A Novel Multi-Level Integrated Roofline Model Approach for Performance Characterization

Tuomas Koskela[1,5,6†], Zakhar Matveev[3], Charlene Yang[1], Adetokunbo Adedoyin[4], Roman Belenov[3], Philippe Thierry[3], Zhengji Zhao[1], Rahulkumar Gayatri[1], Hongzhang Shan[2], Leonid Oliker[2], Jack Deslippe[1], Ron Green[4], and Samuel Williams[2]

[1] NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[2] CRD, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[3] Intel Corporation
[4] Los Alamos National Laboratory
[5] University of Helsinki, Finland
[6] University of Turku, Finland

[†]Contact: tuomas.koskela@helsinki.fi

Abstract. With energy-efficient architectures, including accelerators and many-core processors, gaining traction, application developers face the challenge of optimizing their applications for multiple hardware features including many-core parallelism, wide processing vector-units and on-chip high-bandwidth memory. In this paper, we discuss the development and utilization of a new application performance tool based on an extension of the classical roofline-model for simultaneously profiling multiple levels in the cache-memory hierarchy. This tool presents a powerful visual aid for the developer and can be used to frame the many-dimensional optimization problem in a tractable way. We show case studies of real scientific applications that have gained insights from the Integrated Roofline Model.

Keywords: Performance models, Application Performance Measurement, Roofline, Knights Landing

1 Introduction

As HPC systems move towards exascale computing, the growing complexity in processor micro-architecture makes it more and more challenging to develop performant and energy-efficient applications. As an example, the Intel many-core Xeon Phi processor architecture has introduced a large number of relatively low-frequency cores per chip with additions of on-package high-bandwidth memory and wide vector units. It offers increased computing power for algorithms that can leverage high parallelism and vectorization, at a lower energy cost. Although optimizing (e.g. more parallelism) applications is beneficial on most platforms, a metric is required to guide application developers in their performance optimization efforts so that applications can be optimized for the correct performance

bounds. Guiding users through this optimization process is a challenge that many HPC facilities around the world are facing as they transition their communities to energy-efficient architectures.

As the Mission HPC facility for the United States Department of Energy's (DOE) Office of Science, the National Energy Research Scientific Computing Center (NERSC), located at Lawrence Berkeley National Lab, is addressing this challenge with its broad user community of over 6000 users from 600 projects spanning a wide range of computational science domains. NERSC recently deployed the Cori system — a Cray XC-30 system comprising over 9600 Xeon-Phi 7250 processors (code-named "Knights Landing" or KNL for short).

Each KNL processor includes 68 cores running at 1.4GHz (1.2GHz for AVX code) and capable of hosting four HyperThreads (272 HyperThreads per node). Each core has a private 32KB L1 cache and two 512-bit wide vector processing units. Each pair of cores (called a "tile") shares a 1MB L2 cache and each node has 96GB of DDR4 memory and 16GB of on-package high bandwidth (MCDRAM) memory.

Users are transitioning to this system from NERSC's Edison system that contains roughly 5000 nodes with a more traditional dual-socket Xeon (Ivy-Bridge) architecture. When transitioning from Edison to Cori, users are faced with a number of important changes: the "many-core" nature of Cori, the increased vector widths (512-bit) provided with the AVX-512 instruction set, and changes to the cache-memory hierarchy including the lack of an L3 cache but addition of an on-chip 16GB fast-memory (MCDRAM) memory layer that can be configured as a transparent cache. In order to effectively optimize an application for KNL, users therefore need to know which of these hardware features should be targeted in order to reach the largest gains. And practically, they need to know when to stop: i.e. when they've reached the expected performance for their application on the architecture.

The Roofline model [1] is a visually-intuitive performance model used to bound the performance of applications running on multiple architectures. Rather than using percent-of-peak estimates, the Roofline model can be used to quantitatively compare the performance of an application to the performance bounds, or ceilings, set by the architecture of the compute platform. The Roofline model combines this information into a simple performance figure that can be used to determine both algorithmic and implementation limitations of an application.

The Roofline model characterizes an application's performance in gigaflops per second (GFLOPS) as a function of its arithmetic intensity (AI). AI is the ratio of floating-point operations performed to the bytes transferred from cache or memory during execution. This performance number can be compared against the bounds set by the peak compute performance and the cache/memory bandwidth of the system to determine what is limiting performance. The measurement of AI can be done in multiple ways, based on different levels of the memory hierarchy of modern computer systems. In literature, the Roofline model is often labeled by the level of memory its AI is measured from. The most well-known flavors of Roofline are the Cache-Aware Roofline Model (CARM) that measures

the AI presented to the cache hierarchy [2,3] and the classical Roofline model that measures the DRAM AI (AI after filtering by the cache hierarchy) [1].

Over the past few years, the utility of the Roofline performance model for guiding and tracking application optimizations has been demonstrated [4]. However, these efforts have been limited in a couple of ways: gathering Roofline data was cumbersome and limited in practice to a few code regions manually designated by the programmer and, secondly, data from only one level of the cache-memory hierarchy is typically gathered, while many applications have complex dependencies on the entire hierarchy.

In this paper, we discuss the benefits of an Integrated Roofline Model (IRM) that collects the AI from all available levels of the memory hierarchy, and present a tool, Intel® Advisor, that automates the data collection and visualization in a user friendly fashion. The novelty of this work with regards to [2,3] is the new performance analysis method that uses memory traffic between all levels of the memory hierarchy simultaneously and the demonstration of a tool for automating the data collection. In CARM, one has a single AI (that is usually close to the L1 AI) and can estimate effective memory bandwidth by comparing observed performance to the L1, L2, ..., DRAM ceilings. Thus, if one observes performance between the L2 and L3 ceilings, one can conclude that the average memory bandwidth is somewhere between the L2 and L3 bandwidths. Unfortunately, CARM does not actually calculate the attained memory bandwidth at each level nor does it identify which level might be a performance bottleneck. To that end, we developed and implemented a hierarchical roofline formulation with a unique AI for each memory level and used a cache simulator to accurately calculate the data movement between each level of the memory hierarchy. With this information we can calculate at which level of the memory hierarchy the dominant data movement occurs and the degree to which bandwidth at that level is overprovisioned. An example that illustrates these improvements will be shown in section 4.2. Integration of this technology into a performance tool has allowed it to be applied to full applications, linked to source and assembly, and visualized using the well-known Roofline formulation. It allows us to automatically determine whenever CPU or a given level of memory hierarchy are the primary bottlenecks in terms of throughput and expose it in a visually-intuitive manner.

The remainder of the paper is organized as follows: In Section 2 we discuss the implementation of the Integrated Roofline Model in the Intel Advisor tool. In Section 3, we show case studies of how real science application performance is characterized using the Integrated Roofline Model. In Section 4, we demonstrate two examples where the Integrated Roofline Model has been used to highlight the effects of optimizations on applications and to demonstrate application performance differences of the KNL and Intel Xeon (Haswell generation) architectures that are present in a smaller data-partition on Cori.

