Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor

Douglas Doerfler, Jack Deslippe, Samuel Williams, Leonid Oliker, Brandon Cook, Thorsten Kurth, Mathieu Lobet, Tareq Malas, Jean-Luc Vay, and Henri Vincenti

Lawrence Berkeley National Laboratory
{dwdoerf, jrdeslippe, swwilliams, loliker, bgcook, tkurth, mlobet, tmalas, jlvay, hvincenti}@lbl.gov

Abstract The Roofline Performance Model is a visually intuitive method used to bound the sustained peak floating-point performance of any given arithmetic kernel on any given processor architecture. In the Roofline, performance is nominally measured in floating-point operations per second as a function of arithmetic intensity (operations per byte of data). In this study we determine the Roofline for the Intel Knights Landing (KNL) processor, determining the sustained peak memory bandwidth and floating-point performance for all levels of the memory hierarchy, in all the different KNL cluster modes. We then determine arithmetic intensity and performance for a suite of application kernels being targeted for the KNL based supercomputer Cori, and make comparisons to current Intel Xeon processors. Cori is the National Energy Research Scientific Computing Center's (NERSC) next generation supercomputer. Scheduled for deployment mid-2016, it will be one of the earliest and largest KNL deployments in the world.

1 Introduction

Moving an application to a new architecture is a challenge, not only in porting of the code, but in tuning and extracting maximum performance. This is especially true with the introduction of the latest manycore and GPU-accelerated architectures, as they expose much finer levels of parallelism that can be a challenge for applications to exploit in their current form. To address this challenge, NERSC has established a collaborative partnership with code teams porting their codes to Cori and its Intel Knights Landing (KNL) processors. Called NESAP (NERSC Exascale Science Applications Program), this collaborative will partner the code teams with key personal from NERSC, Cray, and Intel [13,15].

One method being used within NESAP to identify and better understand the fundamental architectural bottlenecks, and hence providing a path to better understand where to focus optimization efforts, is to develop a Roofline Performance Model (Roofline) for KNL [23]. We find that the roofline model provides an important framework for the optimization conversation with code teams. The KNL hardware provides many new features like dual 512-bit vector units, up to 288 hardware threads and the addition of on-package high-bandwidth

memory. The roofline model enables a code team to determine which of these new hardware features they should target. For example, in a memory bandwidth bound code optimizations targeting better vectorization would be fruitless until other optimizations targeting data-reuse are considered.

In this paper, we will present an overview of the Roofline Model, describe the methodology and tools that were used to characterize code performance, and briefly describe some of the optimizations that were made to improve performance.

2 The Roofline Model and Arithmetic Intensity

Bottlenecks associated with in-core computation (as opposed to network communication or I/O) are often characterized by instruction- or data-level parallelism within a loop nest as derived from instruction latencies, throughputs, and vector widths of the target processor [3]. Unfortunately, today, it is far more common that performance bottlenecks are associated with the movement of data through the deep cache/memory hierarchy. In an ideal architecture, cache and memory latencies are effectively hidden through a variety of techniques (out-of-order execution, prefetching, multithreading, DMA, etc.) leaving bandwidth as the ultimate constraint. Thus, loop nest (kernel) execution time can be bound by the volume of data movement and the bandwidth to the level of memory capable of containing that data. This bound can be refined by the instruction- and data-level parallelism inherent in the kernel and demanded by the architecture. Although this bound is specific to a particular problem size, one can transform the relationship in order to bound the performance a processor can attain for a given computation. The resultant Roofline Bound [22,21] is shown in Equation 1 where the *Arithmetic Intensity* (AI) represents the total number of floating-point operations performed by the kernel divided by the total resultant data movement after being filtered by the cache.

$$\text{GFLOP/s} = \min \begin{cases} \text{Peak GFLOP/s} \\ \text{Peak GB/s} \times \text{Arithmetic Intensity} \end{cases} \tag{1}$$

Consider the canonical STREAM TRIAD kernel `x[i]=a[i]+alpha*b[i];`: We observe each iteration of this kernel reads two doubles, performs one FMA, write allocates one double, and writes back one double. This provides an arithmetic intensity of 0.0625 FLOPs per byte. On a system with 10 GB/s of memory bandwidth and 100 GFlop/s of peak performance, the Roofline model will bound performance at 0.625 GFlop/s or less than 1% of peak.

Although the STREAM TRIAD kernel has no data locality, stencils like a canonical 7-point constant coefficient stencil do. Although such a kernel presents 7 reads and one write to the cache subsystem, in an ideal execution, all but one read and one write allocate/writeback should be filtered by the cache. As such, the ultimate arithmetic intensity for such a kernel is 0.291 FLOPs per byte. Failure to attain this arithmetic intensity (as measured by memory controller performance counters) is indicative of a discrepancy between the cache

requirements as presented by the code and the cache capacity provided by the processor, and strongly motivates effective cache blocking (loop tiling).

