to the addition of the improvements described in this paper.

enables flat mode to outperform cache mode, as shown in Fig. 2 by 10-15% without variability.

Fig. 3 compares the performance of Algorithm 1 between KNL and Haswell. On both Haswell and KNL, pure MPI mode was originally found to be more efficient than mixed MPI-OpenMP parallelization. However, implementation modifications described in this section are observed to enable a mixed MPI-OpenMP approach to outperform pure MPI parallelization. The net result of our enhancements is a 2.9$\times$ speedup in the best single-node performance observed on KNL. Furthermore, whereas prior to optimization, a single-node KNL calculation was found to be 1.6$\times$ slower than the corresponding single-node Haswell calculation, it is now found to be 1.8$\times$ faster.

IV. Case Study 2 - Nyx/BoxLib

BoxLib is a software framework for developing parallel, block-structured, adaptive mesh refinement (AMR) codes. Applications of BoxLib include compressible hypersonic flows for cosmology [18], reactive flows for radiating systems

such as supernovae [17], low Mach number flows for stellar convection [23] terrestrial combustion [26], and fluctuating hydrodynamics [24], porous media flows [25], and others. In most of these applications, the physical properties of the simulations are expressed in FORTRAN kernels, while BoxLib itself handles domain decomposition, AMR, memory management, parallelization (both MPI and OpenMP), boundary condition information, inter-node communication, and disk I/O. It also provides multigrid linear solvers on both uniform and adaptively refined meshes. The parallelism in BoxLib codes is hierarchical; inter-node parallelism is expressed via MPI by decomposing the total problem domain into boxes of arbitrary rectangular shape, and distributing them among MPI processes. The intra-node parallelism takes the form of OpenMP directives, spawning multiple threads to work on different regions of each box. In this report, we focus on the cosmology code Nyx [18] as a proxy for other BoxLib applications.

A few kernels dominate code execution time in Nyx. These include the piecewise-parabolic method (PPM) reconstruction of the baryon state data [21], the analytical Riemann solver [20], Gauss-Seidel red-black smoothing during the multigrid solve for self-gravity [19], the equation of state (EOS) [22], and the radiative heating and cooling terms [22]. The first three kernels are "horizontal" operations, in that they require information from neighboring cells, while the latter two are "vertical" (point-wise) operations, requiring no neighboring data.

Horizontal operations involve a large amount of data movement (e.g., stencils) and low arithmetic intensity; consequently they are ideal candidates for optimization via cache reuse. In the initial implementation of OpenMP in Nyx[1], the general approach was to decorate the triply-nested $\{x, y, z\}$ loops in these kernels with `!omp parallel do ... collapse(2)` directives. While simple and non-invasive to implement, this approach led to frequent last-level cache misses, and thus high memory bandwidth. As a result, the strong scaling thread performance saturated at a small number of threads ($\sim$5) per MPI process, due to memory bandwidth saturation. To improve cache reuse and thus thread concurrency, we implemented loop tiling [27], splitting large boxes into smaller tiles, and distributing them among threads. Specifically, an MPI process builds a list of all tiles spanning all boxes which it owns, and then in a parallel region the OpenMP threads iterate over the list of tiles using static chunk size scheduling to determine the number of tiles on which each thread must operate. This list replaces the triply-nested loops over $\{x, y, z\}$, and thus no `collapse` clause is necessary. The loop tiling approach improves data locality as well as thread load balance, and as a result, most horizontal operations in Nyx now strong scale effectively up to $\sim$64 threads on Xeon Phi.[2] In particular, we choose "pencil"-shaped tiles which

[1]Prior to that point, Nyx had been a purely MPI-parallelized application.

[2]Most Nyx kernels have low memory latency, so we see little benefit from using multiple hardware threads per core.

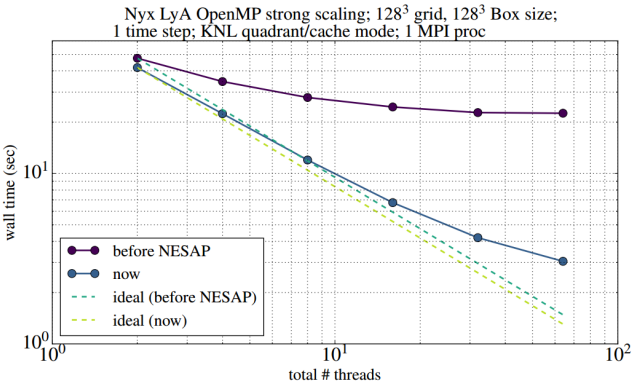

Fig. 4. Thread strong scaling in Nyx on Intel Xeon Phi 7210.

are long in the stride-1 dimension, in order to maximize the effectiveness of prefetching, and short in the other two dimensions, allowing us to fit the tiles into the relatively small last-level cache (1 MB shared between two cores) on Xeon Phi. A typical tile size is $(64, 4, 4)$, while a typical box size is $(64, 64, 64)$.

Optimizing the point-wise operations in Nyx is more challenging, as they benefit less from loop tiling and each kernel has unique performance characteristics. The equation of state, for example, computes the free electron density and the temperature in each cell via Newton-Raphson iteration. The data dependence of the Newton-Raphson algorithm prohibits vectorization. Instead, we have enjoyed success rewriting the algebraic expression of various functions to minimize the number of divides required.

Together, these optimization strategies have resulted in a large improvement in thread scaling on the Xeon Phi architecture. In Figure 4 we show the thread strong scaling on a single pre-production 7210 Xeon Phi processor. When using a single MPI process and 64 OpenMP threads, Nyx now runs $\sim 5\times$ faster on Xeon Phi than it did without the optimizations discussed above. We have found that, even with a large problem (256^3 problem domain, with a memory footprint of $\mathcal{O}(10)$ GB), the L2 cache miss rate is $< 1\%$.

