Roofline-based Performance Efficiency of HPC Benchmarks and Applications on Current Generation of Processor Architectures

JaeHyuk Kwack, Thomas Applencourt, Colleen Bertoni, Yasaman Ghadar, Huihuo Zheng,
Christopher Knight, and Scott Parker
Argonne Leadership Computing Facility
Argonne National Laboratory
Lemont, IL 60439, USA
Email: {jkwack, tapplencourt, bertoni, ghadar, huihuo.zheng, knightc, sparker}@anl.gov

***Abstract*—The emerging pre-exascale/exascale systems are composed of innovative components that evolved from existing petascale systems. One of the most exciting evolutions is ongoing in processor architecture. In this study, we present performance results of a test suite consisting of HPC benchmarks (e.g., HPGMG, and NEKBONE) and HPC applications (e.g., GAMESS, LAMMPS, QMCPACK, and Qbox) on several processor architectures (e.g., Intel Xeon, Intel Xeon Phi, ARM, and NVIDIA GPU). For the baseline performance, we employ the Argonne Leadership Computing Facility (ALCF)'s Theta system, a Cray XC40 system that has 4,392 Intel Xeon Phi 7230 processors with a peak of 11.69 PF. We perform roofline performance analysis for the tests in the test suite and calculate their computational intensities (CI). Based on the CI values and the corresponding achievable performance peaks from the rooflines, we determine their performance efficiencies on the processor architectures.**

***Index Terms*—high performance computing; performance efficiency; roofline performance analysis; x86 processor; Arm processor; GPU;**

I. Introduction

The emerging pre-exascale/exascale systems are composed of many innovative components that evolved from existing petascale systems. One of the most exciting evolutions is ongoing in processor architecture. Intel and AMD continue to develop x86-based highly scalable processors for HPC platforms, and ARM has joined the server-class market with a RISC (Reduced Instruction Set Compute) architecture. In addition, the accelerator-based HPC platforms are dominating the TOP500 list in 2019.

In this study, we present performance test results of a test suite consisting of HPC (High Performance Computing) benchmarks (e.g., HPGMG, and NEKBONE) and HPC applications (e.g., GAMESS, LAMMPS, QMCPACK, and Qbox) on several processor architectures (e.g., Intel Xeon, Intel Xeon Phi, ARM, and NVIDIA GPU). For the baseline performance, we employ the Argonne Leadership Computing Facility (ALCF)'s Theta system, a Cray XC40 system that has 4,392 Intel Xeon Phi 7230 processors with a peak of 11.69 PF. In addition, Intel Xeon Platinum Skylake 8180M Scalable processor and ARM Marvell ThunderX2 processor are selected for the current generation of CPUs, and NVIDIA V100 GPU is used to represent the state-of-the-art accelerators for HPC platforms.

We first perform roofline performance analysis for the test suite and then categorize them according to their computational intensities (CI) (i.e., a ratio of FLOP over data movement). Based on the CI values and the corresponding achievable performance peaks from the rooflines of the processor architectures, we calculate the roofline-based performance efficiencies of the applications in the test suite.

This paper is organized as follows: Section II presents the detailed information about test-bed nodes. In Section III, we provide benchmarking results of all applications as well as the detailed build/runtime environments. Section IV presents the roofline-based performance efficiencies of all codes on test-bed nodes. In Section V, we summarize our work in this study.

II. Employed Processor Architectures

In this study, we employed four types of compute nodes - a single socket Intel Xeon Phi 7230 processor (KNL), a dual socket Intel Xeon Platinum Skylake 8180M Scalable processors (SKX), a dual socket ARM Marvell ThunderX2 processors (TX2), and a single NVIDIA V100 GPU (V100). The following subsections present the detailed information about the processor architectures as well as the measured peak flop-rates and peak memory bandwidths of the nodes.

A. Intel Xeon Phi processor - KNL

The KNL processor is architected to have up to 72 compute cores with multiple versions available containing either 64 or 68 cores as shown in Figure 1. For this paper the 64 core 7230 KNL variant is used. On the KNL chip the 64 cores are organized into 32 tiles, with 2 cores per tile, connected by a mesh network and with 16 GB of in-package multi-channel DRAM (MCDRAM) memory. The core is based on the 64-bit Silvermont microarchitecture with 6 independent out-of-order pipelines, two of which perform floating point operations. Each floating point unit can execute both scalar and vector floating point instructions including earlier Intel vector

extensions in addition to the newer AVX-512 instructions. The peak instruction throughput of the KNL architecture is two instructions per clock cycle, and these instructions may be taken from the same hardware thread. Each core has a private 32 KB L1 instruction cache and 32 KB L1 data cache. Other key features include:

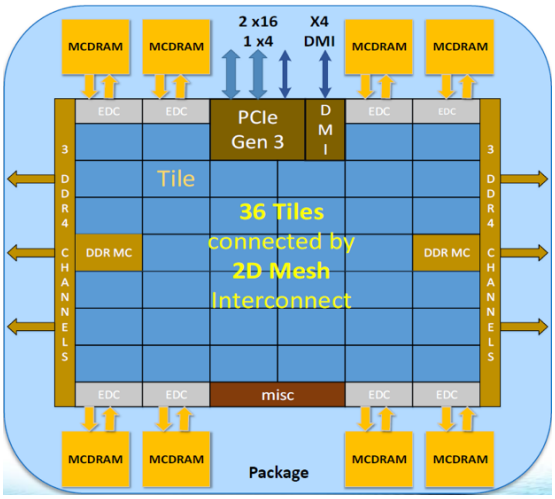


Fig. 1: KNL Processor (Credit Intel)

1) Simultaneous Multi-Threading (SMT) via four hardware threads
2) Two independent 512-bit wide floating point units, one unit per floating point pipeline that allow for eight double precision operations per cycle per unit.
3) AVX-512 vector instructions that leverages 512-bit wide vector registers with arithmetic operations, conflict detection, gather/scatter, and special mathematical operations.
4) Dynamic frequency scaling independently per tile. The fixed clock "reference" frequency is 1.3 GHz on 7230 chips. Each tile may run at a lower "AVX frequency" of 1.1 GHz or a higher "Turbo frequency" of 1.4-1.5 GHz depending on the mix of instructions it executes.

Two cores form a tile and share a 1 MB L2 cache. The tiles are connected by the Network-on-Chip with mesh topology. With the KNL, Intel has introduced on chip in-package high-bandwidth memory (IPM) comprised of 16 GB of DRAM integrated into the same package with the KNL processor. In addition to on-chip memory, two DDR4 memory controllers and 6 DDR4 memory channels are available and allow for up to 384 GB of off-socket memory. The two memories can be configured in multiple ways as shown in Figure 2:

- Cache mode - the IPM memory acts as a large direct-mapped last-level cache for the DDR4 memory
- Flat mode - both IPM and DDR4 memories are directly addressable and appear as two distinct NUMA domains
- Hybrid mode - one half or one quarter of the IPM configured in cache or flat mode with the remaining fraction in the opposite mode

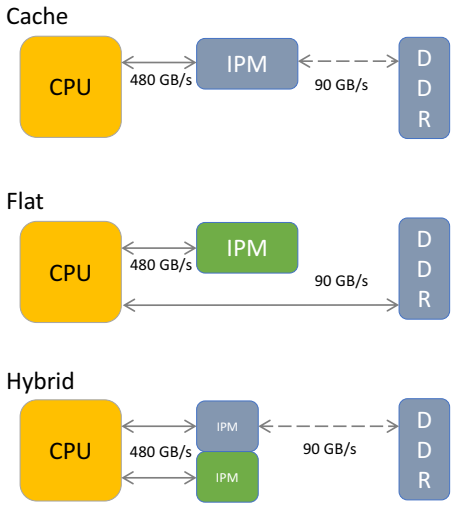


Fig. 2: Memory Modes of KNL processor

B. Intel Xeon Platinum Skylake 8180M Scalable processor - SKX

Intel Xeon Skylake Platinum 8180M, shown in Figure 3, is a 64-bit 28-core x86 multi-socket microprocessor introduced by Intel in mid-2017. The Platinum 8180M is based on the server configuration of the Skylake microarchitecture and contains 2 AVX-512 FMA units which allows 8-wide double precision vectors. The processor supports up to two hyperthreads per core and operates at 2.5 GHz with a TDP of 205 W and has a turbo boost frequency of up to 3.8 GHz. Six-channel DDR4-2666 ECC memory is supported along with up to 768GB of DDR4 memory for the standard models and up to 1.5TB for the enhanced "M" models. 38.5MB of last level L3 cache is provided along with 1 MB of L2 cache per core and 32 KB of L1 cache per core. The cores are connected by a mesh interconnect.