Although the 7-point stencil performs 7 floating-point operations, they are actually a mix of 6 adds and 1 multiply. For architectures that execute multiplies and adds in different pipelines, peak performance may only be attained if the dynamic instruction mix is balanced. In this example, the effective peak is only 58% of the peak on a machine that implements FMA or separate multiple and add pipelines. Although, the bandwidth-intensive nature of the 7-point stencil precludes it from being compute-limited, other kernels may be sensitive to this imbalance.

We may similarly refine the "Peak GFlop/s" of Equation 1 into a function of the instruction-, data-level parallelism within the kernel. Whereas the former is often attributed to a lack of loop unrolling, the latter is often associated with an inability to vectorize the kernel in order to target 128-, 256-, or 512-bit vector instructions. Regardless, the penalty on the performance bounds can be severe — up to 80× on an Intel Knights Landing processor.

Figure 1a presents a generic Roofline model in which performance is plotted as a function of arithmetic intensity. Additional "ceilings" denote restricted performance bounds derived from the lack of parallelism. For each kernel, a series of "walls" can be constructed based on the difference in total data movement (compulsory, capacity, conflict) and the theoretical data movement lower bound (compulsory cache misses) [8]. For working sets that fit in main memory, performance is initially bound by memory bandwidth. Cache blocking will increase arithmetic intensity, but will require some degree of vectorization to improve performance.

3 Target Hardware Architecture

Cori is a Cray XC40 [5] based supercomputer and is being deployed in two phases. Phase 1 uses Intel Haswell multi-core processors and was deployed late-2015.

- Cray XC40 architecture with the Aries Dragonfly topology high speed network
- 1,630 compute nodes, where each node contains 2, 16-core, 2.3 GHz Haswell processors and 128 GB DDR4 2133MHz memory
- 1.92 PFLOP/s (theoretical peak)
- 203 TB aggregate memory
- 30 PB scratch storage with a peak bandwidth of > 700 GB/sec

Phase 2, scheduled for deployment mid-2016, will be an expansion of Cori and add over 9,300 Intel Knights Landing based nodes. Since Cori's KNL based partition is still to be deployed, the KNL results were collected using standalone Intel white boxes with pre-production KNL processors.

- KNL preproduction, B0 stepping
- 64 cores @ 1.3 GHz with 4 hyper-threads per core (256 total threads)
- 16 GB MCDRAM, >460 GB/sec peak bandwidth
- 96 GB (6 x 16 GB) DDR4 @ 2133 GHz, 102 GB/sec peak

For this study, all results are collected with a single Cori Haswell based node and then compared to a single KNL white box. Multi-node analysis will be the subject of future studies. We used MPI and at least one rank per socket to avoid NUMA effects in Cori's dual-socket Haswell node. In addition, for most applications we used the Linux *numactl* utility to control memory affinity on the KNL, targeting MCDRAM only (*numactl -m 1*) or DDR4 only (*numactl -m 0, or without numactl*) in our tests. All applications use double-precision floating-point unless stated otherwise.

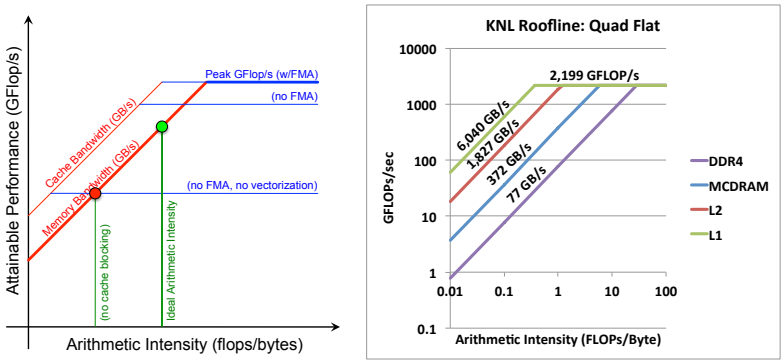
4 Tools and Methods

Using the Empirical Roofline Toolkit (ERT) [23,10], we measured the maximum sustained bandwidth at each level of the cache hierarchy and the maximum sustained floating-point rate for the KNL processor. We configured the toolkit with the following parameters:

- ERT_CC mpiicc
- ERT_CFLAGS -O3 -xMIC-AVX512 -fno-alias -fno-fnalias -DERT_INTEL
- ERT_FLOPS 1,2,4,8,16,32,64,128
- ERT_ALIGN 64
- ERT_MPI_PROCS 1,2,4,8,16
- ERT_PROCS_THREADS 256
- ERT_OPENMP_THREADS 1-256
- ERT_NUM_EXPERIMENTS 3
- ERT_MEMORY_MAX 8589934592

