An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability

Charlene Yang*, Rahulkumar Gayatri*, Thorsten Kurth*, Protonu Basu†, Zahra Ronaghi*
Adedoyin Adetokunbo‡, Brian Friesen*, Brandon Cook*, Douglas Doerfler*
Leonid Oliker†, Jack Deslippe*, Samuel Williams†

*National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
{cjyang, rgayatri, tkurth, zronaghi, bfriesen, bgcook, dwdoerf, jrdeslippe}@lbl.gov
†Computational Research Division
Lawrence Berkeley National Laboratory
{pbasu, loliker, swwilliams}@lbl.gov
‡Los Alamos National Laboratory
aadedoyin@lanl.gov

***Abstract*—System and node architectures continue to diversify to better balance on-node computation, memory capacity, memory bandwidth, interconnect bandwidth, power, and cost for specific computational workloads. For many application developers, achieving performance portability (effectively exploiting the capabilities of multiple architectures) is a desired goal. Unfortunately, dramatically different per-node performance coupled with differences in machine balance can lead to developers being unable to determine whether they have attained performance portability or simply written portable code. The Roofline model provides a means of quantitatively assessing how well a given application makes use of a target platform's computational capabilities. In this paper, we extend the Roofline model so that it 1) empirically captures a more realistic set of performance bounds for CPUs and GPUs, 2) factors in the true cost of different floating-point instructions when counting FLOPs, 3) incorporates the effects of different memory access patterns, and 4) with appropriate pairing of code performance and Roofline ceiling, facilitates the performance portability analysis.**

***Index Terms*—performance portability, performance model, Roofline, KNL, GPU, performance counters**

I. Introduction

Application portability is becoming more and more desirable by many software developers and users as computer architectures diversify [1]–[4]. Depending on the purpose of a given application and target audience, "portability" can mean different things to developers and/or users. For example, it can simply mean that an application can be correctly executed across a variety of hardware architectures or operating systems. Users may also expect a certain level of performance consistently reached on different architectures, especially in the world of high performance computing (HPC).

Unfortunately, there is little consensus on how to exactly define or quantify "performance portability". However, we believe that a reasonable definition should be based on a quantitative measure of how effectively an application uses different aspects on each target architecture. Two definitions of this efficiency are common. First, one could examine the ratio of the code's performance on that architecture to that of its best implementation on all architectures in question – as defined in [5], "application efficiency" . Alternatively, one can examine the ratio of the code's actual performance to that architecture's peak performance – "architectural efficiency" [5]. In the latter approach, it is imperative that any formula for quantifying performance portability should incorporate performance-relevant hardware features and limitations such as peak FLOP rate, memory bandwidth for the different memory levels, instruction issue rates, *etc*.

Although there is little consensus on the definition of performance portability, all definitions and metrics have helped provide a meaningful comparison between performance on different architectures and thus make useful suggestions to future code optimization.

The Roofline performance model is very effective in characterizing codes on a variety of architectures, identifying code bottlenecks, and guiding optimization efforts [6], [7], and it has been widely adopted in HPC performance analysis and workflows [8], [9]. In [5], [10], [11], it has been incorporated in the metric for assessing performance portability across different platforms as well. Compared to the two example definitions of application's efficiency mentioned above, the Roofline model can account for different architecture-specific performance bounds that limit an architecture's sustained performance. Moreover, as Roofline adapts to changes in problem size and different implementations of the code, it provides a more accurate assessment of the application's architectural efficiency than simply percentage of the peak FLOP/s.

There are a few nuances to defining performance portability relative to a performance model like Roofline. First, accurate ceilings must be established through the use of benchmarks rather than a vendor's own specification numbers which can be both theoretical and over-optimistic. Second, a relevant ceiling

needs to be selected in order for meaningful comparisons to be made. This requires some knowledge of the code's instruction mix and dominant memory level. Finally, one can no longer rely solely on canonical FLOPs (those counted by hand) as the presence of not only FMAs, but also divides, exponentials, and logarithms can skew these counts. For example, a floating-point divide should be counted as multiple FLOPs as they are usually implemented by multiple instructions on most modern CPU and GPU architectures.