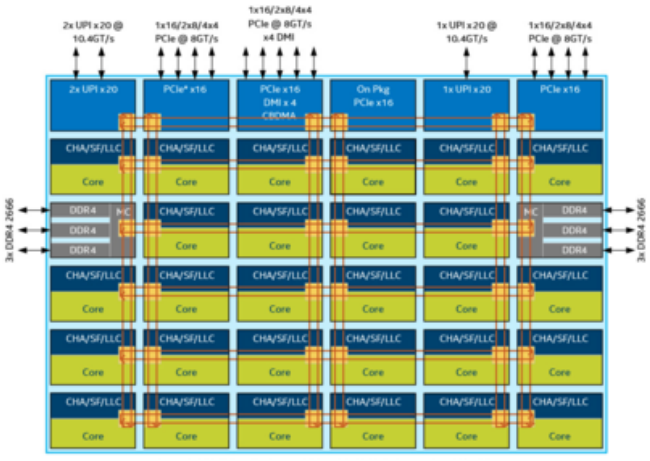


Fig. 3: SKX Microarchitecture (Credit Intel)

C. Arm Marvell ThunderX2 processor - TX2

The Marvell ThunderX2 CN9975, shown in in Figure 4, is an ARM v8.1 processor announced by Cavium in May 2018. It is available with up to 32 cores and supports 1 or 2 socket configurations. This paper utilized the 28-core variant in a 2-socket configuration. The CN9975 operates between 1.8 GHz and 2.4 GHz and supports up to eight-channel of DDR4-2666 memory.

The cores in a socket are connected by a bidirectional ring bus. Each core supports out-of-order (OOO) executions and uses fully pipelined execution units. Two NEON 128-vectors engines are available per core. Additionally two 128-bit load/store units exist which can be used to either load 2x128-bit or load 1x128 and store 1x128 per instruction. The CN9975 can support Simultaneous Multi-Threading (SMT) of up to four hardware threads. The microprocessors in this study were configured to use two hardware threads per core.

The CN9975 has a 32 KB L1 instruction and data cache along with a 256KB L2 per core. The L2 cache can load or store two cache lines simultaneously. The 32MB-L3 cache is distributed and coherent across sockets. The two sockets are connected via the Cavium Coherent Processor Interconnect technologies.

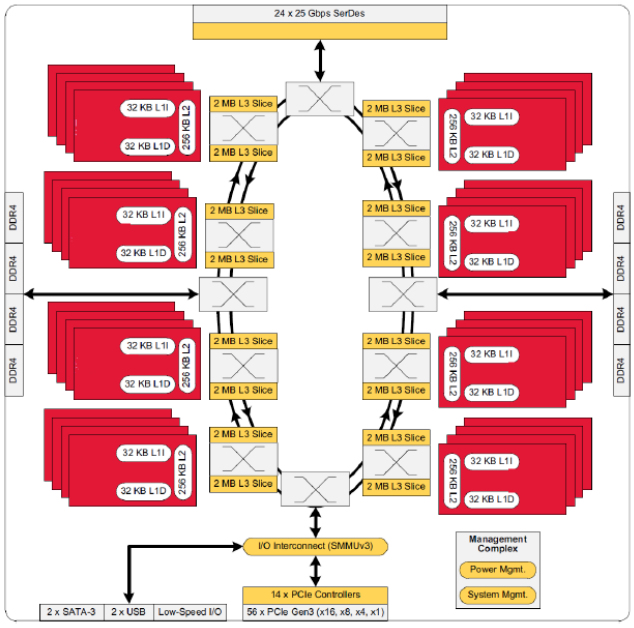


Fig. 4: Arm Marvell ThunderX2 Processor (Credit Arm)

D. NVIDIA V100 GPU - V100

The Nvidia V100 GPU, shown in Figure 5, is composed of 80 Streaming Multiprocessors (SMs), each with 32 FP64 CUDA cores which can compute 1 double-precision FMA per cycle, for a theoretical peak of 5120 double precision floating point operations per cycle. At a max clock rate of 1.53 GHz, the V100 can compute 7.8 TFlops per second. Each SM also has 64 FP32 single-precision and 64 INT32 CUDA cores, as well as 8 tensor cores, which can execute mixed precision computations, including on FP16 and FP32 values.

Threads are scheduled to run on the CUDA cores via two warp schedulers per SM. Each SM has a combined shared local memory and L1 cache of 128 KB. All 80 SMs share a 6 MB L2 cache and share 8 memory controllers. Each pair of memory controllers services a stack of HBM2, for a total of 4 stacks of HBM2 (32 GB) per V100.

Fig. 5: Nvidia V100 GPU [1]

In this work a SuperMicro SuperServer 1029GQ-TVRT was used which consisted of a dual-socket Skylake Intel Xeon Gold 6152 with 4 NVIDIA Tesla V100 SXM2 attached. The 4 GPUs are connected to each other with Nvidia's NVLink interconnect, and connected to the CPUs with PCIe 3.0. The two Xeon sockets are connected with three UPI links. [1]

E. Measured Peak Performance

The Empirical Roofline Tool (ERT) [2] was used to measure peak performance of the employed processor architectures. Table I shows peak flop-rates and memory bandwidths measured on an Intel Xeon Phi 7230 processor (KNL), dual Intel Xeon Platinum Skylake 8180M Scalable processors (SKX), dual Arm Marvell ThunderX2 processors (TX2), and an NVIDIA V100 GPU (V100). The peak flop-rate for the dual TX2 processors was measured via a DGEMM benchmark, since the ERT reported an unreasonable value for it. The ERT failed to report the L1 bandwidth of the V100 GPU and the L3 cache bandwidth of the TX2 processors. The L1 bandwidth of the V100 GPU in Table I is the theoretical peak bandwidth. The LLC data of the KNL processor represents the measured bandwidth of MCDRAM in Table I, while the LLC data of the SKX processors is the L3 cache bandwidth. The DRAM data for the V100 GPU represents the measured peak HBM2 bandwidth, not the DRAM bandwidth of the host processor. Figure 6 presents rooflines of the peak flop-rates, L1 cache bandwidths and DRAM bandwidths on the employed processors. The following compiler flags were used for the ERT measurements:

- ERT_CFLAGS for KNL: -O3 -fno-alias -fno-fnalias -xMIC-AVX512 -DERT_INTEL
- ERT_CFLAGS for SKX: -O3 -fno-alias -fno-fnalias -xCORE-AVX512 -qopt-zmm-usage=high -DERT_INTEL

- ERT_CFLAGS for TX2: -Ofast -mcpu=thunderx2t99 -fsimdmath
- ERT_CFLAGS for V100: -O3
- ERT_GPU_CFLAGS for V100: -x cu