As can be seen in summary Figures 9 and 10 these modification lead to speedups around $2\times$ on KNL. The code currently performs about 40% faster on a single Haswell node than a KNL node. Fig. 11 shows that despite the low L2 miss rate, the amount bytes transferred from DRAM (or MCDRAM) during execution is nearly $5\times$ greater on KNL, presumably due to the lack of an L3 cache.

V. Case Study 3 - CESM

The Community Earth System Model (CESM) [28] developed at the National Center for Atmospheric Research (NCAR) is a coupled earth-system model which consists of multiple-components models: atmosphere, ocean, sea-ice, land-ice, land, river-runoff, and coupler. The cost of the atmosphere components (dynamics and physics) typically dominate

the total run time (60%). We discuss two of the atmosphere components below.

A. MG2

The MG2 kernel is version 2 of the Morrison-Gettleman microphysics package [29], [30]. This component of the atmospheric physics was chosen as it is representative of the general code in the atmospheric physics packages and MG2 typically consumes about 10% of total CESM run time.

Previous studies of the MG2 kernel have shown to be compute bound with poor vectorization due to: short loop bounds O(10), dependent sequence of instructions, and complex branches in the loops. Heavy use is made of the mathematical intrinsics: POW, LOG, EXP, and GAMMA and lack of vectorization forces the use of scalar implementations of these intrinsics. The scalar performance of: POW, LOG, and EXP on KNL was significantly worse than on Haswell. Further, Intel GAMMA function was 2.6× slower on Haswell than the GAMMA function used in the code. MG2 was written with extensive use of elemental functions which were found to inhibit vectorization because of limitations in the compiler.

Below are the major optimization steps applied to MG2; the optimizations performed were shown to improve the performance on both Haswell and KNL [31], [32]:

1) Simplify expressions to minimize number of mathematical operations
2) Use the internal coded GAMMA function
3) Remove the elemental attribute from subroutines and explicitly define the loops in the routines
4) Replace declaration of assumed shaped arrays with explicitly defined loop declarations
5) Replace division with inversion of multiplication
6) Use aggressive compiler flags such as those to reduce numerical accuracy; used with caution
7) Use directives and profile to guide optimizations

Directives can be helpful if you understand why they are needed and were used after trying other code modifications. Use of the vectorization and optimization reports from the Intel compiler help guide the optimization procedure.

Fig. 5 shows the final MG2 performance improvement on a single Haswell node (64 MPI tasks) and a single KNL node (64 MPI tasks × 4 OpenMP threads per task). Speedup from the optimized code on Haswell and KNL are 1.75 and 1.92 respectively. However, single node KNL still does not outperform single node Haswell for this kernel.

B. HOMME

The High Order Methods Modeling Environment (HOMME) model is a atmospheric dynamic core available in CESM. HOMME uses a spectral element method to discretize horizontally and a finite difference approximation vertically[33]. A continuous Galerkin finite-element method[34] is used for the spectral element method. HOMME takes about 35% of the cycles in a typical CESM run.

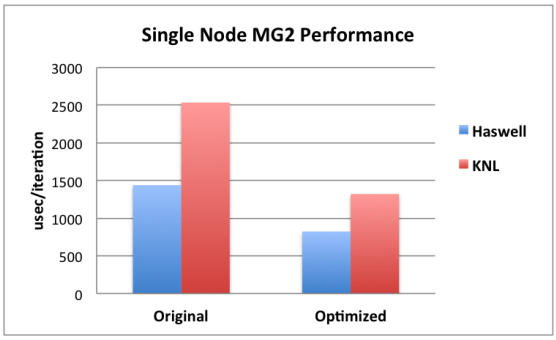

Fig. 5. MG2 performance improvement on Haswell and KNL.

HOMME is primarily a compute bound code, however several known regions of the code are memory bound. Some of the optimization steps performed for HOMME[35] are:

1) Thread memory copies in boundary exchange
2) Restructure data and loops for vectorization
3) Rewrite message passing library and implement specialized communications operations
4) Rearrange calculations to optimize cache reuse
5) Replace division with inversion of multiplication
6) Compile time specification of loop and array bounds

A major change made to HOMME was the redesign of the OpenMP threading scheme. In the original scheme, parallelization was performed over elements at a high-level and over loops at the lower-levels. The revised scheme uses the same high-level parallelization over elements. However, the lower-loop level parallelization has been replaced with a high-level parallelization in the tracer and vertical dimensions. The new scheme supports element, tracer, and vertical high-level parallelization simultaneously. The scheme significantly reduces the number of parallel regions in the code from over a hundred to four. As a consequence of the redesign, the code is much easier to maintain as the `SHARED` and `PRIVATE` OpenMP declarations do not have to be declared.

Fig. 6 shows the HOMME scaling performance on a single KNL node. Nested OpenMP is used in the optimized version. The best KNL time is achieved with 64 MPI tasks, 2 threads each for the element and tracer dimensions and 1 thread for the vertical dimension. While performance saturates with 64 cores and 1 hardware thread per core for the original version on KNL, the optimized version achieved max performance with up 64 cores and 4 hardware threads per core.

Fig. 9 and Fig. 10 shows the final HOMME performance improvement on a single Haswell node and a single KNL node.

C. Issues and Concerns

Performance studies reveal that the scalar version of the Intel mathematical functions: `LOG`, `EXP`, `PWR` were over 5 times slower on KNL than those on Haswell. This is an open issue with Intel under investigation.