2 Intel Advisor Roofline: Underlying Design and Methodology

Three measurements from an application are needed for roofline analysis: the number of floating-point operations executed, the number of bytes transferred from memory and the execution time. In this section we discuss the measurements implemented in Intel Advisor to collect the data required to build the IRM [3,5,6].

In order to measure memory traffic and attribute it to the loops and functions of the application, we use binary instrumentation and cache simulation. We process all memory arguments of the executed instructions; the CARM traffic is obtained by adding their sizes, while feeding their sizes and addresses to the cache simulator provides the numbers for other levels of the memory hierarchy. The cache simulator is configured according to the actual hardware properties including individual caches, cache capacities, as well as core and socket counts.

The traffic for each memory level is defined as a number of bytes transferred between corresponding component and lower (closer to the CPU core) cache or the core itself. The only exception is L1 cache — where we count the total number of bytes accessed by CPU instructions instead, producing approximately the same arithmetic intensity as in the CARM. The difference is caused by the instructions that access memory by bypassing the cache subsystem, e.g., non-temporal stores used to store data not expected to be reused.

We attribute traffic to the instruction that caused it — and, afterwards, group it by loops and functions adding the numbers for corresponding instructions. For example, if a load causes a cache miss and in order to place the new line in cache it is required to evict the line modified earlier, storing it to the next level of the memory hierarchy, the store traffic is attributed to this load instruction. This may sound counter intuitive, but note that to correctly place the loop on the Roofline chart we need to measure the traffic generated while the loop was executing, thus even if it evicts cache lines modified by a preceding loop nest, corresponding store traffic should be taken into account.

Note that all transfers besides "L1 traffic" values are actually done in cache line units, which also affects the traffic. For example, if an instruction loads one byte and the load causes a cache miss and is eventually served by DRAM, we will measure one byte of L1 traffic, and the traffic defined by the cache line size (64 bytes on most modern CPUs) for all other caches and DRAM. This may lead to "L1" arithmetic intensity being larger than L2 or even DRAM, while the opposite is true for the code with good cache line reuse.

We also count the number of floating-point operations using binary instrumentation, analyzing floating-point instructions. For vector instructions, this enables us to accurately count the number of elements actually processed, i.e., properly account for masked vector instructions, noting that instructions can also involve several FLOPs (e.g. for FMA instructions). By implementing a time, traffic and FLOP measurement workflow in Intel Advisor it became possible to fully automate Roofline characterization not only for individual loops or functions, but for full applications.

The binary instrumentation by its nature causes significant overhead. The overhead varies based on the application (see Table 1). In general, the better the application is using the CPU resources, the higher is the influence of instrumentation and cache simulation logic. Particularly, the overhead is usually higher for the code with good cache locality because simulation code, besides consuming computational resources, evicts application data from the cache, increasing cache miss rate; if most accesses from the application are served by DRAM anyway, the impact is lower. The overhead is also affected by the other factors, such as the use of Hyper-Threading or thread synchronization pattern.

Note that in order to correctly measure the execution time of the program and individual loops, memory traffic estimation is done in a second analysis pass. So time is measured in first separate pass, using a low-overhead non intrusive time sampling technique, and therefore cache simulation overhead does not affect the execution time (seconds) and final FLOP/s measurements representativeness and accuracy.

Table 1. Cache simulation overhead for the applications in the paper

Name	**XGC1**	**VASP**	**fwi2d**	**SW4**	**CoMD**	**GPP**	**BIGSTICK**
Overhead	37×	21×	21×	23×	8×	9×	18×

Performance counters were used to validate the cache simulation results prior to this work. We used different compute kernels and benchmarks for validation and the discrepancy was found to be within 10% on average. We chose to use cache simulation, since, even though core (and offcore response matrix) counters can be used to measure read traffic from DRAM, writes to DRAM (caused by dirty lines evictions from LLC) are not measured. Note that although there is an OFFCORE_REQUEST bit corresponding to writebacks, it refers to writebacks from L2 to LLC. Counting writes from LLC to DRAM would require an additional OFFCORE_RESPONSE type. Furthermore, measuring traffic on loop/function boundaries requires source/binary instrumentation that may distort PMU measurements (especially for small inner loops). Also, in parallel code, it only makes sense on parallel region boundaries. Thus, it would not be possible to deep-dive inside the parallel regions and measure roofline data for inner loops or functions.

In this paper we use the following techniques available to analyze Roofline data generated by Intel Advisor. First, Intel Advisor's interactive GUI is utilized to explore Roofline charts directly (Figure 2). Secondly, the Intel Advisor command line interface or Intel Advisor Python customization API([7]) can be used to implement custom extension of Intel Advisor; this approach was used in this paper to generate custom Integrated Roofline charts shown in the article (e.g. see Figure 1).

3 Application Characterization Case Studies

In this section, we show the utility of the roofline model for assessing the performance of scientific applications that routinely run on NERSC systems. Real-life scientific applications often feature complex algorithms that may be very difficult to analyze analytically. For demonstration of the integrated roofline method on a simple analytic benchmark, the reader is encouraged to refer to [8,9]. The applications analyzed in this paper are summarized in Table 2.

Table 2. General characteristics of the application evaluated in this paper. Note, "BW" stands for memory bandwidth

Name	**XGC1**	**VASP**	**fwi2d**	**SW4**	**CoMD**	**GPP**	**BIGSTICK**
Domain	Magnetic Fusion	Materials Science	Geophysics	Geodynamics	Polycrystalline Materials	Material Science	Nuclear Physics
Motif	PIC	DFT + CG	FD Stencils	FD Stencils	MD	DFT	CI
Release	upon request	paid	closed	open source	open source [10]	open source	upon request
MPI **OpenMP**	✓ ✓	✓ ✓	✓ ✓	✓ ✓	✓ ✓	 ✓	✓ ✓
LOC	100k	470k	2K	87k	4K	400	91k
Perf. bound	Gather/ Scatter, Compute	BW, latency	BW	BW, Compute	BW	BW	BW

3.1 XGC1

XGC1 (X-point Gyrokinetic Code) [11] is a fusion plasma physics gyrokinetic particle-in-cell (PIC) code originally from Princeton Plasma Physics Laboratory (PPPL). It is one of the codes selected for the US exascale computing project (ECP). It is primarily used to study turbulence in the edge region of tokamak fusion plasmas — essential for the success of future fusion reactors.

The main unique feature of XGC1 is its use of unstructured meshes to simulate the whole tokamak volume. The mesh is decomposed over MPI ranks into poloidal planes that are connected via a field-following mapping. Within a poloidal plane, particles are decomposed over both MPI ranks and OpenMP threads. XGC1 uses kinetic ions and electrons, and uses a sub-cycling scheme for the electron motion that is the most time-consuming part of the simulation. In a typical production run on Cori, roughly 70-80% of the total CPU time is spent in the electron push kernel [12].