This paper makes several contributions. First, it quantifies the realistic Roofline ceilings for the Intel Knights Landing CPU (KNL) [12] and the NVIDIA V100 Volta GPU [13]. Second, we develop and deploy a methodology for both the KNL and V100 that accurately accounts for the true cost of different instruction mixes - demonstrated with floating-point divides - in the context of the Roofline model. Third, we settle the long-term question of how Roofline accounts for different memory access patterns - demonstrated through moderate stride memory access. Failure to embrace these concepts both over-inflates an architecture's computational ceilings, underestimates an application's arithmetic intensity and computational performance, and results in a large (and erroneous) discrepancy between observation and performance bound. Finally, we show that only with proper application instrumentation and system benchmarking can Roofline be used as the basis for quantitatively assessing performance portability across CPU and GPU architectures.

II. Methodology

In this section, we lay out the methodology to empirically collect Roofline ceilings and Roofline performance data for a given architecture and for a given application, in the context of performance portability. We will introduce the performance portability metric we will use in this paper, the architectures we will study, and the kernel we will use to validate our analysis. Tools will also be introduced to detail the process of collecting empirical architecture characterization data and empirical application performance data.

A. Performance Portability Metric

We adopt the performance portability metric from [5], [10], and specifically, the performance portability $\mathbf{\Phi}$ of an application a solving problem p on a given set of platforms $\boldsymbol{H}$ is,

$$\mathbf{\Phi}(a,p,\boldsymbol{H}) = \begin{cases} \dfrac{|\boldsymbol{H}|}{\sum_{i\in\boldsymbol{H}} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported,} \\ & \forall i \in \boldsymbol{H} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where $e_i(a,p)$ is the application a's architectural efficiency on architecture i, and is obtained using the Roofline performance model [6]:

$$e_i(a,p) = \frac{P_i(a,p)}{\min(F_i, B_i \times I_i(a,p))} \quad (2)$$

where $P_i(a,p)$ is the observed code performance in floating-point operations per second (FLOP/s), F_i is the peak FLOP/s performance of architecture i, B_i is the peak bandwidth of architecture i, and $I_i(a,p)$ is the arithmetic intensity (AI) of application a on architecture i. The denominator $\min(F_i, B_i \times I_i(a,p))$ accounts for when the application is compute bound (i.e. by F_i) as well as when it's bandwidth bound (i.e. by $B_i \times I_i(a,p)$), adding extra accuracy to the calculation of application's architectural efficiency $e_i(a,p)$.

Much of this paper is focused on improving the accuracy of the components of $e_i(a,p)$ — namely $P_i(a,p)$, F_i, B_i, and $I_i(a,p)$. To that end, we deploy empirical benchmarking for F_i and B_i, and also account for the true cost of different instruction mixes and memory access patterns to more accurately measure $P_i(a,p)$ and $I_i(a,p)$.

B. Two Architectures

We investigate two contemporary HPC architectures in this paper — the Intel Knights Landing (KNL) CPU [12] and the NVIDIA V100 Volta GPU [13]. To this end, we utilize the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC) and the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF).

Cori is a CPU-based Cray XC30 system comprised of 2388 Haswell nodes and 9688 KNL nodes. Each KNL node is a single-socket Xeon Phi 7250 processor, with 68 cores, two 512-bit Vector Processing Units (VPUs) per core and four hardware threads per core. 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 memory (HBM). At a nominal frequency of 1.4GHz, each node is advertised to deliver 3 TFLOP/s of peak performance and 490 GB/s of HBM bandwidth [14].

Summit is a GPU-accelerated supercomputer with a total of 4608 nodes, each consisting of 6 NVIDIA V100 GPUs and 2 IBM Power9 CPUs. Each V100 has 80 Streaming Multiprocessors (SMs), and each SM has 64 FP32 cores, 64 INT32 cores, 32 FP64 cores, 8 Tensor Cores, and four texture units. Similar to the KNL CPU processor, the V100 has 16GB of HBM2, and it delivers a theoretical peak performance of 7.8 TFLOP/s and 900 GB/s of bandwidth from the HBM [13].

It is apparent that these two architectures have similar features as they are targeted at the same segment of customers (HPC). However, using the Roofline performance model, we can still show that the subtleties of their differences can be exposed, later in Section III.

C. Machine Characterization

Vendor performance numbers, 3 TFLOP/s for KNL and 7.8 TFLOP/s for V100 as above mentioned, are theoretical bounds on machine performance and may not necessarily reflect the realities of code generation, data locality, instruction issue bandwidth, memory controller efficiency, or power-constrained environments. As such, they can become disconnected from the performance achievable by any real-life code running on that architecture. The Empirical Roofline Tool (ERT) [15] can measure the "sustained" computational

performance, cache bandwidths and memory bandwidth, giving a more realistic set of ceilings when using Roofline for performance analysis.