TABLE I: Measured Peak Flop-rates and Peak Bandwidths on an Intel Xeon Phi 7230 processor (KNL), dual Intel Xeon Platinum Skylake 8180M Scalable processors (SKX), dual Arm Marvell ThunderX2 processors (TX2), and an NVIDIA V100 GPU (V100) - V100 L1 is the theoretical peak

Processor	Flop-rate (TF/s)	L1 (TB/s)	L2 (TB/s)	LLC (GB/s)	DRAM (GB/s)
Single KNL	2.13	6.46	1.911	373	78.5
Dual SKX	3.55	15.91	4.55		209
Dual TX2	0.953	3.37	2.63	1091	224
Single V100	7.83	14.336	3.35		779

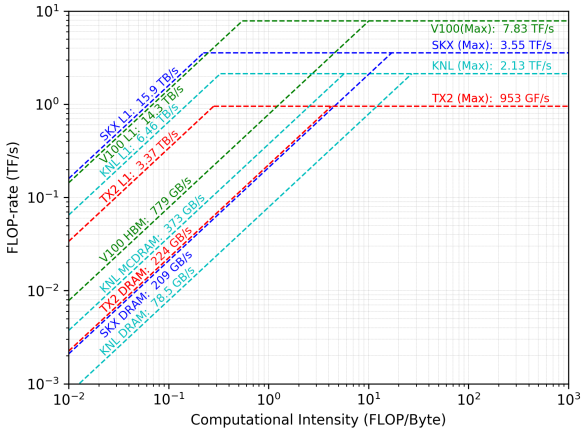


Fig. 6: Measured rooflines on an Intel Xeon Phi 7230 processor (KNL), dual Intel Xeon Platinum Skylake 8180M Scalable processors (SKX), dual Arm Marvell ThunderX2 processors (TX2), and an NVIDIA V100 GPU (V100) - V100 L1 is the theoretical peak

III. Benchmark/Application Performance

We tested two HPC benchmarks (i.e., HPGMG-FV, NEKBONE) and four production-level HPC applications (i.e., GAMESS, LAMMPS, QMCPACK, Qbox) on the processors described in the previous section. From this point, KNL, SKX, TX2 and V100 represent a single KNL 7230 processor, dual Skylake 8180M processors, dual ThunderX2 CN9975 processors, and a single V100 GPU, respectively. This section presents the detailed build/runtime environments of codes on test-beds, numerical results, and performance data as well as quick introduction of codes.

A. HPGMG-FV

HPGMG [3] is an effort for HPC performance benchmarking based on geometric multi-grid methods with emphasis on community-driven development process, long- term durability, scale-free specification and scale-free communication. It provides two implementations, Finite Element (HPGMG-FE) and Finite Volume (HPGMG-FV) implementations; HPGMG-FE is compute-intensive and cache-intensive, while HPGMG-FV is memory bandwidth- intensive. HPGMG-FV has been used for the official list.

HPGMG-FV solves an elliptic problem on isotropic Cartesian grids with fourth-order accuracy in the max norm. It calculates a flux term on each of the 6 faces on every cell in the entire domain. The fourth-order implementation [4] requires 4 times the floating-point operations, 3 times the MPI messages per smoother and 2 times the MPI message size without additional DRAM data movement compared to the second-order implementation [5] proposed originally. The solution process employs the Full Multi-grid (FMG) F-cycle that is a series of progressively deeper geometric multi-grid V-cycles. There are several dozen stencils that vary in shape and size, sweep per step, and the grid sizes vary exponentially. Coarse grid solution process can occur on a single core of a single node, and then the coarse grid solution is propagated to every thread in the system.

HPGMG has shared a MPI+OpenMP version for CPUs [6] as well as a MPI+CUDA version for NVIDIA GPUs [7] with HPC community. Both versions have been actively employed to analyze performance of modern HPC systems [8].

1) Building HPGMG-FV: A MPI+OpenMP version (commit: a0a5510) in [6] is used for KNL, SKX, and TX2, while a MPI+CUDA version (commit: 5ad473d) in [7] is employed for V100. To build HPGMG-FV codes on each processor, the following compilers and optimization flags are used:

- KNL:
  Compiler: Intel 19.0.3.199 20190206
  FLAGS: -O3 -fno-alias -fno-fnalias -xMIC-AVX512 -fopenmp
- SKX:
  Compiler: Intel 19.0.3.199 20190206
  FLAGS: -O3 -xCORE-AVX512 -qopt-zmm-usage=high -fopenmp
- TX2:
  Compiler: Arm Compiler version 19.0
  FLAGS: -Ofast -fopenmp -mcpu=native
- V100:
  Compiler: CUDA V10.0.130
  FLAGS: -O3 -x cu

2) Inputs and runtime configurations: We tested HPGMG-FV with 64^3 to 1024^3 finite-volumes. The corresponding degrees-of-freedom and their accuracy (i.e., numerical errors) are presented in Table II. The optimal MPI+OpenMP and MPI+CUDA configurations for the best performance depends on the processor architecture. After several numerical experiments, we used the optimal configurations presented in Table III.

3) Results: Table IV shows how many degrees-of-freedom HGPGMG-FV solves in a second for the given inputs and processor architectures. As reported in [8], HPGMG-FV performance linearly increases as the input size grows, and then

TABLE II: HPGMG-FV input information in detail

Number of Finite-Volumes	Multi-grid Levels	Degrees-of-Freedom	Numerical Errors
64^3	6	2.62E+05	6.93E-05
128^3	7	2.10E+06	7.45E-06
256^3	8	1.68E+07	5.14E-07
512^3	9	1.34E+08	4.15E-08
1024^3	10	1.07E+09	5.15E-09

TABLE III: HPGMG-FV runtime configurations

Processor	Number of MPI ranks	Number of Threads per MPI rank	Total Threads
KNL	64	1	64
SKX	16	7	112
TX2	16	7	112
V100	1	7	all GPU cores

it converges to certain values (see Figure 7). On KNL, we tested two memory types; one is with DRAM and the other is only with MCDRAM by using "numactl -m 1". Due to the limited memory sizes, HPGMG-FV could not solve 1024^3 finite-volumes on KNL-MCDRAM (i.e., 16GB) and V100 (i.e., 32GB), while others (i.e., KNL-DRAM, SKX and TX2) could solve it. For all inputs, V100 shows the best performance among all employed processors. For the largest problem (i.e., 1024^3), SKX shows the best performance among CPUs, while TX2 shows the best performance for the smallest problem (i.e., 64^3). For 512^3, KNL only with MCDRAM shows almost 3x performance of KNL with DRAM.