Assumed shaped arrays declarations are used extensively in CESM. These declarations have been changed to explicitly

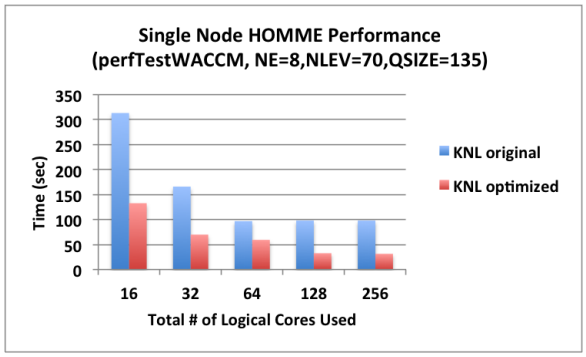

Fig. 6. HOMME scaling performance with original and optimized versions on a single KNL node.

define the array bounds and as a result it improves the speedup in the loops associated with the declarations. A significant performance improvement in HOMME would be achieved if the compiler were able to collapse and linearize the inner most loops. Intel are working on implementing the feature.

The use of the `OMP SIMD` directives were explored for performance portability. However, performance gain is only realized when explicitly providing the aligned list of variables to the directive. However, providing such a list is cumbersome.

The Cray CCE compiler performance was also investigated for MG2 and HOMME, and in many cases ran faster than with the Intel compiler. However more porting and verification work needs to be done to use CCE for the full CESM code.

VI. Case Study 4 - XGC1

The XGC1 code is a full distribution function global 5D gyrokinetic Particle-In-Cell (PIC) code for simulations of turbulent plasma phenomena in a magnetic fusion device. It is particularly well-suited for plasma edge simulations due to an unstructured mesh used in the Poisson equation solver that allows the simulation volume to encompass the magnetic separatrix, the Scrape-Off-Layer (SOL) and the edge pedestal. The main building blocks of the code are the particle pusher, the collision operator and the Poisson solver.

The particle pusher advances a large $\mathcal{O}(10^8)$ ensemble of marker particles in cylindrical coordinates in time by integrating the guiding-center equations of motion [36]. The equations are integrated with a standard RK4 algorithm. Electrons are pushed for $\mathcal{O}(50)$ time steps between field solves and collisions due to their high velocity compared to ions, this is referred to as electron sub-cycling. The non-linear collision operator [37] operates on the particle distribution function, a velocity space mesh. In order to operate on the velocity grid, the PDF from the marker particles is mapped to two dimensional velocity space grid, and the Coulomb collision information is mapped back to the marker particles after the collision operation has been completed. The velocity-grid operation is performed on each cell of a regular real space grid, but these operations can be performed independently, making it a good candidate for parallellization. The Poisson solve is performed using the PETSc library on an unstructured mesh in the R,z - plane and a regular field-line following mesh in the ϕ - direction. For this step, the charges of the marker particles are deposited onto the nodes of the unstructured mesh.

After recent optimizations to its collision kernel [38] and due to the sparsity of the calls to the Poisson solver, most of the computing time in XGC1 is spent in the electron push (pushe) kernel, where the current optimization efforts have been focused. The electron push consists of four main steps: $\vec{B}$-field interpolation, $\vec{E}$-field interpolation, the particle push, and the search for the new particle location; after each RK4 step, a new particle position is obtained and the corresponding unstructured mesh element has to be searched.

In previous work, parallelization with MPI+OpenMP has been implemented in all kernels of XGC1 and the full code scales well up to 10 000's of cores. However, the pushe kernel performance was limited by almost no automatic compiler vectorization. The $\vec{E}$ and $\vec{B}$ interpolation routines consist of loops with trip counts of 3 and 4, respectively, that are executed once per particle per RK4 step. Typically, the compiler would consider these loops too short for vectorization and produce serial code. To enable vectorization, the loops were refactored so that the innermost loop is over a block of particles, whose size can be tuned to the size of the vector register. The compiler has been instructed to vectorize the loops over particle blocks using `!$omp simd` directives that result in good vectorization efficiency. The limiting factor in the interpolation loops is retrieving data from the grid points closest to the particle position, a fundamentally random memory access pattern that results in gather/scatter instructions. We have ameliorated this somewhat by employing a sorting algorithm at the beginning of the sub-cycling that improves data memory locality. With sorting, the $\vec{E}$ and $\vec{B}$-field interpolation kernels gained 1.5× and 1.8× speedups, respectively. On Xeon Haswell processors, the XGC1 pushe kernel benefits from the fast L3 cache that is absent on Xeon Phi (See Fig. 11). Therefore, on Xeon Phi the amount of loads from DRAM is roughly 3× higher than on Haswell. The penalty for cache misses can be partially hidden by using hyper-threading, Figure 7 shows that the performance improves with hyper-threading on Xeon Phi while on Haswell it actually shows slightly worse performance. As a result of the optimizations, the pushe kernel has gained a speedup of 1.9× on Haswell and 1.7× on Xeon Phi. Node-to-node performance comparison, presented in Figure 7 shows that Haswell node performance is still roughly 25% better, attributed to LLC. Our present optimization strategy is to re-order the time step and particle loops to increase the reuse of grid data in the interpolation routines. Theoretically, a close to perfect reuse rate is achievable, since only roughly 1% of particle time steps result in movement across grid elements.

VII. Case Study 5 - WARP-PICSAR

The code WARP is an open-source PIC code dedicated to the modeling of charged particles and electromagnetic fields, including the dynamics of charged particle beams in

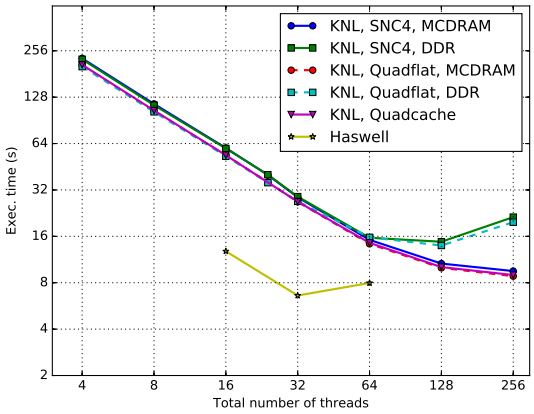

Fig. 7. On-node thread scaling of the XGC1 pushe kernel on different KNL memory modes. The code scales identically on all modes up to 1 thread per core. On multiple threads per core, the DDR memory bandwidth is saturated but MCDRAM provides a speedup.