ERT runs a variety of "micro-kernels" sweeping through a range of parameters, such as the number of processes and threads on the CPU, the number of threadblocks and threads on GPU, the problem size, and the number of trials. In this paper, we deploy four "micro-kernels". Namely, they are the "FMA", "no FMA", "divide with FMA", and "divide without FMA" kernels. All of them are double-precision based, and vectorization opportunities and instruction level parallelism (ILP) are also exploited as much as possible. The divide-related kernels here are specifically designed to study the impact of complex operations (such as divides) on the attainable performance of a given architecture (see Section III-B).

We also use ERT to obtain the memory ceilings for the Roofline model. In particular, we are focused on the HBM level of the memory hierarchy, as it is common for large-scale applications to decompose their dataset to fit into HBM on a node level in order to take full advantage of the HBM's high bandwidth. The GPP kernel chosen for this paper (and its problem size) fits into this description as well (Section II-D), i.e. it performs a single node's worth of work and the dataset fits into HBM, but will exceed L2, on both KNL and V100 architectures.

A caveat about using ERT is that even though it provides more realistic bounds on the attainable performance, by no means should one conclude that every application or even every HPC application can attain the same level of performance. The kernels in ERT are often carefully crafted and tuned to match the target architecture's characteristics to fully exploit its potential, whereas in real-world large-scale applications, this is impossible.

D. The General Plasmon Pole (GPP) Kernel

In order to evaluate Roofline in the context of performance portability and ensure a sufficiently robust methodology, we use a highly parameterized version of the General Plasmon Pole (GPP) kernel [16] from a material science code called BerkeleyGW [17]. The GPP kernel calculates electron self-energy using the common General Plasmon Pole approximation [18], and it's written in C++ and parallelized with OpenMP. The computation in this kernel represents work that typically an individual MPI task would perform in a much larger calculation, spanning hundreds or thousands of nodes. The computation is tensor-contraction like, where a few pre-calculated complex double-precision arrays are multiplied and summed over a certain dimension and collapsed into a small matrix. The problem size chosen for this paper is 512 electrons and 32768 plane wave basis elements, and is a medium problem size in the real world of material science.

The pseudo code of GPP implemented on KNL can be described as...

```
#pragma omp parallel
do band = 1, nbands
 do igp = 1, ngpown
  do ig = 1, ncouls #vectorization
   do iw = 1, nw #typically nw=3; unrolled
     load wtilde_array(ig,igp)
     load aqsntemp(ig,band)
     load eps(ig,igp)
     compute wdiff, delw, sch_array
   update achtemp(iw)
```

and on the V100 as...

```
#threadblock grid: (nbands,ngpown)
do band = 1, nbands
 do igp = 1, ngpown
  do ig = 1, ncouls #threads
   do iw = 1, nw #typically nw=3; unrolled
     load wtilde_array(ig,igp)
     load aqsntemp(ig,band)
     load eps(ig,igp)
     compute wdiff, delw, sch_array
   update achtemp(iw)
```

The reason we chose GPP is that not only does GPP offer abundant levels of parallelism (thread and vector), but it also includes several parameters that we may vary in order to arbitrarily increase arithmetic intensity (increase nw in the iw loop or change data type from complex to real - both of which relate to different realistic problem configurations in the full application), enable strided memory access patterns (modification of the ig loop - related to different indexing in the full code), and quantify the impact of floating-point divides in the context of the Roofline model (replace the divide in the `compute delw` statement with a multiply). As such, this single, well-understood kernel can act as a stand-in for a range of potential application kernels.

E. Application Characterization

Although one could count the number of FLOPs in a source code manually, this process is tedious, error-prone, and not scalable to large applications. Compilers convert high-level languages to low-level instructions and different FLOPs can be mapped to different numbers of instructions. Historically, multiplies and adds were generally one instruction per operation (or FLOP), but complex operations such as divides, exponentials, logarithms, and trigonometrics, require more than one instruction per operation, and the ratio of instructions to FLOPs can vary depending on the input data. Thus, counting FLOPs by simply inspecting the source code can be very erroneous.

When it comes to data movement (volume of data moved between two memory/cache levels), it is common to estimate the total amount of read and written data by all the array sizes. But this, again, can be very inaccurate as there are possibly cache misses, cache reuse, or pre-fetching happening in the memory system, skewing the total count of read/written bytes.