B. NEKBONE

Nekbone [9] is a mini-app derived from the Nek5000 [10] CFD code which is a high order, incompressible Navier-Stokes CFD solver based on the spectral element method. It exposes the principal computational kernels of Nek5000 to reveal the essential elements of the algorithmic-architectural coupling that are pertinent to Nek5000. Nekbone solves a standard Poisson equation in a 3D box domain with a block spatial domain decomposition among MPI ranks. The volume within

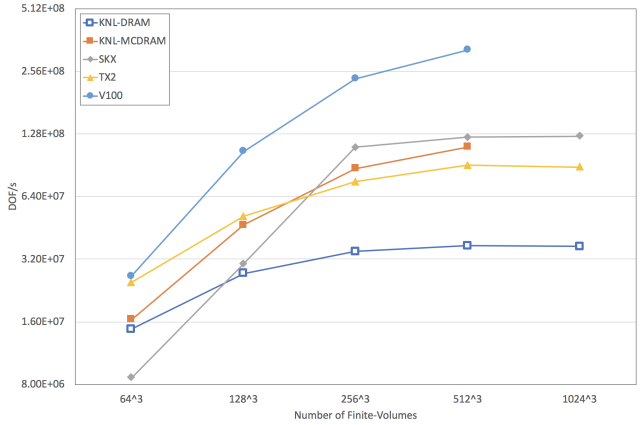


Fig. 7: HPGMG-FV performance (DOF/s) on KNL, SKX, TX2, and V100

a rank is then partitioned into high-order quadrilateral spectral elements. The solution phase consists of conjugate gradient iterations that invoke the main computational kernel which performs operations in an element-by-element fashion. Overall, each iteration consists of invoking routines performing vector operations, matrix-matrix multiply operations, nearest-neighbor communication, and MPI_Allreduce operations. The code is written in Fortran and C, where C routines are used for the nearest neighbor communication and the rest of the routines are in Fortran. It uses hybrid parallelism implemented with MPI and OpenMP. Nekbone is highly scalable and can accommodate a wide range of problem sizes, specified by setting the number of spectral elements and the number of grid points within the elements. Nekbone may be run using MPI ranks, OpenMP threads, or a combination of the two. OpenMP threading in Nekbone is coarse grained with only one parallel region spanning the entire solver. Compute load is distributed across threads in the same manner that it is done across ranks. In every iteration, a fixed set of elements to be updated are assigned to threads or ranks. Thus, the compute load and the amount of synchronization performed by OpenMP threads is nearly identical to that of MPI ranks. The number of elements per rank or thread may be load balanced by ensuring that the configured run contains a number of elements perfectly divisible by the number of ranks and threads.

For this analysis Nekbone was run with a problem consisting of a total of 8960 spectral elements and 12 gridpoints in each direction within an element. On multi-socket systems one MPI rank per socket was used and on single socket systems a single MPI rank was utilized. The number of threads was specified such that two threads per core was used on each processor. This setup produces a problem of the same size on each system with potentially a different decomposition of the work across MPI ranks and threads. Table V shows the Nekbone solver time for each processor along with the number of ranks and threads used.

C. GAMESS

GAMESS (General Atomic and Molecular Electronic Structure System) is a general quantum chemistry and *ab initio* electronic structure code.[11], [12] It has a large variety of capabilities and methods, including *ab initio* SCF energies (e.g. RHF and MCSCF), force fields (e.g., the Effective Fragment Potential), perturbative corrections to Hartree-Fock (e.g., MP2 and RI-MP2), near-linear scaling fragmentation methods (e.g., Fragment Molecular Orbital (FMO) method), *ab initio* gradients, hessians, and geometry optimizations. The majority of the code is written in Fortran, with a parallelization library which wraps MPI communication (called the Distributed Data Interface (DDI) library) written in C, and an optional C++ library with rei-mplementations of certain methods using OpenMP for CPU cores and CUDA for GPU accelerators. The original Fortran also contains OpenMP parallelism for certain methods.

When GAMESS is launched, the number of MPI ranks requested is split into two groups: half of the MPI processes are

TABLE IV: HPGMG-FV performance (DOF/s) on KNL, SKX, TX2, and V100

Processor	# of FVs 64^3	128^3	256^3	512^3	1024^3
KNL-DRAM	1.48E+07	2.73E+07	3.48E+07	3.72E+07	3.68E+07
KNL-MCDRAM	1.64E+07	4.69E+07	8.74E+07	1.11E+08	
SKX	8.61E+06	3.05E+07	1.10E+08	1.23E+08	1.25E+08
TX2	2.47E+07	5.13E+07	7.55E+07	9.04E+07	8.85E+07
V100	2.66E+07	1.06E+08	2.35E+08	3.25E+08	

TABLE V: Nekbone Solver Time (s)

Processor	Solver Time (s)	Ranks	Thds/Rank	El./Rank
KNL	17.11	1	128	8960
SKX	20.15	2	56	4480
TX2	22.07	2	56	4480

"compute processes" which perform the chemistry algorithms, and half are "data servers" which handle distributed memory and dynamic load-balancing. A typical way to run MPI-only GAMESS is to over-subscribe the cores so that each core is running one compute process and one data server.

1) Input information: Of the methods in GAMESS, we select three to investigate: RHF (energy), MP2 (energy), and RI-MP2 (energy). For benchmarking on GPUs, we only consider RI-MP2 energy. The uracil input is from the GAMESS Performance Benchmarks (https://github.com/gms-bbg/performance), with the change that "NPUNCH=0" was added to the $SCF group in the input to decrease file I/O.

2) Build information: The GAMESS calculations were done using commit $43d24fd$ of the development branch of GAMESS.

For SKX, we used Intel 2019 compilers and MPI (19.0.3.199). The standard GAMESS build procedure was used, with GMS_DEBUG_FLAGS=-xCOMMON-AVX512, and the MKL math library was linked in. Due to compiler errors, only the '-O0' flag was passed during compilation for files mpcgrd.src and mpcmsc.src. For OpenMP runs, we also compiled with "-qopenmp". For MPI, flags "I_MPI_PIN_PROCESSOR_LIST=all:map=scatter and I_MPI_HYDRA_ENV=all" were used for MPI-only runs, and for "I_MPI_PIN=enable and I_MPI_HYDRA_ENV=all I_MPI_PIN_DOMAIN=omp" MPI+OpenMP runs. The MPI-only runs were launched with 56 compute processes and 56 data servers (112 MPI ranks), and the MPI+OpenMP runs were launched with 1 compute process, 1 data server, and 112 OpenMP threads.

For KNL, the standard GAMESS build procedure was used, with cray-xc as the target architecture. The -xMIC-AVX512 flag is used during compilation by default. We used Intel 2018 compilers (18.0.0.128) and the MKL math library. For MPI, we used Cray MPI (version 7.7.3). On Theta, the following modules were loaded: craype/2.5.15, PrgEnv-intel/6.0.4, craype-mic-knl, and cray-mpich/7.7.3, and the node was booted into cache-quad mode. The MPI-only runs were launched with 64 compute processes and 64 data servers, and the MPI+OpenMP runs were launched with 1 compute process, 1 data server, and 256 OpenMP threads.

For TX2, we used Arm Allinea Studio 19.1. The build scripts were modified to compile DDI with armclang, and to compile the Fortran source of GAMESS with armflang. For armflang, we used the flags "-i8 -O3 -mcpu=native" and statically linked with the Arm Performance library with "$ARMPL_DIR/lib/libarmpl_ilp64.a" in the link line. For OpenMP runs, we also compiled with "-fopenmp". For MPI, we used MVAPICH2 version 2.3 with flags "MV2_USE_BLOCKING=1, MV2_USE_THREAD_WARNING=0, MV2_ENABLE_AFFINITY=0" for MPI-only runs and "MV2_ENABLE_AFFINITY=0" for MPI+OpenMP runs.

For all OpenMP runs, the stack size of the threads created by the OpenMP runtime were set with "OMP_STACKSIZE=20M".

For V100, we built GAMESS with libcchem, using GNU 4.8.5 and CUDA 9.0.176. The RI-MP2 file was modified slightly to use libcchem as the RI-MP2 method.

For detailed changes from the released version of the GAMESS compilation scripts, contact the authors.

3) Benchmarks and Results: Table VI shows the time to solution for RHF, MP2, and RI-MP2 for the test input (uracil).

TABLE VI: GAMESS time to solution (s) on various architectures

Method	KNL	SKX	TX2	V100
RHF (MPI-only)	467.0	127.1	169.7	-
MP2 (MPI-only)	753.3	209.4	302.8	-
RI-MP2 (MPI-only)	539.7	146.2	217.4	-
RI-MP2 (MPI+OpenMP)	506.1	106.7	177.0	85.2

Using a binary instrumentation tool, we measured the FLOP on SKX for the MPI+OpenMP RI-MP2 calculation that is 9618.88 Gflops.