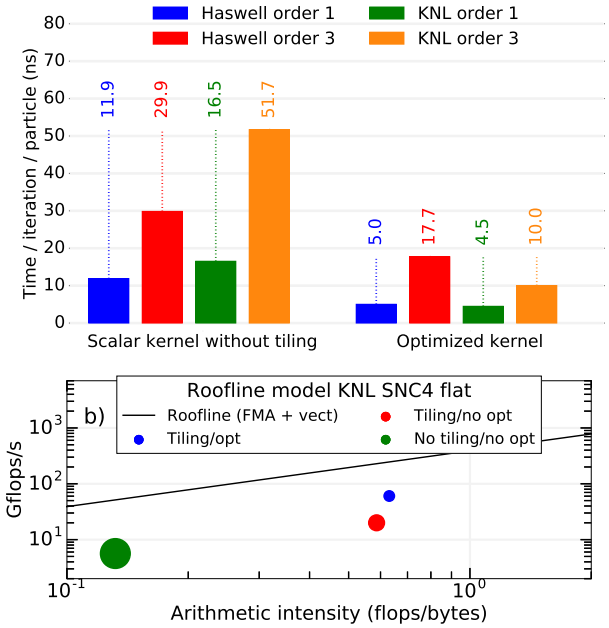

Fig. 8. (a) - PICSAR simulation times per iteration and per particle on a Cori phase 1 node and a single KNL for the non-optimized kernel (scalar subroutines without tiling) and the present optimized kernel with order 1 and order 3 interpolation shape factors. (b) - Roofline performance model applied to the PICSAR kernel with order 1 interpolation. The roofline is represented by the black line. The green marker represents the non-optimized version of the code (original kernel), the red marker the kernel with the tiling and the blue one the fully optimized code. Marker size is proportional to the simulation time.

particle accelerators, and the physics of high-intensity laser-matter interactions [39]. PICSAR is an open-source Particle-In-Cell FORTRAN+Python library designed to provide high-performance subroutines optimized for many-integrated core architectures [40], [41] that can be interfaced with WARP.

PICSAR follows the evolution of a collection of charged macro-particles that evolve self-consistently with their electro-magnetic fields. A macro-particle represents a group of real particles of the same kind with the same kinetic properties (speed and propagation direction). Each particle's property is stored in an aligned and contiguous 1D array for all particles (structure of arrays). Field structured grids are stored in 3D arrays for each component. A MPI domain decomposition (Cartesian topology in PICSAR) is usually used for the intra-node parallelism. The classical PIC loop contains 4 computational steps. 1) Field gathering: fields seen by the particles is interpolated from the grids. 2) Particle pusher: Particles are moved forward via the fields. This step is followed by the communication of the macro-particles between MPI domains. 3) Current/charge deposition: The generated current and charge is deposited on the grids. This step is followed by the communication of the guard-cells for the current grids. 4) Maxwell solver: The field grids are updated from the currents. This step contains the communication of the guard-cells for the field grids. Steps 1) and 3) constitutes interpolation processes between the grids and the particles and uses B-spline shape factors of given orders. The number of grid cells and vertexes involved in these interpolation processes and border guard-cells increase with the shape factor order.

Cartesian based PIC codes have a low flop/byte ratio that leads non-optimized algorithms to be highly memory-bound [41]. Large field and particle arrays cannot in cache in most simulations. Because two consecutive particles in memory can be localized at very distant physical positions, different portions of the field arrays are therefore constantly reloaded leading to poor cache reuse, especially during interpolation processes. In order to improve memory locality, cache-blocking of the field arrays has been achieved by dividing MPI domains into smaller subsets called tiles. Tile size is carefully chosen to let local field arrays fit in L2 cache on Haswell and KNL. OpenMP is used to parallelize over the tiles inside MPI regions. With more tiles than OpenMP threads, a natural load-balancing is achieved with the `GUIDED` or the `DYNAMIC` schedulers. Furthermore, a particle sorting algorithm has been tested to further improve memory locality. Because of its cost, the sorting is performed after every set number of iterations.

Tiling has created new particle exchanges between tiles in addition to exchanges between MPI ranks. This operation is done by transferring macro-particle properties between tile arrays all located in the same shared-memory space. A communication pattern by block using nested parallel regions has been developed to avoid memory races when several particles are transferred to the same tile. The tiling communication has been merged with the MPI communication preprocessing loop for further gain in performance.

The second phase of optimization focused on vectorization. In the original code, the main hotspots was the interpolation process whereas the particle pusher was automatically and efficiently vectorized over the particles. Thus, in the current/charge deposition, the particle loop has been modified

to remove dependencies and the data structure has been changed to align grid vertexes in memory enabling an efficient vectorization of the deposition process.

Performance results are shown in Fig. 8(a). For order 1 shape factor, all performed optimizations lead to a 2.4× speedup on Haswell node and 3.7× speedup on KNL in comparison with the scalar version of the code without tiling. Final times are similar between Haswell and KNL. At order 3 (preferred in production rns), 1.7× speedup on Haswell and 5× speedup on KNL are observed. KNL outperforms Haswell with a speedup of 1.6×.

The Roofline model [41], [42], [43], [44] applied to PICSAR is shown in Fig. 8(b). The tiling increases the flop/byte ratio as shown by the transition from the green marker (original code) to the red one. Vectorization and other optimization them mainly increases the flops/s leading to the blue marker (fully-optimized code).