ERT performs a sweep of all the specified MPI rank combinations specified by ERT_MPI_PROCS. For each MPI sweep it executes a computational kernel with ERT_FLOPS operations per loop iteration. ERT keeps the total concurrency (ERT_PROCS_THREADS) fixed for each sweep, so as the number of MPI ranks increases, the number of threads per rank decreases an equal amount. We used the toolkit's nominal *driver1* and *kernel1*. The toolkit then searches all results and uses the maximum values found for the L1, L2 and DRAM interfaces. The results are shown in Figure 1b. The Linux utility *numactl* was used to target MCDRAM (flag *-m 1*) or DDR4 (flag *-m 0*) respectively.

The KNL is capable of being configured in multiple different MCDRAM and sub-NUMA modes. An explanation of all the possible configurations is beyond the scope of this paper, but can be found in Sodani's Hot Chips presentation [18]. The ERT was applied using Quad-Cache, Quad-Flat, Sub-NUMA Cluster 2 (SNC2) and Sub-NUMA Cluster 4 (SNC4) modes to determine if there was a significant difference in performance. All four modes provided equivalent floating-point performance, which is expected with the ERT as the peak floating-point rates are achieved with a working set that fits in L1 cache. The MCDRAM performance does vary, with Quad-Cache mode giving the lowest performance, Quad-Flat and SNC2 providing near equal bandwidths, and SNC4 giving the best performance, 20% higher than Quad-Cache. The results of the ERT are used to form the roofline for the applications and kernels in Section 5.



(a) Generic Roofline Performance (b) KNL ERT Performance

Figure 1: 1a) Generic Roofline representation showing the ultimate bounds on performance, bandwidth, and arithmetic intensity, with ceilings to denote limitations from a lack of various forms of parallelism. 1b) Applying the Roofline Toolkit, we estimate KNL maximum sustained memory bandwidth for the L1 and L2 caches, the MCDRAM and DDR4 bandwidth, and the maximum sustained performance in GFLOP/s.

Table 1: ERT Performance for Different KNL Memory Modes

	Quad Cache	Quad Flat	Sub-NUMA Cluster 2	Sub-NUMA Cluster 4
GFLOP/s	2,205	2,199	2,224	2,212
MCDRAM GB/s	345	372	381	415
DDR4 GB/s	-	77	77	77

Unless otherwise noted, *all application results presented in the following sections are for Quad-Flat mode* as at time of this study it was the most mature from a software and firmware perspective. In addition, we did collect application data for SNC2, SNC4 and in some cases Quad-Cache but did not see performance differences greater than 20% from that obtained with Quad-Flat and we feel Quad-Flat is representative of all modes except Quad-Cache. We will do a more extensive and detailed comparison in future studies.

We used Intel's Software Development Emulator (SDE) [19,17] to count the number of floating-point operations. SDE is capable of dynamic instruction tracing, and we use this capability to obtain total instructions executed, the instruction type (e.g. read, read width, single-precision, double-precision, fused multiply-add, SIMD width, etc.), and the instruction set architecture grouping (e.g. SSE, AVX, etc.).

For this study, we use Intel's VTune Amplifier XE performance analysis tool to measure data movement at both the DDR and MCDRAM memory interfaces. A tutorial for using SDE and VTune to calculate AI can be found on the NERSC web site [14,7].

5 Applications and Kernels

5.1 WARP-PICSAR

WARP is an open-source particle-in-cell (PIC) code designed to simulate charged particle beams and laser-matter interaction [9]. To aid in preparing for Cori, the library PICSAR has been developed. This library contains a Fortran kernel based on WARP with optimized subroutines. These high-performance subroutines are interfaced with a python class that can be imported and used in WARP scripts. It also contains a stand-alone Fortran code that is used as a test bed for optimization and profiling. The typical PIC kernel is composed of a time loop with four intermediate steps: the Maxwell solver, the field gathering, the particle pusher and the current deposition [4]. For many cases, interpolation processes such as the current deposition and the field gathering represents the most costly steps and are weakly vectorized in their common form.

A first optimization is the implementation of a hybrid threaded parallelization. PIC codes usually use a domain decomposition with one MPI process per subdomain. OpenMP provides a second level of parallelization inside subdomains. The subdomains are then divided into tiles, i.e. small portion of the subdomain having their own particle property arrays. Tiling improves memory locality and significantly diminishes RAM memory access (cache reuse). With more tiles than OpenMP threads, tiles computation is automatically load balanced with the OpenMP scheduler. On Haswell, field grid arrays can be fully contained in L2 when tile dimension is sufficiently small (below 8x8x8). On KNL, tile field arrays can be in L2 (512 KB) whereas all the problem is contained in the HBM.