Without accurate counting of FLOPs or data movement, many aspects of the Roofline analysis break down, thus we

found it imperative to leverage tools to measure both the data movement and the FLOPs [19]. To that end, we use Intel's Software Development Emulator (SDE) [20], [21], LIKWID [22], [23], and NVIDIA' `nvprof` [24] to collect performance data for GPP. For data movement, we examine only HBM to L2 data movement.

Together with runtime, we calculate the sustained performance (GFLOP/s) for the GPP kernel by,

$$\text{Performance} = \frac{\text{SDE or } \texttt{nvprof} \text{ FLOPs}}{\text{Runtime}}, \tag{3}$$

and the arithmetic intensity (FLOPs/Byte) by,

$$\text{AI} = \frac{\text{SDE or } \texttt{nvprof} \text{ FLOPs}}{\text{LIKWID or } \texttt{nvprof} \text{ data movement}}. \tag{4}$$

More details can be found in [25].

LIKWID's `likwid-perfctr` utility allows users to access hardware performance counters on supported architectures, and it has a few pre-defined performance groups, for users to collect information such as L1 cache miss rate, branching miss rate, and μOPs stalls. In particular, its "HBM_CACHE" performance group captures the number of read and write transactions on a certain memory/cache level, and it is used in this paper to collect the data movement between L2 and HBM. `likwid-perfctr` also has a "FLOPS_DP" performance group, but due to the limitations of KNL hardware counters, LIKWID, as well as other tools that read the same counters such as Intel VTune [26], do not give sufficient accuracy in their estimate of FLOPs. As such, we examined alternate techniques that can capture the FLOP characteristics accurately.

Intel SDE's instruction mix histogram tool can capture dynamic instructions executed, instruction length, instruction category, and ISA (instruction set architecture) extension grouping on Intel architectures. The counting methodology [21] developed by Intel is vector length-aware, precision-aware, FMA-aware, and divide-aware (we chose to not exploit the mask-aware capability of its methodology). We use this methodology as well as the post-processing scripts developed by NERSC [27] to collect the FLOPs for GPP in this paper.

On V100, the profiling tool, `nvprof`, is used to collect both FLOPs and the data movement. Particularly, we will use the `flop_count_dp`, `dram_read_transactions` and `dram_write_transactions` metrics from `nvprof`. The `flop_count_dp` gives us the direct FLOP count, and we obtain the ttoal data movement by scaling the total number of (read and write) device transactions by 32 (the size of each transaction is 32 bytes).

III. Results

In this section, we will first show some benchmarking results, demonstrating the importance of empirically measuring the Roofline bounds as well as collecting application performance data. Then with two examples, different arithmetic intensities and different memory access patterns, we will show how Roofline can be used in the context of performance portability to analyze the impact of architectural differences on codes' performance.

A. Establishing Accurate Roofline Ceilings

On KNL, with full-FMA codes (16 double-precision FLOPs per vector FMA instruction), 64 cores (common practice), 2 VPUs, and a 1.2GHz clock frequency, one may calculate the theoretical compute ceiling as:

$$64 \times 8 \times 2 \times 2 \times 1.2 = 2.46 \text{ TFLOP}/s \tag{5}$$

Note that on KNL, the peak performance is different than the marketing number 3 TFLOP/s mentioned in Section II-B because the clock frequency for full-AVX codes is reduced from the nominal 1.4GHz by 200MHz [14], and it is rare for applications to use 68 cores instead of 64.

Similarly, for full-FMA codes with 80 SMs, 32 FP64 cores per SM, 2 FLOPs per FMA instruction, and a 1.53GHz boost clock frequency, the V100 compute ceiling is:

$$80 \times 32 \times 2 \times 1.53 = 7.83 \text{ TFLOP}/s \tag{6}$$

On both KNL and V100, the no-FMA theoretical ceilings are calculated as one-half of the corresponding FMA ceilings.

As stated in Section II-C, vendor specification numbers may not be realistic performance bounds for an architecture. Rather, the ERT tool should be used to help establish a more accurate and realistically achievable set of Roofline ceilings (performance bounds). To that end, Figures 1 and 2 show the discrepancy between the theoretical numbers and the empirically measured compute and bandwidth bounds for KNL and V100. The sustainable HBM bandwidth, measured by ERT, is 341.8 GB/s on KNL and 828.8 GB/s on V100, which is about 30% and 8% lower than their respective vendor specifications (490 GB/s [12] and 900 GB/s [13]). Similarly, The empirical FMA ceiling is 2.7% lower than its theoretical counterpart on KNL (2.39 vs. 2.46 GFLOP/s), and is 9.8% lower on the V100 (7.07 vs. 7.83 TFLOP/s). When FMA is disabled in the compiler, there is a 21.9% gap between the empirical and theoretical no-FMA ceilings on KNL (0.96 vs. 1.23 TFLOP/S), but only a 9.7% gap on the V100 (3.54 vs. 3.92 TFLOP/s).