Within the electron push kernel, communication between threads is only needed roughly every 50 electron time steps. The electron push algorithm uses a 4^{th} order Runge-Kutta scheme to integrate the gyrokinetic equation of motion [11]. The computation has a high flop to byte ratio, but CPU time is dominated by indirect memory accesses and latency from gather/scatter instructions

due to the random motion of particles across the grid. This is a typical feature of PIC codes and is exacerbated in XGC1 by the unstructured mesh.

We run XGC1 on a single KNL node in quad-cache mode for an artificially small problem size, to make data collection with Advisor feasible. The mesh consists of 7500 nodes in a single axisymmetric plane and the number of particles is 25,000 per thread, 1.6 million total. We run the code for two ion time steps with 50 electron sub-cycles per time step, without collisions, using 16 MPI ranks and 4 OpenMP threads per rank. The Advisor data is collected on rank 0 only.

The five most time-consuming loops of XGC1 are shown on the Roofline chart in figure 1. All the loops shown are called from the electron push kernel, which dominates the ion push by a factor 50 in call volume. The loops all have good cache locality in the L2 cache, shown by a large increase in AI from LLC to L2, and the highest-performing loops also have good L1 locality, while the rest have only moderate L1 locality. The kernels clearly are not bandwidth bound, with the possible exception of bicub_mod:295 being bound by the L1 bandwidth. It also is worth noting that the DRAM traffic within these loops is too small to be measured by Advisor. That is, the loops are fully running from cache. The loops are vectorized and do not fall clearly into either bandwidth or compute bound regimes. Our hypothesis is that they are bound by cache throughput which is specific to the load instruction. This has been confirmed by instruction set analysis which shows gather/scatter instructions are generated in the loops. Our conclusion is the high-performing loops are bound by L1 instruction throughput and the lower-performing loops by L2 throughput. One could measure a memory bandwidth roofline for gather/scatter instructions to present the more realistic performance bound in the roofline model, this is planned for future work.

3.2 fwi2d

Fwi2d (full waveform inversion) is a seismic imaging code from the Paris School of Mines (MinesParisTech) used to obtain subsurface images from low frequency wave velocity fields. This is critical for successful exploration and reservoir delineation in oil and gas exploration, but such algorithms can be used for civil engineering as well. FWI is a computationally-intensive process that requires propagating waves using time (or frequency) domain wave equation solvers. We consider here a second-order wave equation and we are using a time domain finite difference (TDFD) approach based on explicit finite difference schemes in time (4^{th} order) and space (8^{th} order) with a quasi-Newton (with L-BFGS algorithm) optimization scheme for the model parameters update. The stencil in use to compute the derivatives has an extremely strong impact on the flop per byte ratio, on the memory accesses, and finally on the implementation efficiency. Those algorithms are also known as time reversal techniques that need cross correlation of a forward propagating field and a backward propagating field. This method usually leads to heavy I/O to keep snapshots of the forward wavefields. In the present isotropic acoustic 2D implementation of FWI, we keep the snapshots in memory for simplicity. The main advantage of a 2D implementation is to quickly evaluate new features such as more complicated wave equations, new

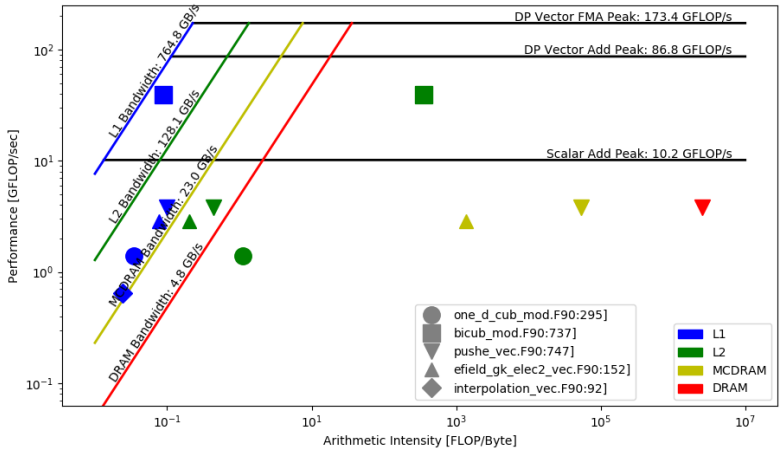

Fig. 1. XGC1 Roofline figure for Cori (KNL) in a quad-cache configuration. The symbols represent different loops and the colors of the markers represent AI's of the loops measured from different memory levels. The memory bandwidth ceilings are similarly colored. That is, points of a given color should be compared against the memory ceiling of the same color. Although `bicub_mod` attains near L1 roofline performance, most other kernels are well-below the L2 and scalar add rooflines.

cost functions, finite-difference stencils or boundary conditions and evaluation of new memory hierarchies in the context of snapshot management. This technique is popular due to its simple stencil based implementation on a Cartesian grid and its natural parallelization in the shot domain, e.g. the parallelization is achieved through standard MPI seismic shot distribution and OpenMP for domain decomposition within the PDE loop.

In the example runs, we try to understand OpenMP scalability and the performance impact of absorbing boundary condition that potentially impact data alignment and vectorization. As presented on Figures 2 and 3, we can analyze the data movement between memory levels for each loop. Since the number of FLOPs is the same between those levels, any move comes from a change in the number of bytes transferred. The amplitude of the horizontal movement of each dot is indicative of potential bottlenecks. For example, the triangles on Figure 3 are very close for the L1, L2 and L3 levels (blue, green, yellow) and far to the right for the DRAM to LLC traffic (red triangle). This demonstrates that data resides in the cache and is not impacted by DRAM bandwidth. A look at the vectorization analysis will then give some recommendations on how to improve the performance (we ultimately want those triangles against the roofline). Another interesting behavior is visible with the blue and green diamonds of Figure 3 where we do not have any data traffic between DRAM-LLC and LLC-L2 levels

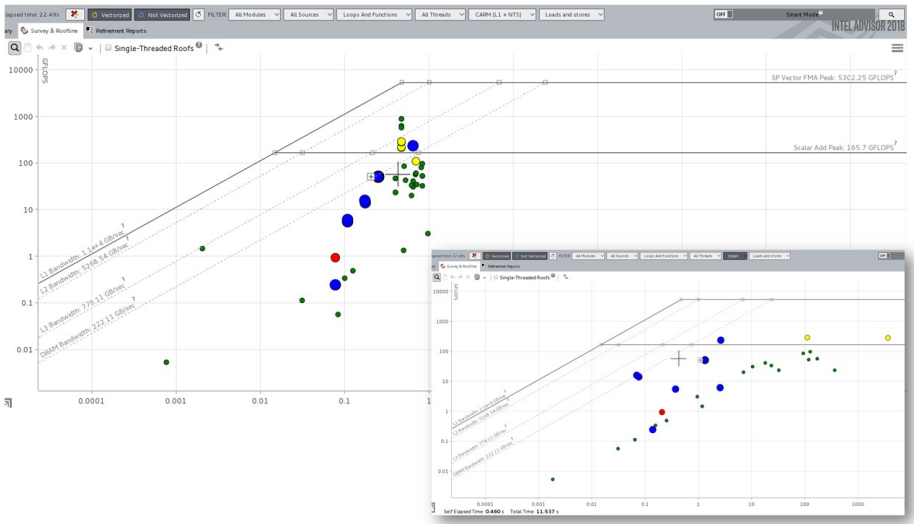

Fig. 2. fwi2d Intel Advisor 2018 "Roofline" view screen-shots. These two pictures represent the data traffic between L1 and register (top left) and between L1 and L2 (bottom right). As the number of FLOPs remains constant between the pictures, we can see that several functions/loops have the same volume of bytes transferred while some other points suffer from cache misses that strongly increase the data traffic.