The roofline model provides a useful way to visualize code performance and frame the optimization conversation with code teams. The approach used hear has been used in other cases [41] and is now included in NERSC's overall app-readiness strategy - as a way to review current code performance and discuss optimization avenues with code teams.

VIII. Summary of NESAP Application Performance

Application speedups obtained during the NESAP period are shown in Fig. 9 for representative problems or kernels on a single KNL and Haswell node. The speedups achieved range from slightly over 1x to well over 10x. In most cases, significant speedups were achieved on both KNL and Haswell. Code improvements targeting the many-core Xeon-Phi architecture nearly always end up in code-restructuring (e.g. for vectorization and improved data-locality) that essentially improve code performance on all architectures. In most cases, the impact of the performance improvements is higher on KNL than Haswell, however. This is because the fewer and faster cores on Haswell (with shared L3 cache) are more forgiving to common code issues: imperfect thread-scaling and vectorization for example.

An example of the trend is the WARP code, where tiling optimization targeting data reuse in the L2 caches has a more significant impact on KNL than Haswell where the L3 cache was being effectively used even before the tiling optimization. The BerkeleyGW application provides another scenario for this effect. In this case, the code is marginally CPU bound (an AI around 3) but the key kernel was found to be using scalar instructions. Restructuring the code to enable vectorization yields larger speedups on KNL than Haswell due primarily to the wider (512 bit vs 256 bit) vector units.

A notable exception to this trend is the Boxlib application. In this case, the speedups obtained are actually larger on Haswell than KNL. In this case, the application is memory bandwidth bound before the tiling optimizations described above. The application, being significantly more bandwidth starved on Haswell (which lacks MCDRAM), has more performance to gain on Xeon by tiling/blocking optimizations.

The speedups shown in Fig. 9 show the scope of the work performed in the NESAP program as well as the relative potential for speedups in typical science applications on KNL. However, the speedups, alone, don't provide a measurement of absolute performance on the KNL or of expected relative performance between runs of similar scale on the Cori Phase 1 (Haswell) nodes vs Cori Phase 2 (KNL) nodes. To address the latter issue, we compare the node level performance of the applications on single node runs on Haswell and KNL. We put off for the future a discussion of internode scaling and assume that, to first order, we can expect similar scaling as the interconnect for the two Cori phases is identical. The comparison is shown in Fig. 10.

The "speedups" on KNL relative to Haswell shown in Fig. 10 actually range from slowdowns of 50% to speedups of near 300% when comparing against optimized codes (blue bars) on Haswell. We also show, where possible, a comparison (orange bars) against the original un-optimized code on Haswell. In this case, we see speedups of up to 700% (or 8x).

When comparing optimized versions on both KNL and Haswell, we see that EMGeo, MILC, Chroma and MFDN (SPMM) have the largest gains. These applications all share a commonality - they are all known to be memory-bandwidth bound and achieving near roofline performance on both architectures. The ratio of performance on KNL vs Haswell closely matches that ratio of the memory bandwidths available on the nodes (400-450 GB/s vs approximately 120 GB/s).

MILC and Chroma are Quantum Chromodynamics (QCD) applications that achieve maximum data-reuse and vectorization on their 4D stencil computations through the use of the QPHIX library. As shown in Fig. 11, these codes demonstrate high speedups (3x) when running out of MCDRAM vs DDR on the KNL.

EMGeo and the MFDN SPMM kernel are both sparse matrix-vector or matrix-matrix multiply dominated with measured effective bandwidth near that of that stream benchmark on each platform. In these cases, the main optimization was performing solves on multiple right-hand sides (RHS) concurrently in order to reuse data in the sparse matrices. Though, the codes remain memory-bandwidth limited even after such optimizations. Again, Fig. 11 shows 3× or greater speedups running out of MCDRAM vs DDR on the KNL.

The case of MFDN is unique among these four applications in that the scientific use case does not fit within the 16GB of MCDRAM on node. In fact, the team anticipates using nearly all the memory on each KNL node. However, the team was able to identify important arrays, those representing the vectors which the sparse matrix operates on, to explicitly place into MCDRAM with appropriate FASTMEM directives. This approach allowed the team to outperform runs with the KNL booted with the MCDRAM configured as a cache.

Applications with higher arithmetic intensities, not fundamentally limited by memory bandwidth, that effectively use the AVX512 ISA are also able to see relatively higher performance on KNL vs Haswell. HACC and BerkeleyGW are examples of applications in this category with KNL speedups

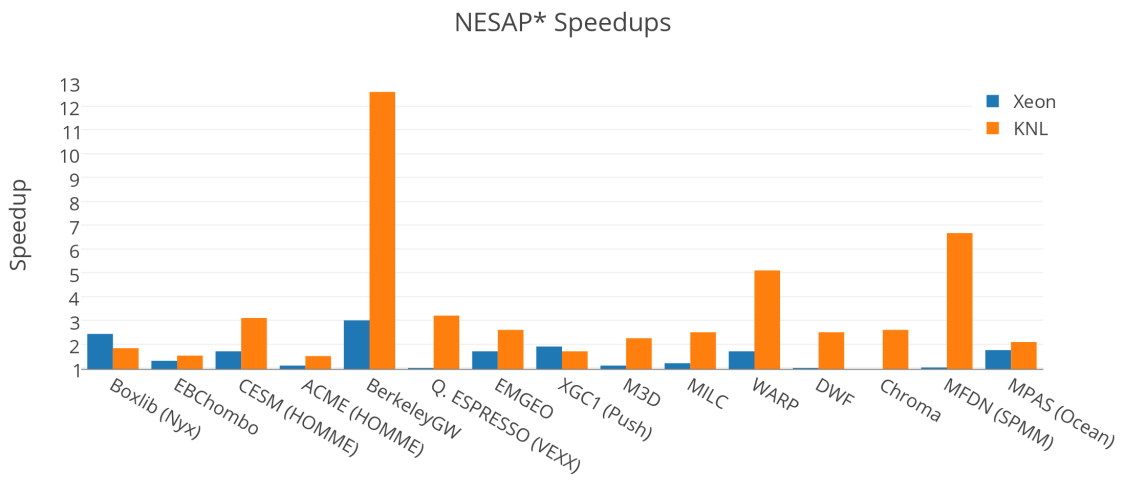

Fig. 9. Speedups obtained in NESAP applications over the course of the NESAP program.