Direct current deposition and field gathering interpolation steps were rewritten to enable more efficient vectorization than the classical form [20]. Vectorization is done by adding !$OMP SIMD directives. In addition, a particle cell sorting process has been added. Performed on every given time step in each tile, it further improves cache reuse and memory locality while accessing particle properties, especially during the current deposition and field gathering.

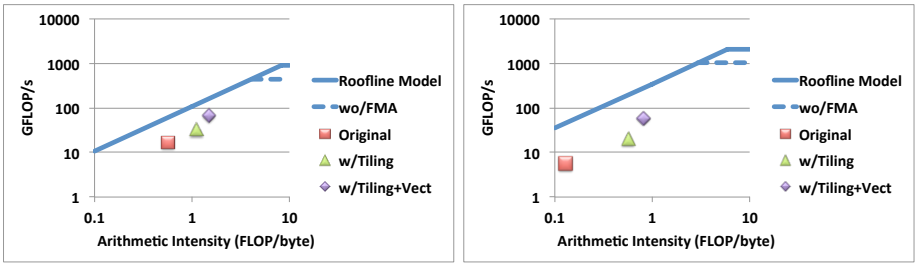
As a test case, we consider a Maxwellian homogeneous plasma with initial thermal velocity of 0.1c. The domain discretization is of 100x100x100 cells with 20 super-particles per cell. The simulations are performed on a node of Haswell with 2 MPI tasks and 16 OpenMP threads, and on Intel Xeon Phi KNL with 4 MPI tasks and 32 OpenMP threads. Performance can be slightly better when hyper-threading is used on KNL: we use 2 threads per core. Tile dimension is of 8x8x8 cells. The tile size is 250 KB for internal temporary current grids (used for vectorization), 31 KB for local current grid and 640 KB for particle properties. On Haswell, the global field arrays (27 Mb) fit in L3 and the local tile field arrays fit in L2. On KNL, the problem is fully contained in the HBM. Local tile field arrays fit in L2. Memory management is still under study.

Arithmetic intensities for each of the optimization steps is shown in Table 2. Figure 2 illustrates applying the results to the Roofline Model, demonstrating how tiling and vectorization improve AI and increase overall performance, reaching a higher memory bandwidth ceiling.

After tiling and vectorization optimizations, memory locality is improved, resulting in an overall performance improvement for both architectures. Performance relative to the original code improves by a factor of 4.0 for Haswell and 10.8 for KNL MCDRAM. Although final KNL performance is 0.89 times that of Haswell, it is important to note that both architectures benefited from the optimizations with KNL demonstrating the largest individual gain. For KNL, the speedup seen by using MCDRAM vs. DDR is 1.2, demonstrating that to some degree PICSAR is memory bandwidth bound.

Table 2: PICSAR Arithmetic Intensity and Performance

Optimization	Haswell		KNL MCDRAM		KNL DDR		KNL/HSW
	AI	GFLOP/s	AI	GFLOP/s	AI	GFLOP/s	Speedup
Original	0.57	16.7	0.13	5.6	0.13	5.4	0.34
Tiling	1.10	32.0	0.56	20.0	0.56	19.2	0.63
Tiling+Vectorization	1.50	67.5	0.81	60.4	0.81	49.4	0.89



(a) Haswell Roofline (b) KNL Roofline

Figure 2: The tiling optimization increases AI and moves the data point to the right. Applying the vectorization optimization allows PICSAR to take advantage of the additional effective memory bandwidth and increases the overall performance for both architectures.

5.2 EMGeo

In geophysical-imaging, medium properties can be studied by performing scattering experiments using electromagnetic or seismic waves. Quantities such as densities, elasticities, stress etc. of the medium can be obtained from fitting the observed measurements to the results predicted by a simulation. The code EMGeo performs these simulations and solves the inverse scattering problem in the Laplace-Fourier domain [16]. We focus only on the Seismic part and forward step of the inverse scattering problem, which involves inverting a large sparse matrix. For this purpose, EMGeo uses an Induced Dimensional Reduction (IDR) Krylov subspace solver.