(no yellow or red diamonds). The increase in data movement between L1 and registers demonstrates a possible issue with vectorization and data alignment.

Using the multi-level Roofline view for all functions and loops is a nice companion to vectorization analysis in order to characterize a full application at the function/loop levels on a given platform or when moving from one machine to another generation. To illustrate this, we performed tests on a dual-socket server using Intel® Xeon® Gold 6148 processors codenamed Skylake with 20 cores running at 2.4GHz and 192GB of DDR3-2667 DRAM, leading to a stream triad bandwidth of about 200GB/s. This architecture is different from the KNL processor of the Cori system, corresponds to the latest Xeon architecture, and contains a number of enhancements (AVX512 instruction set and larger cache sizes). In some cases, applications may need to be optimized for the new cache hierarchy. In fwi2d this is reflected in the need to reshape the cache block sizes to fit L2 and benefit from its bandwidth. More details are available at [13].

3.3 VASP

The Vienna Ab-initio Simulation Package (VASP) [14,15] is a widely used materials science application, supporting a wide range of electronic structure methods, from Density Functional Theory (DFT) to many-body perturbation approaches (GW and ACFDT). VASP solves a set of eigenvalue and eigenfunction (wavefuntion) pairs for the many-body (electrons) Schrödinger equation iteratively

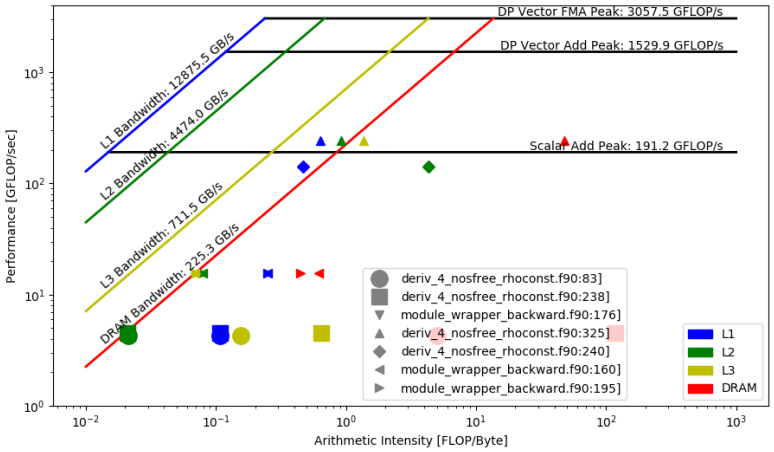

Fig. 3. Roofline for Fwi2d using 1 MPI rank of 40 OpenMP threads running on a dual socket Skylake server. Selected loops are from the stencil routines and boundary conditions. We can see, the data movement between memory levels is very different for each loop and each of the loops is not bounded by the same level, i.e by the same bandwidth. This is one of the key features of the multi-level view.

within a planewave basis set (Fourier space). VASP computation consists of many independent 3D FFTs, matrix-matrix multiplications, matrix diagonalizations, and other linear algebra methods, such as Gram-Schmidt orthogonalization, Cholesky decomposition, and matrix inversion. Therefore VASP heavily depends on optimized mathematical libraries.

VASP was written in Fortran and parallelized with MPI. To exploit more energy-efficient processors like Intel KNL, the VASP developers have added OpenMP directives to the code recently to address increased on-node parallelism. In the hybrid MPI+OpenMP VASP code, the bands are distributed over MPI tasks, and the coefficients of the bands are distributed over OpenMP threads, either explicitly via OpenMP directives, or implicitly via the use of threaded libraries like FFTW or LAPACK/BLAS3. To exploit wider vector units, VASP employs OpenMP SIMD constructs using both explicit loop-level vectorization via `omp simd` and sub-routine/function vectorization through `omp declare simd`. To effectively vectorize the nested function calls and complex loop structures mixing scalar and vector code that compilers often fail to auto-vectorize, the code employs a combination of user-defined high-level vectors together with OpenMP SIMD loop vectorization. More details about the OpenMP implementation and SIMD optimizations in VASP can be found in [16].

Figure 4 shows the four most time consuming loops in the hybrid MPI+OpenMP VASP (code path: hybrid functional calculation with the damped iteration scheme) on the roofline chart for Cori KNL. VASP was run with four KNL nodes in quad-

cache using 64 MPI tasks and 8 OpenMP threads per task (16 MPI tasks per node). Note, the performance data is collected on rank 0 and the flops are normalized to full node by simply multiplying the relevant values by the number of MPI tasks per node. The benchmark used is a 256-atom Silicon supershell with a vacancy. The hybrid MPI+OpenMP VASP (last commit 10/16/2017) was compiled with the Intel compiler (2018.0.128) and was linked to the MKL, ELPA (2016.05.004), and Cray MPICH (8.6.0) libraries.

Figure 4 shows that the performance of the most time consuming loops (each of them accounting for 4-6% of the total execution time) are well below the scalar add peak. Except the loop, apply_gfac_exchange_, being MCDRAM bandwidth bound, the dots for the rest of the loops are well below the corresponding level's memory bandwidths, indicating they are neither compute or bandwidth bound but likely latency bound. Advisor detected inefficient memory access patterns and assumed dependencies in these loops. The large separation between the MCDRAM and DRAM arithmetic intensities is indicative that the working set fits well in the MCDRAM cache. Note that the four most time consuming loops shown in the figure are not the ones that execute the most floating-point calculations. VASP executes most of the floating-point calculations via the BLAS and FFT routines in MKL whose performances, being closer to or above the double precision vector peak, result in a shorter execution times.