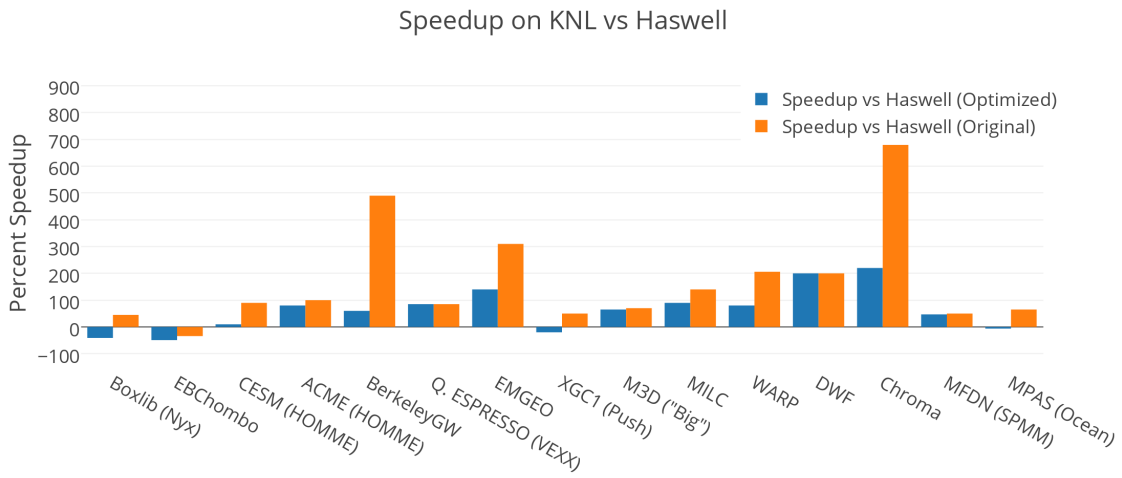

Fig. 10. Performance speedup on KNL vs Haswell. The blue bars represent a comparison to the optimized code on Haswell, while orange represent comparison to the pre-NESAP code on Haswell.

of 50-60% on the KNL nodes over the Cori-Phase 1 nodes. In Fig. 12, one can see, that more than any other application considered, BerkeleyGW sees speedups using AVX512 instructions vs scalar instructions.

The applications which perform faster on Haswell than KNL typically are often those with additional reuse out of the L3 cache on the Xeon architecture. As described above and shown in Fig. 11, this is evident especially in Boxlib and XGC1. This is not to say that MCDRAM is necessarily an ineffective cache (though latencies are significantly higher than a Xeon L3), but that the bandwidth advantage of MCDRAM is at least partially mitigated if codes have good reuse out of cache.

No applications showed a significant change in performance when fused multiply-add (FMA) instructions were disabled using the "-no-fma" compiler flag, as shown in Fig 12. Note, however, that applications that depend significantly on libraries are expected to be unaffected by this flag even though the compiler may heavily utilize FMA instructions.

IX. Conclusions

NERSC partnered with Cray, Intel and 20 application development teams to evaluate and optimize applications for the Cori system. While the effort has led to significant performance gains, it has more importantly led to a better understanding of how the NERSC workload is expected to perform on the Cori system and to the development of an overall optimization strategy.

The most straightforward beneficiaries of KNL are applications that are memory bandwidth bound and fit within the MCDRAM (including via the use of `FASTMEM` directives). However, for many applications with arithmetic intensities

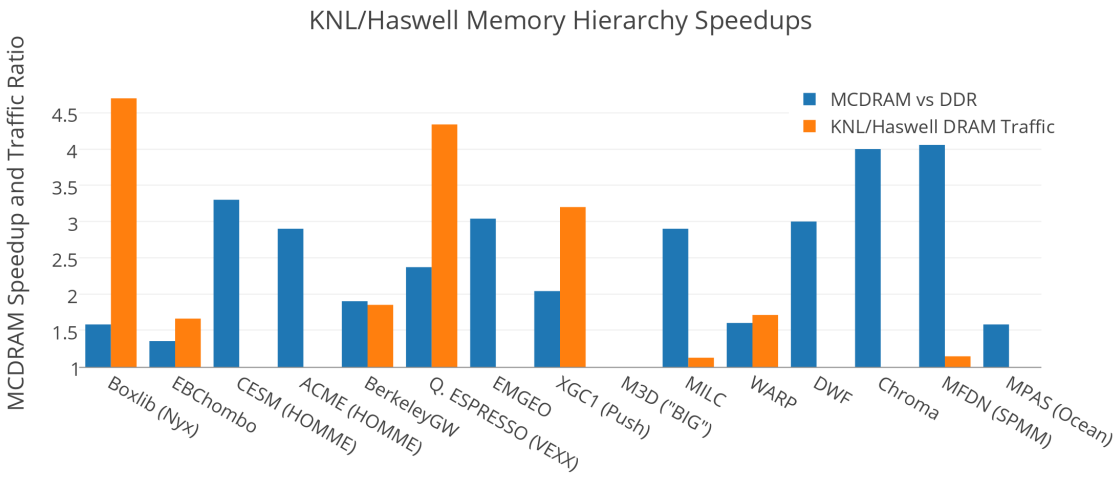

Fig. 11. Blue: application speedups from running with MCDRAM (either in Flat mode or Cache mode) vs running entirely out of off-package DRAM. Orange: ratio of the bytes transferred from DRAM (including MCDRAM) during program execution on KNL vs Haswell.