The Sparse Matrix Vector (SpMV) product is responsible for two thirds of the total runtime. EMGeo performs SpMV operations using two low-bandwidth

Table 3: EMGeo Arithmetic Intensity and Performance. "Best" referts to SB in Haswell and loop reordering in KNL

	Haswell (1 Sckt)		KNL MCDRAM		KNL DDR		KNL/HSW
Optimization	AI	GFLOP/s	AI	GFLOP/s	AI	GFLOP/s	Speedup
Original	0.31	19.2	0.27	71.1	0.27	23.5	3.7
SELL	0.27	16.9	0.24	71.0	0.24	21.2	4.2
SB	0.34	20.2	0.28	62.3	0.28	20.9	3.1
SELL+SB	0.31	19.2	0.26	63.9	0.26	19.6	3.3
nRHS+SELL+Best	1.29	77.7	0.76	278.5	0.76	65.8	3.6

matrices (with maximum of 12 nonzero per row). We use the larger matrix in our benchmark. The production code evaluates about 256 independent right hand sides (RHS) in column major format. All the arrays are stored in double-complex data format.

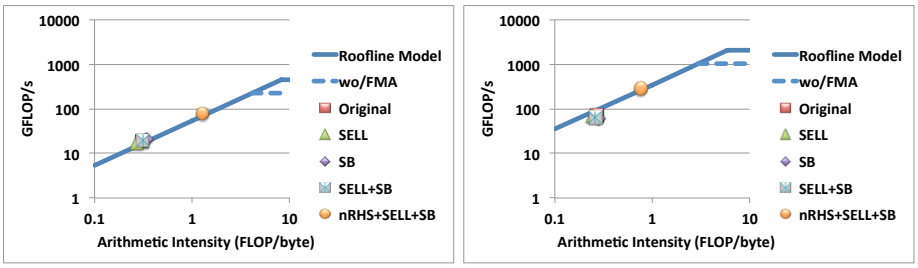
We use Sliced ELLPack (SELL) sparse matrix format, Spatial Blocking (SB), and multiple right hand sides (nRHS) cache blocking optimizations to increase AI and thus the performance in the SpMV operations.

Table 3 summarizes our optimization improvements in the SpMV benchmark. For EMGeo, we only used a single socket on the Cori Haswell node to avoid NUMA issues and aid in analyzing the code characteristics. With full optimization, the GFLOP/s rate increases by a factor of 4.1 on Haswell and 3.9 on KNL. The KNL rate is 3.6× better than in a single Haswell socket, mainly due to the high memory bandwidth in KNL, where the benchmark is memory bandwidth bound. The SELLPack format reduces the FLOP count and data movement, so the AI and GFLOP/s values do not reflect the actual improvement in execution time where we see a 5.0× speedup in Haswell, a 4.8× speedup on KNL and a 3.6× speedup of KNL relative to Haswell.

Although the SB optimization improves the performance in Haswell, it degrades the performance in KNL, even after tuning the cache blocks size. We believe that the SB technique is effective when a large shared cache memory is available, which is the case for Haswell, but not KNL. We replace the SB in KNL's code in the last row of Table 3 with loop reordering, which is equivalent to SB of size one. Our roofline analysis shows that our optimizations improved the arithmetic intensity from 0.3 to 1.3 in Haswell and from 0.3 to 0.8 in KNL, as shown in Figure 3. The SpMV optimization translates into an overall speedup of 1.7× in Haswell and 1.9× in KNL for the forward step of the full application, using a grid of size $100 \times 50 \times 50$. The Details of this study are available in [11].

5.3 MFDn

The Many-body Fermion Dynamics for nuclei (MFDn) code is a nuclear physics code in which the lowest few eigenvalues and eigenvectors of a very large real sparse symmetric matrix are found though iterative means [12]. Sparse matrix vector and and sparse matrix transpose vector products are key kernels in the iterative eigensolver. The sparse matrix is stored in a compressed sparse block coordinate (CSB_COO) [2,1] format which allows efficient linear algebra operations on



(a) Haswell Roofline (1 Socket) (b) KNL Roofline

Figure 3: The EMGeo roofline analysis shows that on Haswell and KNL the code is memory bound, and despite the optimizations performed on the code only modest improvements in performance are made. However, adding multiple right hand sides improves memory locality and hence improves AI, with a corresponding improvement in performance.

the large sparse matrix. The sparse matrix elements and corresponding indices account for 64 GB of the memory and the input/output vectors account for up to 16 GB depending on the specific problem.