4) Discussion: As shown in Table VI, the time-to-solution on TX2 is about 1.3x to 1.6x slower than on SKX.

For the MPI-only runs, we note that because of how MPI is used in GAMESS (typically oversubscribed with half of the MPI ranks as "compute processes" and the other half as "data servers"), the specific MPI and available flags in that MPI can have an effect on performance. In Intel MPI 2019, the flag I_MPI_WAIT_MODE was removed. This flag has an effect on GAMESS performance when oversubscribing cores, since it allows the data servers to wait for messages instead of polling the fabric. When Intel MPI 2017 was used, we see the following timings in Table VII.

TABLE VII: GAMESS performance comparison of Intel MPI 2017 and Intel MPI 2019 on SKX

Method	Time (s) with Intel 2017	% of Intel 2019 time
RHF (MPI-only)	77.3	60.82%
MP2 (MPI-only)	127.6	60.94%
RI-MP2 (MPI-only)	89.7	61.35%
RI-MP2 (MPI+OpenMP)	102.1	95.69%

D. LAMMPS

LAMMPS is a molecular simulation code commonly used for modeling various states of matter (liquids, surfaces, solids, biopolymers) and supports multiple physical models, particle types, and sampling methods.[13], [14] LAMMPS is a highly modular and extensible code written in C/C++ and parallelized with MPI+X, where X includes multiple options and is incorporated as a set of add-on packages separate from the core LAMMPS code. MPI parallelization is enabled via spatial domain decomposition whereby the simulated system is partition into separate sub-domains assigned to MPI ranks. Additional parallelism via programming model X (OpenMP, CUDA/OpenCL, Kokkos, explicit vectorization) is then used to further reduce the time-to-solution typically via parallelization over particles within a sub-domain (e.g. force-decomposition methods).

1) Building LAMMPS: An unaltered version of LAMMPS, 19Feb19, was downloaded as a tarfile from the LAMMPS website and used for analysis of the reactive forcefield ReaxFF using the DOE CORAL-2 LAMMPS benchmark.

a) CPU Architecture: To enable simulations with ReaxFF models, the base USER-REAXC package was installed along variants from the USER-OMP and KOKKOS packages for additional performance. The Makefiles packaged with LAMMPS were used with appropriate modifications for the respective hardware. For KNL, the Makefile available in src/MAKE/OPTIONS/Makefile.knl was used as-is. For reference, the following compiler optimization flags were used: '-xMIC-AVX512 -O2 -fp-model fast=2 -no-prec-div -restrict -DLAMMPS_MEMALIGN=64' For SKX, the same Makefile could be used replacing the instruction set flag with -xCORE-AVX512 and adding -qopt-zmm-usage=high. For TX2, the following compiler flags were used: '-Ofast -fopenmp -mcpu=native'. For builds with Kokkos, the KOKKOS_ARCH variable was set to 'KNL', 'SKX', and 'ARMv8-TX2' for the respective hardware and KOKKOS_DEVICE was set to 'OpenMP'.

b) GPU Architecture: The ReaxFF model in LAMMPS is only supported on GPUs via the KOKKOS package. To compile LAMMPS with Kokkos for V100, the base Makefile src/MAKE/OPTIONS/Makefile.kokkos_cuda_mpi was updated to set KOKKOS_ARCH as 'Volta70' and KOKKOS_DEVICE as 'Cuda,OpenMP'.

c) LAMMPS Inputfile: The LAMMPS ReaxFF use-case examined here is the same HNS input provided as a CORAL-2 benchmark. The number of particles selected for this study was 36,480 ($\sim$ 300 particles per thread; 6x5x4 replica of input configuration) and is representative of weak-scaled science runs that utilize substantial fractions of Theta at ALCF. The same system was used for runs on all hardware investigated.

2) LAMMPS Results: For the CPU runs on KNL, SKX, and TX2, both the OpenMP and Kokkos parallelized variants of the reax/c model were used in a series of runs with various combinations of MPI ranks and OpenMP threads-per-rank. Because of additional optimizations in the USER-OMP package for reax/c, the multi-threaded version even with a single thread generally has reduced time-to-solutions compared to the MPI-only version. For the GPU runs, the reax/c model is only enabled using Kokkos (via the CUDA backend) and there is generally not a benefit (today) for having multiple MPI ranks per GPU (though we still tested multiple ranks per GPU). For each type of hardware examined, the run that yielded the best (smallest) time-to-solution has been reported. For KNL, 32 MPI ranks with 4 OpenMP threads per rank yielded the best time-to-solution of 220.70 ms/step. For SKX, 28 MPI ranks with 4 OpenMP threads per rank resulted in the best time-to-solution of 98.50 ms/step, roughly a factor of 3x faster than KNL. For TX2, 14 MPI ranks with 8 OpenMP threads returned the best time-to-solution of 172.78 ms/step, which is $\sim$28% improvement compared to KNL. In general, using two hardware threads per core for the CPU runs yielded the lowest time-to-solution compared to using one (or more) hardware threads.

On V100, the lowest time-to-solution was achieved using 1 MPI ranks at 50.42 ms/step. We have also measured FLOP on SKX and KNL architectures.

Table VIII shows the time-to-solution for Reax/c model for Pure HNS crystal.

TABLE VIII: Time per step (ms) on various architectures for LAMMPS runs with ReaxFF

Model	KNL	SKX	TX2	V100
Reax/C	220.70	98.50	172.78	50.42

3) LAMMPS Discussion: In this study we have looked at the performance in terms of time to solution of reactive force field within LAMMPS application. Currently, one can use both MPI with OpenMP threading on CPU architecture or use KOKKOS with CPU or CUDA backend with USER_REAXC package. Our results indicated that overall among plain CPU (using MPI with OpenMP threading) KOKKOS with CPU backend and KOKKOS with CUDA backend, KOKKOS with CUDA backend is about 10% compare to time on SKX. One of the most time consuming segments of molecular dynamic simulations in LAMMPS is time spent in pair and neighbor list calculations. We have list the time per second on each architecture for comparison. As it is shown in Table IX pair calculations takes about 123 millisecond per step on TX2 while it takes 16.70 millisecond on V100. The same trend is observed for Neighbor list formation. If one compare the time for pair calculations vs neighbor list on each architecture, pair calculations is more expensive.

TABLE IX: Time per step (millisecond) on various architectures for Pair and Neighbor calculations - LAMMPS

Kernel	KNL	SKX	TX2	V100
Pair	151.72	64.71	123.69	16.70
Neighbor list	4.70	1.57	2.49	0.92

E. QMCPACK

QMCPACK (https://qmcpack.org/) [15] is an open source quantum Monte Carlo package for ab-initio electronic structure calculations. It supports calculations of metallic and insulating solids.

QMCPACK uses a Metropolis Monte Carlo algorithm which generates samples sequentially via a random walk along a Markov chain. Each OpenMP thread execute independent Markov chains or walkers. After each walker has completed a number of steps, the simulation is completed. Hence, the more worker you have, the more computation you do.

Our figure of merit (FOM) measures how many walkers have be moved in one second.

In a case of dual socket systems (i.e., SKX and TX2), the FOM is computed on one socket and multiplied by two. This is done to avoid any NUMA effect and to reduce the time needed by a binary instrumentation tool to collect the FLOP count.

1) Compilation: QMCPACK v3.7.0 is used. CMake options were used to enable timers (–DENABLE_TIMERS=1) and enable support for Instrumentation and Tracing Technology (ITT) API (-DUSE_VTUNE_API=1). Other than that, default option was used.