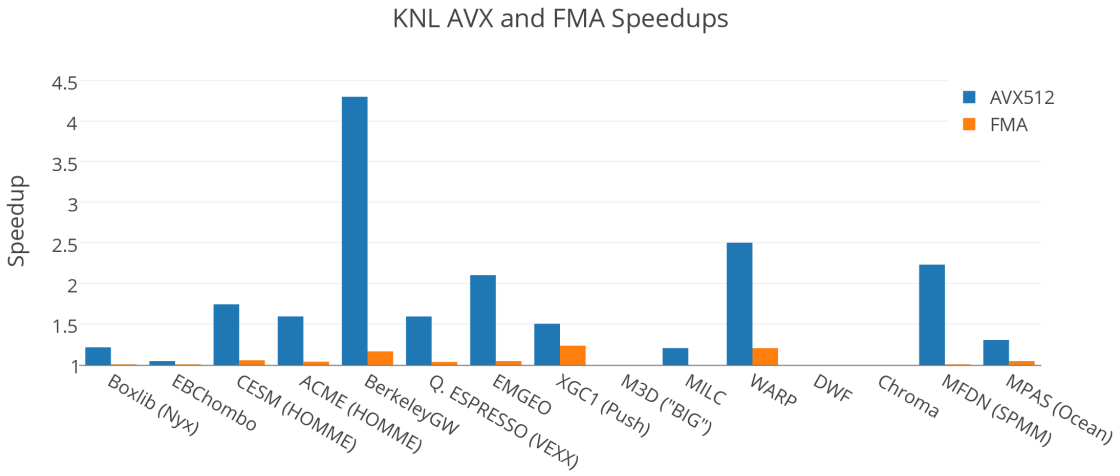

Fig. 12. Ratio of application performance in fully optimized build compared to builds utilizing the "-no-vec" or "-no-fma" flag in the Intel compiler suite. The asterisk on Quantum ESPRESSO and EBChombo indicates that these apps depend heavily on math libraries that are not affected by the compiler options.

near 1, it becomes essential to use all aspects of the KNL hardware efficiently (including physical cores and hardware threads, wide-vector units, L2 caches and MCDRAM) to outperform traditional Xeon architectures.

Acknowledgment

Research used resources of NERSC, a DOE Office of Science User Facility supported by the Office of Science of the U.S. DOE under Contract No. DE-AC02-05CH11231.

This article has been authored at Lawrence Berkeley National Lab under Contract No. DE-AC02-05CH11231 and UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the United States Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

Research on QE by PK was conducted at the Center for Nanophase Materials Sciences, which is a DOE Office of Science User Facility.

We acknowledge helpful conversations with Nicholas Wright, Brian Austin, Antonio Valles, Jeongnim Kim, Mike Greenfield, Richard Mills, Karthik Raman, Larry Meadows, David Prendergast.

REFERENCES

[1] http://www.nersc.gov

[2] NERSC and DOE Requirements Reviews Series: http://www.nersc.gov/science/hpc-requirements-reviews/

[3] P. Hohenberg and W. Kohn, "Inhomogeneous electron gas," Phys. Rev., vol. 136, pp. B864-B871, Nov. 1964.

[4] W. Kohn and L.J. Sham, "Self-consistent equations including exchange and correlation effects," Phys. Rev., vol. 140, pp. A1133-A1138, Nov. 1965.

[5] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. Dal Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, R. M. Wentzcovitch, "QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials," J. Phys.: Condens. Matter., vol. 21, pp. 395502, Sep. 2009.

[6] C. J. Cramer and D. G. Truhlar, "Density functional theory for transition metals and transition metal chemistry," Phys. Chem. Chem. Phys., vol. 11, pp. 10757-10816, Oct. 2009.

[7] R. Baer, E. Livshits, and U. Salzner, "Tuned range-separated hybrids in density functional theory," Annu. Rev. Phys. Chem., vol. 61, pp. 85-109, May 2010.

[8] J. P. Perdew, A. Ruzsinszky, L. A. Constantin, J. Sun, and G. I. Csonka, "Some fundamental issues in ground-state density functional theory: A guide for the perplexed," J. Chem. Theory Comput., vol. 5, pp. 902-908, Mar. 2009.

[9] K. Burke, "Perspective on density functional theory," J. Chem. Phys. vol. 136, pp. 150901, Apr. 2012.

[10] W J. Cohen, P. Mori-Sánchez, and W. Yang, "Challenges for density functional theory," Chem. Rev., vol. 112, pp 289-320, Dec. 2011.

[11] A. D. Becke, "Perspective: Fifty years of density-functional theory in chemical physics," J. Chem. Phys., vol. 136, pp. 18A301, Apr. 2014.

[12] L. Lin, "Adaptively compressed exchange operator," J. Chem. Theory Comput., vol. 12, pp. 2242-2249, Apr. 2016.

[13] E. J. Bylaska, K. Glass, D. Baxter, S. B. Baden, J. H. Weare, "Hard scaling challenges for ab initio molecular dynamics capabilities in NWChem: Using 100,000 CPUs per second," J. Phys.: Conf. Ser., vol. 180, pp. 012028, 2009.

[14] I. Duchemin and F. Gygi, "A scalable and accurate algorithm for the computation of HartreeFock exchange," Comp. Phys. Comm., vol. 181, pp. 855-860, Jan. 2010.

[15] T. A. Barnes, et. al., "Improved treatment of exact exchange in Quantum Espresso," To be published, 2016.