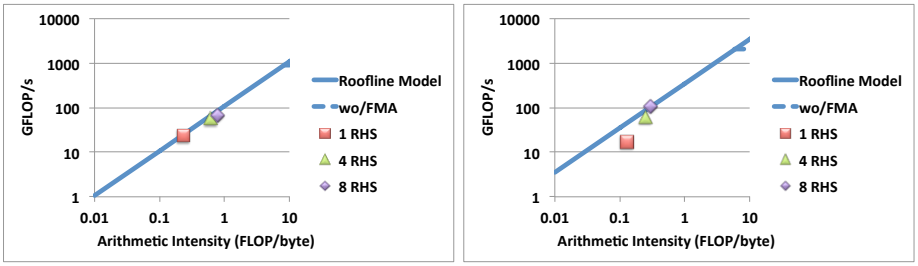
Improving data reuse, allowing vectorization and effectively using as much aggregate bandwidth as possible are key challenges. We therefore replaced sparse matrix vector (SpMV) with sparse matrix-matrix (SpMM) operations on blocks of vectors. To better utilize memory bandwidth we explicitly place the input/output vectors in MCDRAM and the rest of the code and data reside in DDR4. Generally the larger the block of vector operations that can be done simultaneously (the number of right hand sides (nRHS)) the better the performance, however the number of vectors is limited by the available MCDRAM.

Our test problem consists of 2 protons and 6 neutrons. The sparsity structure is determined by the many body basis states and quantum selection rules resulting in a quasi-random distribution of nonzero matrix elements. That it is, the matrix is not banded or well structured. The test problem for KNL is designed to run on 4,560 nodes with a total nxn matrix with $n = 3e11$. Our single node test case simulates the work of one node responsible for an mxm block of the matrix with $m = 1e10$ with a local sparsity of $5e{-}7$. This corresponds to approximately 7.5e9 nonzero matrix elements. For consistency all of our tests were done with a CSB block size, $\beta = 16000$.

The performance results are summarized in Table 4 and Figure 4. Since both MCDRAM and DDR4 are used in this implementation, arithmetic intensity is calculated using the sum of the data movement for both of the memories. All floating-point calculations are single precision and the Roofline model in Figure 4 is adjusted accordingly, although current performance is no where close to GFLOP/s ceiling. Using 8 RHS improves performance by a factor of 2.9 for Haswell and 6.4 for KNL. KNL performance is 1.6 times that of Haswell, and 3.6 times better than using DDR only, the latter demonstrating MFDn is memory bandwidth bound.

Table 4: MFDn Arithmetic Intensity and Performance

	Haswell		KNL MCDRAM		KNL DDR		KNL/HSW
nRHS	AI	GFLOP/s	AI	GFLOP/s	AI	GFLOP/s	Speedup
1	0.23	23.2	0.13	17.1	0.13	13.5	0.74
4	0.62	56.8	0.25	62.4	0.25	27.8	1.1
8	0.80	67.5	0.30	109.1	0.30	30.7	1.6



(a) Haswell Roofline (b) KNL Roofline

Figure 4: MFDn clearly meets the bandwidth bound portion of the roofline model (single precision FP only). By increasing the number of simultaneous vectors (RHS), performance improves as arithmetic intensity increases. However, the number of RHS is limited by available MCDRAM capacity.

5.4 BerkeleyGW

BerkeleyGW is a materials science application for computing excited state properties of materials - those associated with electrons populating orbitals beyond the quantum ground state[6]. BerkeleyGW takes as input the ground-state data computed from a number of DFT codes like Quantum ESPRESSO, SIESTA, PARATEC. The code is dominated by dense linear algebra (Matrix Multiply (GEMM), Diagonalization and Inversion), FFTs and hand tuned code representing tensor contraction like operations expressed as large array reductions. We predominantly focus on the hand-tuned routines in our KNL preparation, which in recent years has become a more significant amount of the runtime of a GW calculations due to changing use cases. The performance of the FFTs and linear algebra steps will be discussed in a separate article on BGW performance.

Following our optimization process, we show the following data points for the baseline MPI-only code. in Table 5 and Figure 5.

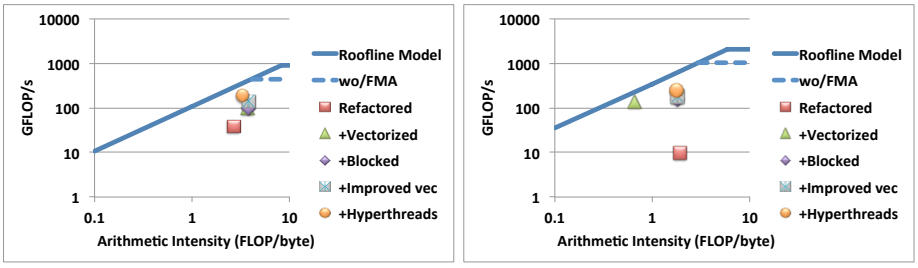
- We refactor primarily to support OpenMP threads and improved data-locality. The code at this point has a three loop structure, with an outer loop targeted at MPI, and nested inner-loops with large trip counts targeting threads and AVX parallelization.
- We factor the code to support compiler auto-vectorization by moving a innermost trip-count 3 loop outwards, remove cycle statements and conditionals.
- We add a layer of cache-blocking to effectively reuse the L2. We reordered loops to improve vectorization (moving a loop of trip count 3 outwards), improving AI. We introduce cache blocking around the trip-count 3 loop. On Haswell, we are able to effectively use L3 and so no again in AI is seen.