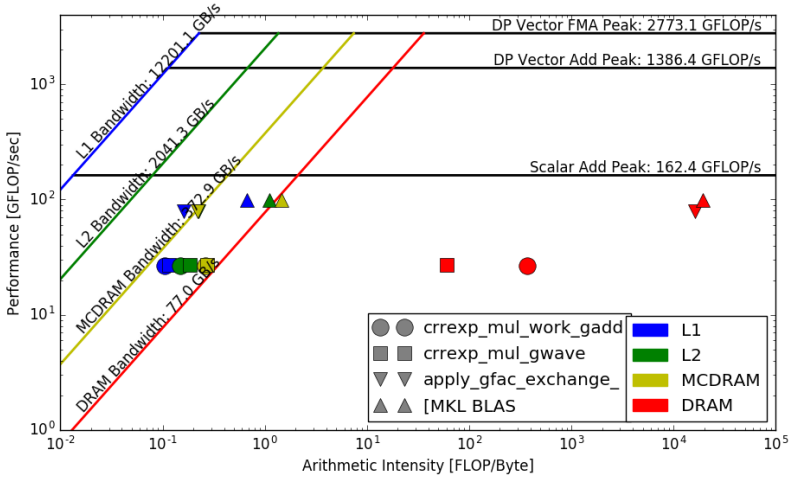

Fig. 4. VASP Roofline for KNL. The four most time consuming loops are shown in the chart. Except the loop, apply_gfac_exchange_, which appears to be the MCDRAM bandwithd bound, the rest of the loops are neither compute or bandwidth bound.

3.4 BIGSTICK

BIGSTICK [17] implements the parallel Configuration Interaction (CI) method (widely used to solve the nuclear many body problem) using Fortran 95 and MPI+OpenMP. Perhaps the greatest challenges in efficient implementation and execution of the CI method is its immense memory and data movement requirements. In CI, the non-relativistic many-body nuclear Schrödinger equation is cast as a very large sparse matrix eigen-problem with matrices whose dimension can exceed ten billion. With typical sparsity (between 1 and 100 nonzeros per million matrix elements) such large-scale sparse matrix eigen-problems place high demands on memory capacity and memory bandwidth. To reduce the memory pressure, BIGSTICK reconstructs nonzero matrix elements on the fly. As a results, the memory requirements, compared with the stored matrix approach, is reduced 10-100×. To further reduce memory requirements, BIGSTICK may be run in a hybrid MPI+OpenMP mode. Figure 5 shows the Roofline for BIGSTICK on KNL using 1 MPI process of 64 threads. The data set used is b10nmax4 (^{10}B), an ab initio calculation that has five protons and five neutrons (10B); the designation Nmax=4 describes the model space and signifies the maximum excitation in units of harmonic oscillator energies. Generally speaking, two interrelated factors drive down both AI and performance — high-stride memory access patterns and a lack of vectorization. However, the large DRAM AI suggests the MCDRAM cache is effectively capturing the working set.

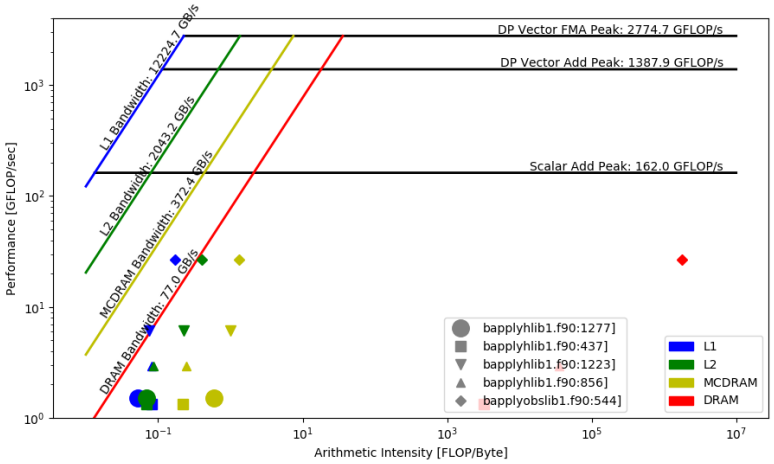

Fig. 5. Roofline for BIGSTICK on KNL for 1 MPI process with 64 OpenMP threads. The low performance is due to high stride data access without vectorization.

3.5 SW4

SW4 [18] is a block-structured, finite difference code that implements substantial capabilities for 3D seismic modeling. SW4 is parallelized with MPI and has performance kernels (SW4Lite) threaded with OpenMP.

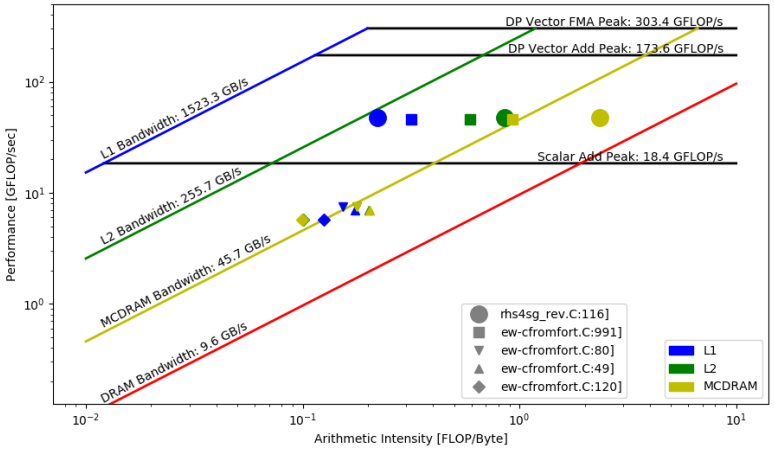

Fig. 6. SW4 running with 8 MPI ranks of 8 threads on KNL in quad-cache. Observe four kernels are strongly tied to MCDRAM(LLC) bandwidth.

Figure 6 shows the Roofline model for SW4 running with 8 processes of 8 threads on KNL in quad-cache. The performance of four of the kernels are directly tied to MCDRAM bandwidth, while the fifth, rhs4sg_rev, dominates the runtime, underperforms its MCDRAM limit, but exceeds the scalar FMA performance. L1, L2, and MCDRAM(LLC) AIs are widely separated indicating multiple levels of locality and reuse distances. Note, we do not show DRAM AI as the problem size completely fits in MCDRAM.

The rhs4sg_rev and addsgdfort_indrev are the most time consuming routines in the project. Nearly 80% of the computation time is spent in these routines. From the Figure 6, we can observe that both these routines are optimized efficiently and since there are sizable gaps between L1, L2 and LLC, it indicates that there is reuse in all three levels of cache. We do not show the DRAM data in Figure 6, since the problem size shown fits completely in the MCDRAM.

4 Guided Optimization Case Studies

4.1 CoMD

CoMD [19,20,21,22], a proxy/mini application developed at Los Alamos National Laboratory (LANL), was designed to mimic the workloads of ddcMD [23] and Scalable Parallel Short-range Molecular Dynamics (SPaSM) [23] and is a material science application for performing molecular dynamics (MD) simulation on polycrystalline materials. MD algorithms are classified as N-body problems with an approximate complexity of $O(n^2)$ or lower. The two types of force calculations implemented in CoMD (typically the bottleneck of MD algorithms) is the Leonard Jones (LJ) and Embedded Atom Model (EAM). CoMD is implemented in C with MPI and OpenMP.