For SKX and KNL, we compiled the code using icc 2019 with '-O3 -xCOMMON-AVX' flags. For TX2, we used armclang with '-O3 -mcpu=native -ffast-math' flags. Both were linked against the optimized math libraries from the vendor (Arm Performance Libraries for Arm and Intel Math Kernel Library for Intel).

2) Input: The QMCPACK manual can be found at https://docs.qmcpack.org/qmcpack_manual.pdf.

The simulated system is known as S32 and consist of a 32 repeats of a NiO primitive cell leading to 128 atoms and 1536 electrons. The system is a part of the QMCPACK standard benchmark suite (see section 3.9.3 of the manual). The input file used can be found directly in the QMCPACK repository qmcpack/tests/performance/NiO/sample/dmc-a128-e1536-cpu/NiO-fcc-S32-dmc.xml. We disabled that last DMC block for this papers.

3) Threading: One thread per core is used where used for all the architecture. This is achieved by using numactl, OMP_PLACES and OMP_PROC_BIND with the adequate option. 'lstopo' and the QMCPACK binary 'qmc-check-affinity' were used to verify the correctness of the binding.

4) Result: When QMCPACK is run with the '–enable_timefines' option, multiple timers are presented in the output. We report the time spend in the 'DMC' section. Indeed, in real world usage, this section takes the vast majority of the runtime.

TABLE X: QMCPACK FOM measurement

	DMC Time	Walker	Socket	FOM
KNL	65.01	64	1	0.98
SKX	16.173	28	2	3.43
TX2	57.52	28	2	0.97

In Table X, TX2 shows similar performance to KNL. SKX shows around 3x speed-up compared to KNL and TX2. The FLOP counts in Table XI correspond to DMC part of QMCPACK.

TABLE XI: QMCPACK FLOP measurements

	Walker	GFLOPS	GFLOPS/walker
KNL	64	19233.86	300.5
SKX/TX2	28	8326.73	297.4

5) Effect of SoA (Structure-of-Array): The performance of QMCPACK has been improved by adopting SoA instead of AoS (Array-of-Structure). Since the SoA approach improves data cache hit ratio, the performance gain by SoA depends on the data cache performance. Table XII shows QMCPACK DMC time with AoS (without SoA) and the speed-up of SoA over AoS on SKX and TX2. The speedup by SoA is much higher on SKX than on TX2, because the data cache performance of SKX is much better than the cache performance of TX2, as seen in Table I.

TABLE XII: QMCPACK DMC Time with AoS and Speed-up of SoA over AoS

	Walker	DMC Time (AoS)	SpeedUp of SoA over AoS
SKX	28	53.35	3.3×
TX2	28	82.23	1.43×

F. Qbox

Qbox (http://qboxcode.org) is a C++ MPI/OpenMP scalable parallel implementation of first-principles molecular dynamics based on the plane-wave, pseudopotential density functional theory formalism. It uses FFTW for 3D Fast Fourier Transformation and ScaLAPACK for parallel dense linear algebra. The code has been running at large scale (10k - 100k) on various supercomputers, including ALCF Mira and Theta, NERSC Cori.

1) Setup of the benchmarks: In this study, we focus on the performance at a single node level on SKX, KNL, and TX2. The benchmarks are based on rel1_66_2 public release of Qbox (https://github.com/qboxcode/qbox-public). We built Qbox with the following compiler flags.

- SKX: mpiicc -xCORE-AVX512 -align -fp-model fast=2 -no-prec-div -g -qopenmp -O3
- KNL: CC -xMIC-AVX512 -align -fp-model fast=2 -no-prec-div -g -qopenmp -O3
- TX2: mpicxx -g -Ofast -mcpu=native

We linked the code to the vendor provided libraries for FFT and ScaLAPACK: MKL on SKX and KNL, and Arm Performance library on TX2.

The physical system we choose for our benchmarks is a silicon carbide periodic solid which contains 64 atoms (32 silicon and 32 carbon atoms) and 256 electrons. We perform the ground state calculation using PBE0 hybrid functional. In order to guarantee the same work load across different runs, we fix the total number of self-consistent iterations to be 5.

We set OMP_NUM_THREADS=1, and 1 MPI rank per core on all the architectures. MPI processes are arranged in a two dimensional array (8×7 on SKX and TX2, 8×8 on KNL).

On KNL, we used the cache-quadrant memory mode. On TX2, we specified -bind-to socket.

2) Results and discussion: Table XIII shows the time-to-solution including breakup time spent on the three major functions. The dominant function is *exc* which computes the Hartree-Fock exact exchange between pairs of electrons. This involves N^2 of small 3D Fourier transformations (on a $66 \times 66 \times 66$ grid in this case), where N is the number of orbitals (N=128 in this case). As one could see the performance ratio between SKX, KNL and TX2 is 1:0.56:0.89.

TABLE XIII: Qbox time to solution (second) on various architectures

Kernel	KNL	SKX	TX2
exc	24.15	16.76	19.278
hpsi	2.06	0.47	0.74
wf_update	1.63	0.40	0.38
Total Walltime	33.76	18.94	21.32

The FLOP is measured to be 997.18 GFlops.

G. Per-node/per-watt performance comparison

Figure 8 presents per-node application performance on KNL, SKX, TX2 and V100 over KNL application performance data. The performance on V100 is the best in all applications executed on V100 (i.e., HPGMG, GAMESS, LAMMPS). Among CPU architectures, SKX shows the best performance for all applications except NEKBONE.

Figure 9 provides per-watt application performance on KNL, SKX, TX2 and V100 over KNL data. Thermal Design Power (TDP) data in Table XIV were used as reference powers for the employed processor architectures. For highly vectorized proxy applications (i.e., HPGMG and NEKBONE), it turns out the per-watt performance on KNL is better than on SKX and TX2. Due to the high power efficiency of GPUs, the performance difference between V100 and CPUs becomes more significant in Figure 9 than in Figure 8. Because of lower power consumption of TX2 than SKX, the difference in per-watt performance between SKX and TX2 comes to be less than in per-node performance.

TABLE XIV: Thermal Design Powers (TDPs) of KNL, SKX, TX2 and V100

	Watt/socket	Watt/node
KNL	215	215
SKX	205	410
TX2	170	340
V100	250	

IV. Roofline-based Performance Efficiency

The roofline analysis model [16] is a powerful tool for HPC code developers since it provides visually intuitive plots for the performance of their applications and kernels in terms of computational intensity (CI) and flop-rate. Intel Advisor started providing a useful cache-aware roofline analysis model [17] in a handy way since its version 2017 update 2; however, it is not widely compatible with other compilers (e.g., PGI, CRAY, GNU, ARM, LLVM) and other processor-architectures (e.g., AMD, ARM, GPU), so general roofline analysis technique has been investigated [18] [19]. In this study, we measured FLOP with a binary instrumentation tool, and it was verified with manually calculated FLOP data from NEKBONE. The difference between the measured data and the calculated data was less than 4%. The data transfers were measured with hardware performance counters via the Linux perf command [20].

TABLE XV: Measured FLOP and data transfer, and computational intensity

	GFLOP	Memory Read/Write (GiB)	Memory-based Computational Intensity
HPGMG-FV	13303.9	13440.0	0.99
NEKBONE	2666.6	3838.0	0.69
GAMESS	9618.9	548.1	17.55
LAMMPS	4997.3	32075.7	0.16
QMCPACK	16653.5	3038.8	5.48
Qbox	997.2	2913.2	0.34

Table XV shows the measured FLOP and memory read/write data as well as the corresponding memory-based computational intensities (CIs). Figures 10a,10b and 10c are applications' flop-rates and CIs under rooflines of KNL, SKX and TX2. Using the CIs, the roofline-based peak flop-rates of all applications were computed as presented in Table XVI. The performance efficiencies in Table XVI are the ratio of measured flop-rates over roofline-based peak flop-rates. For KNL, we used MCDRAM bandwidth to compute roofline-based peak flop-rates of applications. Since HPGMG-FV and NEKBONE are well-optimized for MCDRAM that is smaller but faster than DRAM, their performance efficiencies on KNL are much higher than other production-level applications (i.e., GAMESS, LAMMPS, QMCPACK and Qbox).