– We replace the complex divide with a manual divide over the real absolute value of the complex number in order to avoid x87 instruction generation.
– We put back in the explicitly complex divide but utilize the compiler flag *-fp-model fast=2* which avoids x87 instructions by assuming there is no overflow concern. Additionally, we run with 2 threads per core, which is where most of the speedup occurs.

In summary, a few key lessons stand out from BerkeleyGW. For a code with an AI between 1-10 (i.e. near the roofline cusp), good performance requires, good data reuse out of L2 cache to reach the highest AI (KNL's lack of L3 can make it more punishing), placement of data in HBM, and good use of the vector processing units are all essential for good performance. The current limiting factor is the latency in the divides, lack of multiply add balance and remaining conditionals. After all the optimizations, performance relative to the original code improves by a factor of 11.8 for Haswell and 25.8 for KNL MCDRAM. The KNL demostrated a 1.35 times improvement over Haswell. Comparing KNL MCDRAM to KNL DDR, using MCDRAM allows for a performance improvement factor of 1.75.

Table 5: BerkeleyGW Arithmetic Intensity and Performance

Optimization	Haswell		KNL MCDRAM		KNL DDR		KNL/HSW
	AI	GFLOP/s	AI	GFLOP/s	AI	GFLOP/s	Speedup
Refactored	2.64	38.7	1.93	9.80	1.93	9.80	0.25
+Vectorized	3.68	100.3	0.66	143.4	0.66	55.1	1.43
+Blocked	3.77	100.3	1.79	153.2	1.79	140.8	1.53
+Improved Vect	3.78	142.6	1.80	178.4	1.80	142.1	1.25
+Hyperthreads	3.27	186.9	1.76	252.6	1.76	144.0	1.35



(a) Haswell Roofline (b) KNL Roofline

Figure 5: BerkeleyGW is a good example of an application that benefited from blocking, threading and vectorization improvements. For Haswell, *+Blocked* provides no further improvement over *+Vectorized*, due to the fact that the working set fits within the Haswell L3 cache. For KNL, *+Vectorized* performance is limited by MCDRAM BW (due to its low AI) and *+Blocked* is necessary to see improvements in further optimizations.

5.5 Performance Summary and Observations

Table 6 shows the fully optimized performance for each application or kernel. They all demonstrated significant performance gains over the baseline code for both Haswell and KNL architectures. The KNL architecture showed overall better performance than Haswell with the exception of PICSAR, however both architectures benefited significantly from the optimization process.

Table 6: Performance Summary

	GFLOP/s			Speedup	
	Haswell	KNL MCDRAM	KNL DDR	KNL/HSW	MCDRAM/DDR
PICSAR	67.5	60.4	49.4	0.89	1.2
EMGeo (SpMV)[1]	77.7	181.0	43.6	2.33	4.2
MFDn	67.5	109.1	30.7	1.62	3.6
BerkeleyGW	186.9	252.6	144.0	1.35	1.75

Applying results to the Roofline Model, no application or kernel had an AI that put it in the regime of being computational bound, all were in a region in which memory bandwidth was the limiting factor to performance. EMGeo and MFDn were clearly bandwidth bound, while PICSAR and BerkeleyGW showed there is headroom for further optimization.

We observe that Haswell consistently attains a higher AI than KNL. As all applications perform the same number of floating point operations, we conclude that KNL generally moves more data to/from main memory than Haswell. As a result, the theoretical performance benefits of higher MCDRAM bandwidth are not fully realized. Exploration of the performance tradeoffs between larger on-chip L2/L3 caches (reduced data movement) and reduced computational performance (fewer cores for constant chip area) are in order.

6 Conclusion and Outlook

In this study we have developed a Roofline Model for the Intel Knights Landing processor and have estimated upper bounds for L1, L2, MCDRAM and DDR4. We then measured the performance of a suite of NESAP applications (or proxy kernels) and used the Roofline Model to determine to what degree they were compute- or memory-bound. Each application developer then explored a variety of optimizations to improve both arithmetic intensity and overall performance. We then re-evaluated the impact of those optimizations relative to the Roofline ceilings. All of the evaluated applications were able to substantially improve their overall performance on both Haswell and KNL processors, often by increasing the computational arithmetic intensity and improving memory bandwidth utilization. Having a visual representation of the performance ceiling helps guide application experts to appropriately focus their optimization efforts.