Initial observations showed that the LJ force kernel strongly dominates the run time. Therefore, for the purpose of demonstrating the viability of the Roofline approach, we will focus on the LJ force kernel.

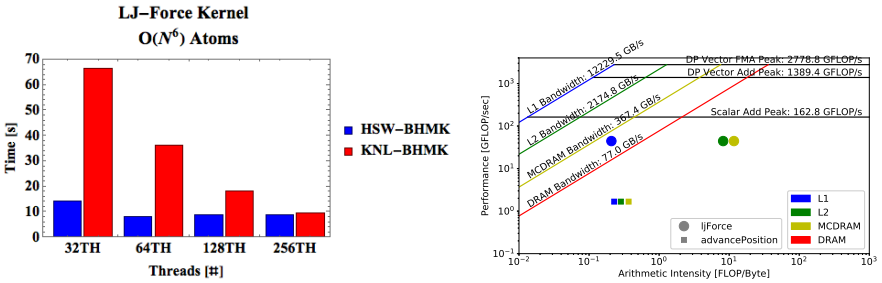

Fig. 7. (Left) Out-of-the-box CoMD LJ-Force on KNL generally underperforms Haswell (oversubscription beyond 64 threads). (Right) Roofline analysis of CoMD on KNL shows the LJ kernel dominates the run time and has high data locality. Note, HyperThreading on Haswell only supports 64 simultanous threads.

Figure 7(left) shows out-of-the-box CoMD performance on KNL compared to a two-socket Haswell node (thread concurrencies greater than 64 indicate oversubscription). Data provided by Intel Advisor, Figure 7(right), shows that the LJ kernel dominates the run time and has substantial L2 and MCDRAM data locality. However, it clearly shows that there is a great deal of L1 data movement (much lower L1 AI). Advisor noted the lack of vectorized loops, unaligned data, and non-unit stride stores. Therefore, the conclusion from the roofline analysis was the performance is not bound by memory bandwidth, since good locality is achieved, and optimization efforts on KNL should focus on leveraging the vector instructions to reach higher compute ceilings.

After several optimization sessions with the goal of improving the data and thread level parallelism, figure 8 shows approximately 30% improvement in the LJ-force kernel performance on both KNL and HSW. The modernization effort in

CoMD for the LJ-force kernel includes improving vectorization via simd clauses, branch hints for simd, simdized functions, alignment, compiler hints, and data structure transformations. However, the roofline performance figure indicates that work remains to bridge the remaining performance gaps.

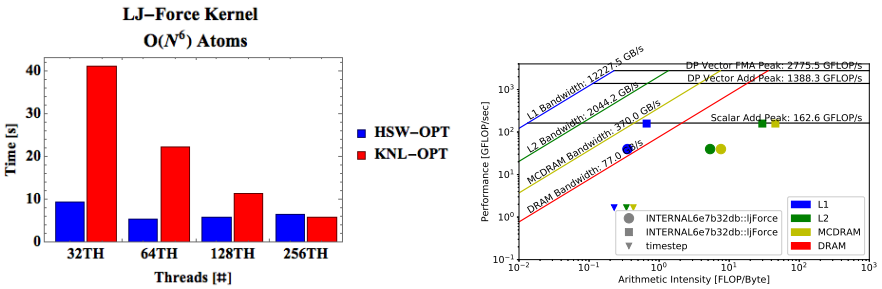

Fig. 8. (Left) Optimized CoMD LJ-Force on KNL and Haswell. (Right) Roofline analysis of optimized CoMD on KNL. Note, HyperThreading on Haswell only supports 64 simultanous threads.

4.2 BerkeleyGW (GPP)

BerkeleyGW [24] is a material science application that predicts the excited-state properties of a wide range of materials. GPP [25] is a proxy code for BerkeleyGW, written in Fortran90 and parallelized with OpenMP. It calculates the electron self-energy using the common General Plasmon Pole (GPP) approximation [26]. The computation represents the work an individual MPI task would perform in a much larger calculation (typically spanning hundreds or thousands of nodes). - meaning properties of system where the electrons are in an excited configuration. In computational terms, it performs a tensor-contraction like operation, where a few pre-computed complex double-precision arrays are multiplied/summed over a certain dimention and collapse into a 3x3 matrix. The core expression is:

$$\Sigma_n = \sum_{n'} \sum_{\mathbf{G}\mathbf{G}'} M^*_{n'n}(-\mathbf{G}) M_{n'n}(-\mathbf{G}') \frac{\Omega^2_{\mathbf{G}\mathbf{G}'}}{\tilde{\omega}_{\mathbf{G}\mathbf{G}'}\left(E - E_{n'} - \tilde{\omega}_{\mathbf{G}\mathbf{G}'}\right)} v(\mathbf{G}'), \qquad (1)$$

where Σ represents the correlation energy of an electron state, denoted by n. Ω, $\tilde{\omega}$ and M are precomputed complex double-precision array in reciprocal (G) space, $v = 1/\mathbf{G}^2$ is the electronic Coulomb interaction, and E is the energy at which we evaluate the material response - typically at or near the energy of the electron orbital. The code implements several nested loops, where the innermost loop iterates over the longest dimension of arrays. For a small BerkeleyGW problem, with 512 electrons (bands) and 32,768 plane wave basis elements, the innermost loop, for example at line 303 in the code [25], must read/write 2MB of data for each outer iteration. As this 2MB working set doesn't fit into the

L2 cache on either Haswell or KNL, a cache-blocking strategy is deployed. The following pseudo code demonstrates the original code:

```
do my_igp = 1, ngpown ! OpenMP
  do iw = 1 , 3
    do ig = 1, igmax
      load wtilde_array(ig,my_igp) !512KB per row
      load aqsntemp(ig,n1) !512KB per row
      load eps(ig,my_igp) !512KB per row
```

and the optimized code, with cache blocking:

```
do my_igp = 1, ngpown ! OpenMP
  do igbeg = 1, igmax, igblk
    do iw = 1 , 3
        do ig = igbeg, min(igbeg + igblk,igmax)
          load wtilde_array(ig,my_igp)
          load aqsntemp(ig,n1)
          load eps(ig,my_igp)
```

Using 32 threads on the Cori/Haswell nodes and 64 threads on the Cori/KNL nodes (quad-cache), Figure 9 shows that this core computational loop is MCDRAM-bound on KNL and L3-bound on Haswell, since there is essentially no difference between the LLC, L2 and L1 AI's. This means that data is being streamed from the LLC and there is no reuse of data in L1 or L2. Table 3 shows that when one applies cache blocking, the LLC AI improves by 3×, which is due to the fixed trip count of three for loop *iw*. Likewise, Figure 9 shows significant separation appears between the L2 and LLC AI's after cache blocking - meaning reuse out of L2 has been achieved. The GFLOPS/s performance has improved by 16-18%, lower than 3×, because there are divide, shuffle and unpack instructions involved in the innermost loop.