Table XVII and Figure 11 show relative performance efficiency based on KNL performance efficiency. While HPGMG-FV and NEKBONE show relatively similar performance efficiencies on KNL, SKX and TX2, GAMESS shows higher efficiency on TX2, and LAMMPS, QMCPACK and Qbox show higher efficiency on SKX.

V. Concluding Remarks

We executed performance tests for two HPC benchmarks and four production-level HPC applications on compute nodes with a single Intel Xeon Phi 7230 processor (KNL), dual Intel Skylake 8180M processors (SKX), dual Arm Marvell ThunderX2 processors (TX2), and a single NVIDIA V100 GPU (V100).

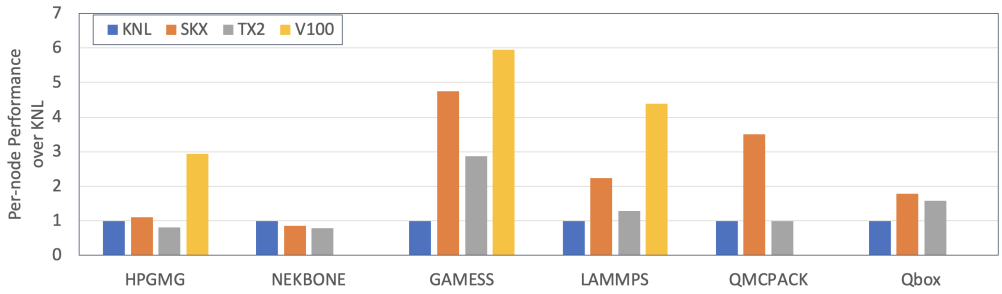


Fig. 8: Per-node application performance comparison on KNL, SKX, TX2 and V100

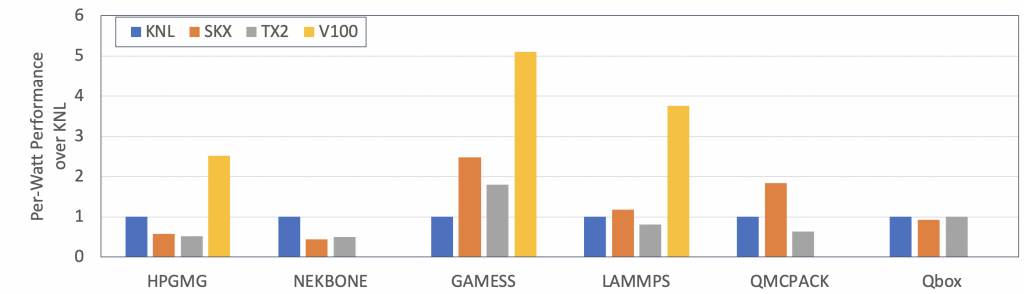
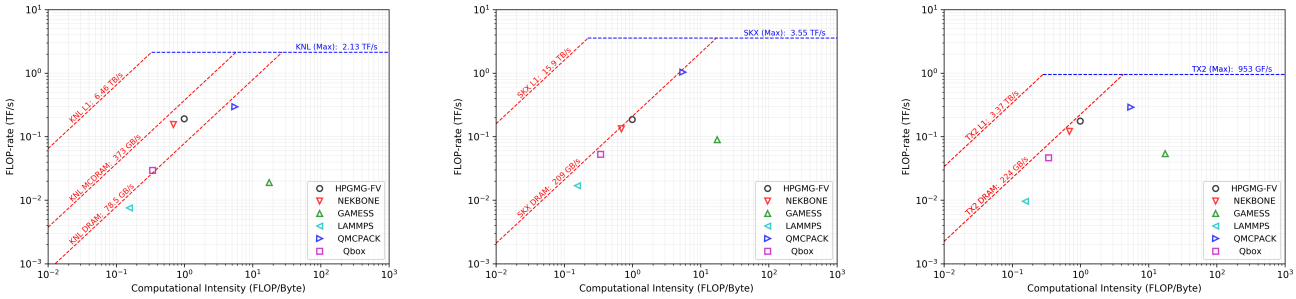


Fig. 9: Per-watt application performance comparison on KNL, SKX, TX2 and V100



(a) KNL (b) SKX (c) TX2

Fig. 10: Roofline Performance Analyses on KNL, SKX and TX2

We first investigated the optimal configurations for benchmark/application codes based on the best wall times on each test-bed node. With the optimal configurations, we compared per-node application performances on KNL, SKX, TX2, and V100. As shown in Figure 12a, V100 performance is the best among all processors, and SKX performance is the best among all CPU processors. In per-watt performance comparison on average in Figure 12b, the performance gap between V100 and SKX becomes more notable than in Figure 12a, while the per-watt performance difference between SKX and TX2 becomes less significant than the difference in per-node performance.

For roofline performance analysis on CPUs (i.e., KNL, SKX and TX2), we measured the peak flop-rates and peak memory bandwidths of the CPU nodes. FLOP and data transfers of six HPC benchmarks/applications were measured via a binary instrumentation tool and hardware performance counters, respectively. Using the performance data (i.e., FLOP and data transfer) of each application, we computed computational intensity of each application. The roofline-based peak performance of each application was calculated based on the CI value. The roofline-based performance efficiency of each application was evaluated by the ratio of the measured flop-rate over the roofline-based peak flop-rate. The HPC benchmark codes such as HPGMG-FV and NEKBONE show relatively similar performance efficiency on KNL, SKX and TX2, while production-level codes show relatively higher performance efficiencies on either SKX or TX2. On average, the roofline-based performance efficiency on SKX and TX2 are similar to each other, as shown in Figure 12c.

In this study, we evaluated performance characteristics of the current generation of processor architectures with six important HPC benchmark/application codes. It turns out the roofline-based performance efficiency provides us with interesting performance features of applications on multiple

TABLE XVI: Roofline-based performance efficiency on KNL, SKX and TX2

	KNL			SKX			TX2		
	FLOP-rates (GFLOP/s)	Peak (GFLOP/s)	Efficiency (%)	FLOP-rates (GFLOP/s)	Peak (GFLOP/s)	Efficiency (%)	FLOP-rates (GFLOP/s)	Peak (GFLOP/s)	Efficiency (%)
HPGMG-FV	191.5	369.2	51.9%	186.0	206.9	89.9%	176.9	221.7	79.8%
NEKBONE	155.9	259.2	60.1%	132.3	145.2	91.1%	120.8	155.6	77.6%
GAMESS	19.0	2130.0	0.9%	90.1	3550.0	2.5%	54.3	953.0	5.7%
LAMMPS	7.5	58.1	13.0%	16.9	32.6	51.9%	9.6	34.9	27.6%
QMCPACK	295.86	2044.2	14.5%	1029.7	1145.4	89.9%	289.5	953.0	30.4%
Qbox	29.5	127.7	23.1%	52.6	71.5	73.6%	46.8	76.7	61.0%