[16] G. O. Young, Synthetic structure of industrial plastics (Book style with paper title and editor), in Plastics, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 1564.

[17] A. S. Almgren, V. E. Beckner, J. B. Bell, M. S. Day, L. H. Howell, C. C. Joggerst, M. J. Lijewski, A. Nonaka, M. Singer, and M. Zingale. CASTRO: A New Compressible Astrophysical Solver. I. Hydrodynamics and Self-gravity. *The Astrophysical Journal*, 715(2):1221, 2010.

[18] Ann S. Almgren, John B. Bell, Mike J. Lijewski, Zarija Lukić, and Ethan Van Andel. Nyx: A Massively Parallel AMR Code for Computational Cosmology. *The Astrophysical Journal*, 765(1):39, 2013.

[19] W. Briggs, V. Henson, and S. McCormick. *A Multigrid Tutorial*. Society for Industrial and Applied Mathematics, 2nd edition, 2000.

[20] Phillip Colella and Harland M Glaz. Efficient solution algorithms for the Riemann problem for real gases. *Journal of Computational Physics*, 59(2):264 – 289, 1985.

[21] Phillip Colella and Paul R Woodward. The Piecewise Parabolic Method (PPM) for gas-dynamical simulations. *Journal of Computational Physics*, 54(1):174 – 201, 1984.

[22] Zarija Lukić, Casey W. Stark, Peter Nugent, Martin White, Avery A. Meiksin, and Ann Almgren. The Lyman-α forest in optically thin hydrodynamical simulations. *Monthly Notices of the Royal Astronomical Society*, 446(4):3697–3724, 2015.

[23] A. Nonaka, A. S. Almgren, J. B. Bell, M. J. Lijewski, C. M. Malone, and M. Zingale. MAESTRO: An Adaptive Low Mach Number Hydrodynamics Algorithm for Stellar Flows. *The Astrophysical Journal Supplement Series*, 188(2):358, 2010.

[24] Andy Nonaka, Yifei Sun, John B. Bell, and Aleksandar Donev. Low Mach number fluctuating hydrodynamics of binary liquid mixtures. *Communications in Applied Mathematics and Computational Science*, 10(2):163–204, 2015.

[25] George Shu Heng Pau, John B. Bell, Ann S. Almgren, Kirsten M. Fagnan, and Michael J. Lijewski. An adaptive mesh refinement algorithm for compressible two-phase flow in porous media. *Computational Geosciences*, 16(3):577–592, 2012.

[26] Will E. Pazner, Andrew Nonaka, John B. Bell, Marcus S. Day, and Michael L. Minion. A high-order spectral deferred correction strategy for low Mach number flow with complex chemistry. *Combustion Theory and Modelling*, 20(3):521–547, 2016.

[27] Weiqun Zhang, Ann S. Almgren, Marcus Day, Tan Nguyen, John Shalf, and Didem Unat. BoxLib with Tiling: An AMR Software Framework. *CoRR*, abs/1604.03570, 2016.

[28] CESM website http://www.cesm.ucar.edu/

[29] H. Morrison and A. Gettelman A New Two-Moment Bulk Stratiform Cloud Microphysics Scheme in the Community Atmosphere Model, Version 3 (CAM3). Part I: Description and Numerical Tests. J. of Climate, 21(15):36423659, 2008

[30] A. Gettelman and H. Morrison Advanced Two-Moment Bulk Microphysics for Global Models. Part I: Off-Line Tests and Comparison with Other Schemes. J. of Climate, 28(3):12681287, 2015.

[31] NERSC NESAP CESM Case Study: https://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/application-case-studies/cesm-case-study

[32] Youngsung Kim, John Dennis, Christopher Kerr, Raghu Raj Prasanna Kumar, Amogh Simha, Allison Baker, Sheri Mickelson Procedia Computer Science Special Issue: International Conference on Computational Science 2016, ICCS 2016. June 6-8, 2016, San Diego, California. Volume 80, 2016, pages 1450-1460.

[33] A. Simmons and D. Burridge An energy and angular momentum conserving vertical finite-difference scheme and hybrid vertical coordinates. Monthly Weather Review, 109: 758-766, 1981

[34] M. Taylor, J. Tribbia, and M. Iskandarani The spectral element method for the shallow water equations on the sphere Journal Computational Physics, 130: 92-108.

[35] John Dennis, Chris Kerr, Youngsung Kim, Raghu Kumar, Rashmi Oak, Amogh Simha Update on status of CESM on many-core. 2016 CESM Workshop. SEWG (Software Engineering Working Group) Presentation. June 20-23, 2016, Breckenridge, Colorado.

[36] S. Ku, et. al. Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. Nuclear Fusion, vol. 49 no. 11, Article 115021, 2009.

[37] E. S. Yoon, C. S. Chang A Fokker-Planck-Landau collision equation solver on two-dimensional velocity grid and its application to particle-in-cell simulation. Physics of Plasmas, 21, 032503 (2014)

[38] R. Hager, et. al. A fully non-linear multi-species FokkerPlanckLandau collision operator for simulation of fusion plasma. Journal of Computational Physics, col. 315, no. 15, pp 644-660, 2016

[39] WARP Particle-In-Cell code website http://warp.lbl.gov/ Accessed: 2016-08-30

[40] H. Vincenti, et. al. An efficient and portable SIMD algorithm for charge/current deposition in Particle-In-Cell codes, Computer Programs in Physics, To be published, 2016

[41] D. Doerfler, et. al. Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor. To be published, 2016.

[42] Williams, S.: Auto-tuning Performance on Multicore Computers. Ph.D. thesis,EECS Department, University of California, Berkeley (December 2008)

[43] Williams, S., Watterman, A., Patterson, D.: Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Communica-tions of the ACM (April 2009)

[44] Williams, S. Roofline performance model, http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/