Table 3. Integrated Roofline AI and performance for GPP on Haswell and KNL with and without cache blocking. Observe the 3× increase in LLC AI.

	Haswell		KNL	
gppKernel.f90:303	Original	Cache Blocked	Original	Cache Blocked
L1+NTS AI	0.31	0.31	0.64	0.64
L2 AI	0.46	0.62	0.51	0.69
LLC AI	0.46	1.40	0.56	1.71
DRAM AI	3.31	3.44	26.45	26.83
GFLOPS/s	148.57	172.48	242.55	287.28

Table 3 and Table 4 show that the (single) AI obtained from CARM is exactly the same as the L1 AI obtained in the Integrated Roofline Modeling (n.b., CARM GFlop/s is different as it was collected in a separate run that resulted in a slightly different run time). While the CARM Roofline approach, in Figure 10,

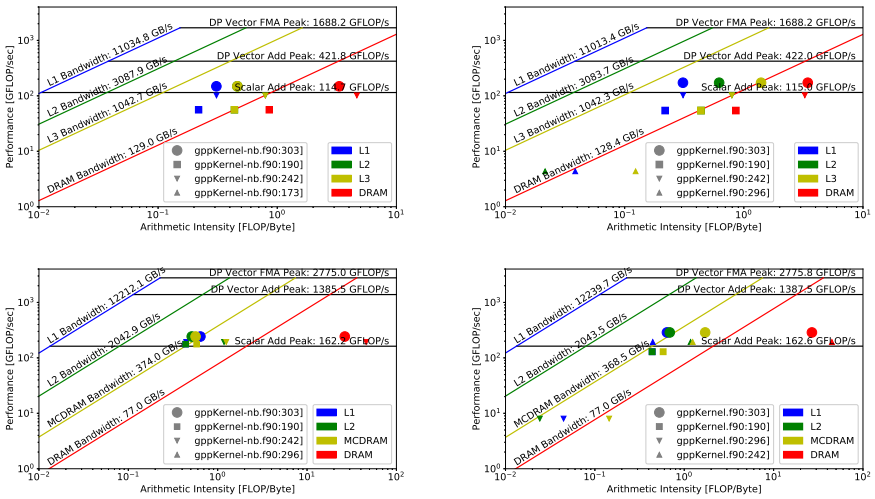

Fig. 9. Integrated Roofline for GPP on Haswell (top) and KNL (bottom) before (left) and after (right) cache blocking. Observe the lack of differentiation for L2 and L3 AI's in the left figures for loop at line 303.

can show the upward movement of the performance through optimization, it fails to provide any information on which level of cache this optimization has affected, whereas in the Integrated Roofline (Figure 9), the increase of LLC AI clearly indicates that data has been blocked to L2 and traffic between L2 and LLC has been reduced.

Table 4. CARM Roofline AI and performance for GPP on Haswell and KNL with and without cache blocking.

	Haswell		KNL	
gppKernel.f90:303	Original	Cache Blocked	Original	Cache Blocked
CARM AI	0.31	0.31	0.64	0.64
GFLOPS/s	147.60	169.83	250.88	290.03

5 Related Work

The literature is filled with many performance models and tools specialized for varying levels of detail and different bottlenecks. Whereas the Integrated Roofline presented in this paper includes all levels of the cache hierarchy, the original Roofline model [1] focused on only a single level at a time (e.g. DRAM). As accurately measuring data movement can be a hard problem (hence the cache

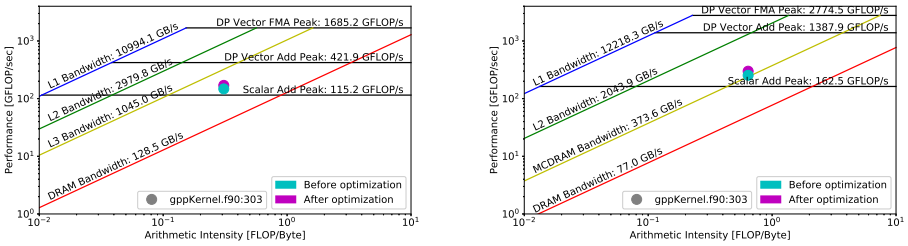

Fig. 10. CARM Roofline for GPP on Haswell (left) and KNL (right) before (cyan) and after (magenta) cache blocking.

model used in in the Integrated Roofline), the Cache-Aware Roofline Model [2] transforms the problem by fixing arithmetic intensity at the L1 and infers locality based on the position of performance relative to bandwidth ceilings.

Whereas, Roofline presents an idealized machine that can perfectly overlap computation with L1, L2, and DRAM data movement, many real processors may not be able to realize this. To that end, the Execution-Cache-Memory Model was developed [27] to more accurately capture how specific processor microarchitectures fail to perfectly overlap communication and computation.

Rather than modeling the cache hierarchy, tools like PAPI [28] and LIKWID [29] can directly read the hardware performance counters that record various compute and data movement events. Unfortunately, performance counter tools are only as good as the processor vendor's implementation of the underlying performance counters. Where the counters are inaccurate, incomplete (e.g. failing to incorporate masks when counting vector flops), or simply missing, tools will not provide the requisite data. Additionally, such tools often rely on coarse-grained sampling techniques that are error-prone on short loops or are otherwise challenged in attribution. The latter, attribution (i.e. which loop caused the data movement observed on a shared uncore counter), is a direct motivator for out cache simulator based approach. Nevertheless, performance counter sampling incurs minimal performance impact and thus enables full-application instrumentation at scale.

When performance falls out of the throughput-limited regime, overheads and inter-process communication can dominate an application's performance characteristics. To that end, depending on message size, messages per synchronization point, and the performance of the underlying communication layer, the LogP or LogGP models may be more appropriate [30,31]. For single-node runs where there is no inter-process communication, models like LogCA [32] should be adapted to incorporate the overhead of OpenMP parallelization.

Whereas our tool is nominally geared for analyzing threaded single-process executions, other tools like TAU [33] or HPCToolkit [34] are more adept at integrating and analyzing higly-concurrent, distributed-memory performance characteristics to identify communication or computational load imbalance. Simi-

larly, when running on Cray supercomputers, one may use CrayPat [35] to instrument and analyze performance. In addition to timings, CrayPat provides access (via PAPI) to the underlying performance counters in order to measure cache, memory, and floating-point performance. Nevertheless, neither PAPI, LIKWID, CrayPat, TAU, or HPCToolkit have an underlying performance model that can be used for performance analysis rather than simply performance instrumentation.

6 Summary and Outlook

We have shown in this paper that utility of an "Integrated" Roofline Model approach for evaluating and guiding the optimization of application performance. This novel approach, as implemented in the latest Intel Vector Advisor tool via an included cache simulator, provides simple to understand visual indication of which (if any) level of the cache-memory limits performance and the amount of data-reuse present in each. For example, for a given loop or function, the visual separation of the four plotted points corresponding to AI's from L1, L2, LLC, and DRAM corresponds directly to cache-reuse. We observe that the effect of cache-blocking optimizations can be easily visualized in this manner.