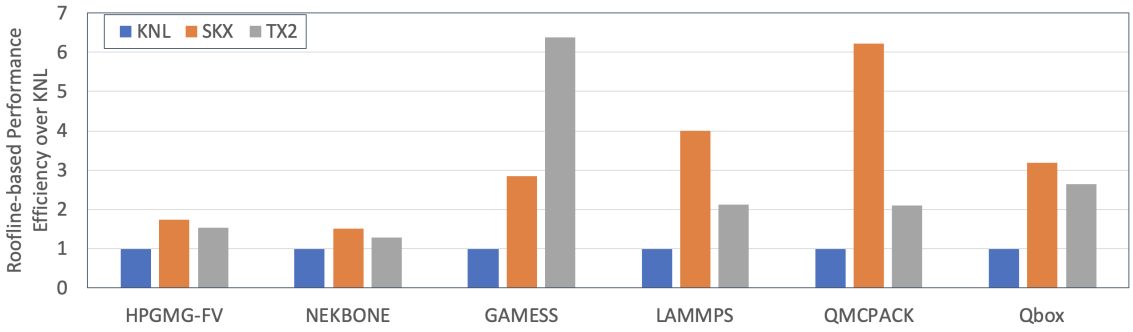


Fig. 11: Relative Roofline-based Performance Efficiency over KNL Efficiency

TABLE XVII: Relative Roofline-based Performance Efficiency over KNL Efficiency

	KNL	SKX	TX2
HPGMG-FV	1.00	1.73	1.54
NEKBONE	1.00	1.52	1.29
GAMESS	1.00	2.85	6.39
LAMMPS	1.00	4.00	2.13
QMCPACK	1.00	6.21	2.10
Qbox	1.00	3.18	2.64

platforms. We plan to perform roofline-based performance efficiency tests on a larger number of applications to reflect the spectrum of ALCF workloads on the next generation of processor architectures.

ACKNOWLEDGMENT

This Work was supported by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

REFERENCES

[1] NVIDIA, "Nvidia tesla v100 gpu architecture," 2017. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[2] "Empirical Roofline Tool Web page," https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/software/ert/, 2019. [Online]. Available: https://crd.lbl.gov/departments/computer-science/PAR/research/roofline/software/ert/

[3] "HPGMG Web page," https://hpgmg.org, 2019. [Online]. Available: https://hpgmg.org

[4] S. Williams, "4th order hpgmg-fv implementation," *HPGMG BoF, Supercomputing*, 2015.

[5] M. Adams, J. Brown, J. Shalf, B. Straalen, E. Strohmaier, and S. Williams, "Hpgmg 1.0: A benchmark for ranking high performance computing systems," *LBNL Technical Report, LBNL 6630E*, 2014.

[6] "HPGMG Github," https://github.com/hpgmg/hpgmg, 2019. [Online]. Available: https://github.com/hpgmg/hpgmg

[7] "HPGMG-CUDA Bitbucket," https://bitbucket.org/nsakharnykh/hpgmg-cuda.git, 2019. [Online]. Available: https://bitbucket.org/nsakharnykh/hpgmg-cuda.git

[8] J. Kwack and G. H. Bauer, "HPCG and HPGMG benchmark tests on multiple program, multiple data (MPMD) mode on Blue Watersa Cray XE6/XK7 hybrid system," *Concurrency Computat: Pract Exper.*, 2017. [Online]. Available: https://doi.org/10.1002/cpe.4298

[9] "Nekbone repository," https://asc.llnl.gov/CORAL-benchmarks/.

[10] P. Fischer, J. Lottes, D. Pointer, and A. Siegel, "Petascale algorithms for reactor hydrodynamics," *Journal of Physics: Conference Series*, 2008.

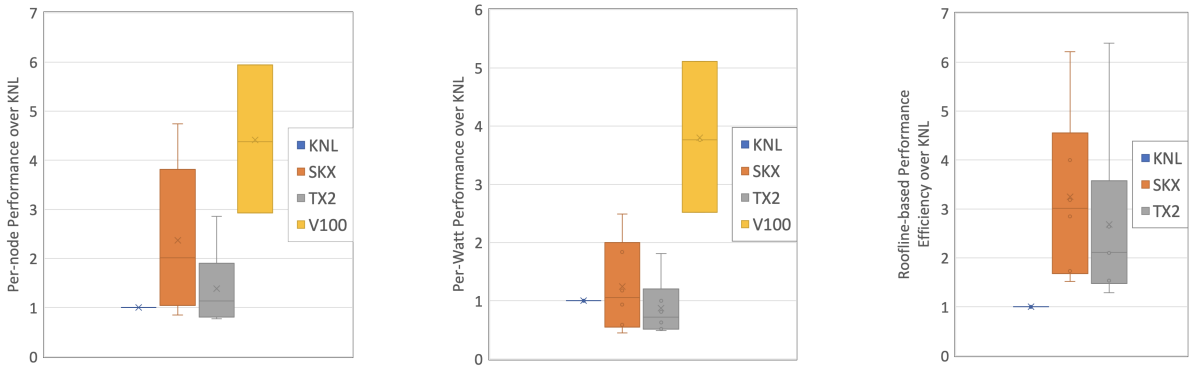
[11] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, and J. A. Montgomery, "General atomic and molecular electronic structure system," *Journal of Computational Chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.540141112

[12] M. S. Gordon and M. W. Schmidt, "Chapter 41 - advances in electronic structure theory: Gamess a decade later," in *Theory and Applications of Computational Chemistry*, C. E. Dykstra, G. Frenking, K. S. Kim, and G. E. Scuseria, Eds. Amsterdam: Elsevier, 2005, pp. 1167 – 1189.

[13] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, pp. 1–19, 1995. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002199918571039X

[14] "LAMMPS Web page," https://lammps.sandia.gov, 1995. [Online]. Available: https://lammps.sandia.gov

[15] J. Kim, A. D. Baczewski, T. D. Beaudet, A. Benali, M. C. Bennett, M. A. Berrill, N. S. Blunt, E. J. L. Borda, M. Casula, D. M. Ceperley, S. Chiesa, B. K. Clark, R. C. Clay, K. T. Delaney, M. Dewing, K. P. Esler, H. Hao, O. Heinonen, P. R. C. Kent, J. T. Krogel, I. Kylänpää, Y. W. Li, M. G. Lopez, Y. Luo, F. D. Malone, R. M. Martin, A. Mathuriya, J. McMinis, C. A. Melton, L. Mitas, M. A. Morales, E. Neuscamman, W. D. Parker, S. D. P. Flores, N. A. Romero, B. M. Rubenstein, J. A. R. Shea, H. Shin, L. Shulenburger, A. F. Tillack, J. P. Townsend, N. M. Tubman, B. V. D. Goetz, J. E. Vincent, D. C. Yang, Y. Yang, S. Zhang, and L. Zhao, "QMCPACK: an open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids," *Journal of Physics: Condensed*



(a) Per-node performance (b) Per-watt performance (c) Roofline-based efficiency

Fig. 12: On average per-node/-watt performance comparisons and roofline-based performance efficiency comparison over KNL

Matter, vol. 30, no. 19, p. 195901, apr 2018. [Online]. Available: https://doi.org/10.1088/1361-648x/aab9c3

[16] S. Williams, A. Waterman, and A. Patterson, "Roofline: an insightful visual performance model for floating-point programs and multicore architectures," *Commun ACM.*, vol. 53, pp. 65–76, 2009.

[17] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: upgrading the loft," *IEEE Comput Archit Lett.*, vol. 13, pp. 21–24, 2014.

[18] J. Kwack, G. Arnold, C. Mendes, and G. H. Bauer, "Roofline analysis with Cray performance analysis tools (CrayPat) and roofline-based performance projections for a future architecture," *Concurrency Computat Pract Exper.*, 2018. [Online]. Available: https://doi.org/10.1002/cpe.4963

[19] "General Roofline Evaluation Gadget Webpage," https://github.com/ncsa/GREG, 2019. [Online]. Available: https://github.com/ncsa/GREG

[20] "Perf Wiki-page," https://perf.wiki.kernel.org/index.php/Main_Page, 2019. [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page