[1] Haswell performance is for a single socket.

Acknowledgments This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231.

J.D. was supported by the SciDAC Program on Excited State Phenomena in Energy Materials funded by the U. S. Department of Energy, Office of Basic Energy Sciences and of Advanced Scientific Computing Research, under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory

References

1. Aktulga, H.M., Buluc, A., Williams, S., Yang, C.: Optimizing sparse matrix-multiple vector multiplication for nuclear configuration interaction calculations. In: International Parallel and Distributed Processing Symposium (IPDPS 2014) (05/2014 2014)
2. Aktulga, H.M., Yang, C., Ng, E.G., Maris, P., Vary, J.P.: Improving the scalability of a symmetric iterative eigensolver for multi-core platforms. Concurrency and Computation: Practice and Experience 26(16), 2631–2651 (2014), http://dx.doi.org/10.1002/cpe.3129
3. Carr, S., Kennedy, K.: Improving the ratio of memory operations to floating-point operations in loops. ACM Trans. Program. Lang. Syst. 16(6), 1768–1810 (Nov 1994), http://doi.acm.org/10.1145/197320.197366
4. C.K. Birdsall, A.B.L.: Plasma Physics Via Computer Simulation. Series in Plasma Physics, CRC Press (2005)
5. Cray xc series supercomputers, http://www.cray.com/products/computing/xc-series
6. Deslippe, J., Samsonidze, G., Strubbe, D.A., Jain, M., Cohen, M.L., Louie, S.G.: Berkeleygw: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures. Computer Physics Communications 183(6), 1269 – 1289 (2012), http://www.sciencedirect.com/science/article/pii/S0010465511003912
7. Doerfler, D.: Understanding application data movement characteristics using intel vtune amplifier and software development emulator tools. In: IXPUG 2015. Berkeley, CA (Sept 28 - Oct 2 2015)
8. Hill, M.D., Smith, A.J.: Evaluating Associativity in CPU Caches. IEEE Trans. Comput. 38(12), 1612–1630 (1989)
9. Laboratory, L.B.N.: Warp website, http://warp.lbl.gov
10. Ligocki, T.: Roofline toolkit, https://bitbucket.org/berkeleylab/cs-roofline-toolkit
11. Malas, T., Kurth, T., Deslippe, J.: Optimization of the sparse matrix-vector products of an idr krylov iterative solver for the intel knl manycore processor. In preparation
12. Maris, P., Aktulga, H.M., Caprio, M.A., ÃIJmit V ĂĞatalyÃijrek, Ng, E.G., Oryspayev, D., Potter, H., Saule, E., Sosonkina, M., Vary, J.P., Yang, C., Zhou, Z.: Large-scale ab initio configuration interaction calculations for light nuclei. Journal of Physics: Conference Series 403(1), 012019 (2012), http://stacks.iop.org/1742-6596/403/i=1/a=012019
13. NERSC: Cori, https://www.nersc.gov/systems/cori/

14. NERSC: Measuring arithmetic intensity, https://www.nersc.gov/users/application-performance/measuring-arithmetic-intensity
15. Nesap, http://www.nersc.gov/users/computational-systems/cori/nesap/
16. Petrov, P.V., Newman, G.A.: 3d finite-difference modeling of elastic wave propagation in the laplace-fourier domain. GEOPHYSICS 77(4), T137–T155 (2012), http://dx.doi.org/10.1190/geo2011-0238.1
17. Raman, K.: Calculating "flop" using intel software development emulator (intel sde) (March 2015), https://software.intel.com/en-us/articles/calculating-flop-using-intel-software-development-emulator-intel-sde
18. Sodani, A.: Knights landing (knl): 2nd generation intel xeon phi processor. In: Hot Chips 27. Flint Center, Cupertino, CA (August 23rd-25th 2015), http://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.25-Tuesday-Epub/HC27.25.70-Processors-Epub/HC27.25.710-Knights-Landing-Sodani-Intel.pdf
19. Tal, A.: Intel software development emulator, https://software.intel.com/en-us/articles/intel-software-development-emulator
20. Vincenti, H., Lehe, R., Sasanka, R., Vay, J.: An efficient and portable SIMD algorithm for charge/current deposition in Particle-In-Cell codes. ArXiv e-prints (Jan 2016)
21. Williams, S.: Auto-tuning Performance on Multicore Computers. Ph.D. thesis, EECS Department, University of California, Berkeley (December 2008)
22. Williams, S., Watterman, A., Patterson, D.: Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Communications of the ACM (April 2009)
23. Williams, S., Stralen, B.V., Ligocki, T., Oliker, L., Cordery, M., Lo, L.: Roofline performance model, http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/