The integration with Intel Advisor enables the automated collection of integrated roofline information, not just for simplistic kernels but for real, large-scale applications with multiple bottlenecks. It effectively combines the process of collecting a profile and analyzing the performance limiters of the hotspots into a single step. The result is the presentation of actionable performance data for all the top hotspots in an application.

At LBNL and LANL, this information is used to inform code teams which viable optimization paths are available to them and which KNL architectures features they should target. For example, if an application bandwidth bound in certain cache or memory layer, it isn't likely very profitable to tackle improved vector code generation — but adding a layer of cache blocking or tiling (e.g. in the KNL MCDRAM) would likely enable greater performance.

In many cases, even with the increased granularity of information provided by all four points in the Integrated Roofline Model, it isn't always immediately clear what the performance limiter of an application is. For example, none of the measure performance points may lie clearly on one of their respective ceilings. In practice what we have found, is the Roofline model, and IRM in particular, are, however, always great tools for starting or framing a conversation with facility users around performance. Asking the question "why are my performance points not on their ceilings" nearly always leads to fruitful investigation whereby the code teams learns something new and deeper about their application — well beyond using walltime to profile an application alone.

In many cases, the answer to the above quoted question is related to the fact that the ceilings for each level of the cache-memory hierarchy are based on streaming (unit-stride) access patterns, but real applications often exhibit strided or random memory access patterns. It is straightforward to empirically

compute "effective" ceilings for such access patterns (which are typically lower than unit-stride access patterns) but we leave a detailed discussion of extending the utility of the Roofline model in this way to a future work.

In addition, one may, in principle, add additional levels (and corresponding performance points) to the Integrated Roofline Model corresponding to data accessed from off-node (e.g. via MPI communication), from I/O during execution, or other limiters of in-core performance (e.g. floating-point divides). This is important not only for applications that are communication or I/O bound, but also for applicaitons that depart from the multiply-add idiom that modern architectures are optimized for. This is a fruitful avenue for future work.

References

1. S. Williams, A. Waterman, D. Patterson, "Roofline: an insightful visual performance model for multicore architectures", Communications of the ACM (CACM), vol. 52, no. 4, pp. 65-76, 2009.
2. A. Ilic, F. Pratas, L. Sousa, "Cache-aware Roofline model: Upgrading the loft", IEEE Computer Architecture Letters, vol. 13, no. 1, pp. 21-24, 2014.
3. D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, Z. Matveev, "Performance analysis with cache-aware roofline model in intel advisor", High Performance Computing & Simulation (HPCS), 2017.
4. D. Doerfer, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J. Vay, H. Vincenti, "Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor", Intel Xeon Phi User Group Workshop (IXPUG), 2016.
5. Intel Advisor Roofline. `https://software.intel.com/en-us/articles/intel-advisor-roofline`
6. Intel(r) Advisor Roofline Analysis. CodeProject, February, 2017 `https://www.codeproject.com/Articles/1169323/Intel-Advisor-Roofline-Analysis`
7. How to use Intel Advisor Python. Intel Developer Zone, June, 2017. `https://software.intel.com/en-us/articles/how-to-use-the-intel-advisor-python-api`
8. T. Koskela, et. al., "Performance Tuning of Scientific Codes with the Roofline Model", Tutorial in SC, 2017. `http://bit.ly/tut160` `https://sc17.supercomputing.org/full-program/`
9. T. Koskela, et. al., "A practical approach to application performance tuning with the Roofline Model", Tutorial submitted to ISC'18.
10. Classical molecular dynamics proxy application, Exascale Co-Design Center for Materials in Extreme Environments `exmatex.org`, `https://github.com/ECP-copa/CoMD.git`
11. S. Ku, et. al., Nuclear Fusion, vol. 49 no. 11, Article 115021, 2009
12. T. Koskela, J. Deslippe, "Optimizing fusion PIC code performance at scale on cori phase two", ISC, LNCS, p. 430-440, 2017.
13. `https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview`.
14. G. Kresse, J. Furthm_ller. "Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set", Comput. Mat. Sci., 6:15, 1996.

15. http://www.vasp.at/
16. F. Wende, M. Marsman, Z. Zhao, J. Kim, "Porting VASP from MPI to MPI+OpenMP [SIMD] - Optimization Strategies, Insights and Feature Proposals, IWOMP, 2017.
17. H. Shan, S. Williams, C. Johnson, K. McElvain, W. Ormand, "Parallel Implementation and Performance Optimization of the Configuration-Interaction Method", Supercomputing (SC), 2015.
18. H. Johansen, A. Rodgers, N. Petersson, D. McCallen, B. Sjogreen, M. Miah, "Toward Exascale Earthquake Ground Motion Simulations for Near-Fault Engineering Analysis" in Computing in Science & Engineering, vol. 19, Issue 5, 2017.
19. J. Mohd-Yusof, CoDesign Molecular Dynamics (CoMD) Proxy App, LA-UR-12-21782, Los Alamos National Lab, 2012.
20. P. Cicotti, S. Mniszewski, L. Carrington, "An Evaluation of Threaded Models for a Classical MD Proxy Application", Hardware-Software Co-Design for High Performance Computing, 2014.
21. A. Adedoyin, A Case Study on Software Modernizationusing CoMD - A Molecular Dynamics Proxy Application, LA-UR-17-22676, Los Alamos National Lab, 2017.
22. D. Gunter, A. Adedoyin, Kokkos Port of CoMD Mini-App, DOE COE Performance Portability Meeting 2017.
23. T. Germann, K. Kadau, S. Swaminarayan, "369 Tflop-s molecular dynamics simulations on the petaflop hybrid supercomputer 'Roadrunner"', CCPE, 2009.
24. https://berkeleygw.org
25. https://github.com/cyanguwa/BerkeleyGW-GPP
26. J. Soininen, J. Rehr, E. Shirley, "Electron self-energy calculation using a general multi-pole approximation", Journal of Physics: Condensed Matter, 15(17), 2003.
27. J. Treibig, G. Hager, "Introducing a performance model for bandwidth-limited loop kernels", PPAM, 2010.
28. http://icl.cs.utk.edu/papi
29. https://github.com/RRZE-HPC/likwid
30. D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation", PPoPP, 1993.
31. A. Alexandrov, M. Ionescu, K. Schauser, C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model", JPDC, 1997.
32. M. Bin Altaf, D. Wood, "LogCA: A Performance Model for Hardware Accelerators", ISCA, 2017.
33. S. Shende, A. Malony, "The TAU Parallel Performance System", IJHPCA, 2005.
34. L. Adhianto , M. Fagan , M. Krentel , G. Marin , J. Mellor-crummey , N. Tallent, "HPCToolkit: Performance Measurement and Analysis for Supercomputers with Node-level Parallelism", Workshop on Node Level Parallelism for Large Scale Supercomputers, 2008.
35. http://docs.cray.com