While the FMA:no-FMA empirical ceilings on V100 show the expected 2:1 ratio, no-FMA performance is well under 50% of FMA peak on KNL. To investigate this discrepancy, we experimented with hand-crafted intrinsics [28] kernels in ERT (nominally, ERT uses kernels written in C). With careful design of instruction pipeline and rigorous tuning of intrinsic calls, we are able to reach a 1.2 TFLOP/s no-FMA ceiling on KNL. However, as it is unproductive and unrealistic to rewrite large, real-world applications in intrinsics, all experiments in this paper will be based on compiled C code on both platforms. To that end, when analyzing performance in this paper, we will use ERT's empirical ceilings based on compiled C code on both KNL and V100 and incur the lower 959.5 GFLOP/s no-FMA ceiling on KNL.

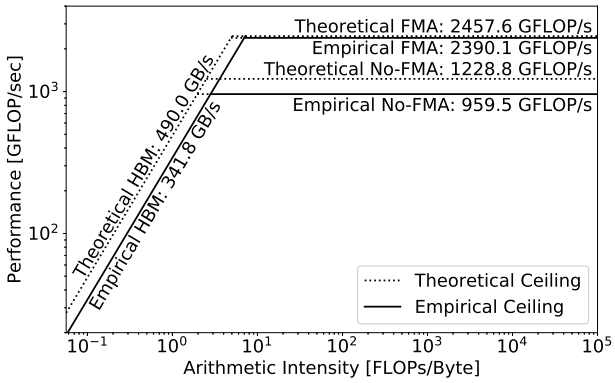

Figure 1. Performance ceilings on KNL: theoretical vs empirical. Theoretical ceilings are obtained either by Equation (5) or vendor specification data. Empirical ceilings (realistic performance bounds) are obtained through ERT. Observe the substantial difference in bandwidth and performance without FMAs.

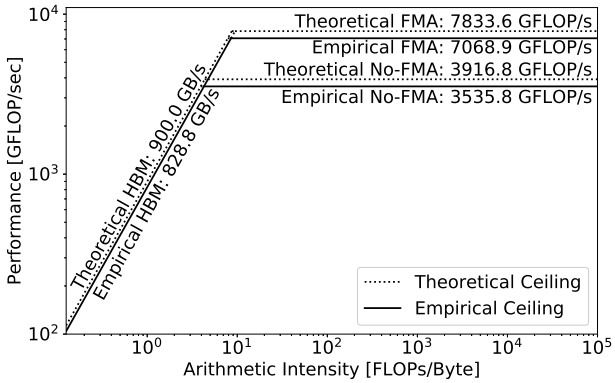

Figure 2. Performance ceilings on V100: theoretical vs empirical. Theoretical ceilings are obtained either by Equation (6) or vendor specification data. Empirical ceilings are obtained through ERT. Observe sustained bandwidth is very close to advertised, while there's more than a 10% difference in compute.

B. Accurately Modeling Divides in Roofline

Accurately counting the total number of FLOPs executed by a kernel is extremely important as it directly affects the calculation of both the arithmetic intensity and the GFLOP/s on the Roofline. If more FLOPs are executed than the source code enumerates, then arithmetic intensity and the true GFLOP/s are both higher than one would have nominally calculated. This increase in arithmetic intensity and performance can result in the kernel being pushed beyond the machine balance to the point where it is unencumbered by memory bandwidth, and the kernel could be much closer to a nominal compute ceiling than one realizes. For example, consider floating-point divides. Some ISA's implement divide with a single non-pipelined instruction while others rely on the compiler to generate a sequence of FMA's following a reciprocal estimate. In either case, simply counting the number of divides in the source code and equating them with multiples or adds is completely inappropriate as the former ignores pipeline stalls while the latter ignores additional instructions. We posit a similar conclusion of exponentials, logarithms, and trigonometric functions.

On both KNL and V100, with a sufficient optimization level, the compiler will emit a reciprocal estimate followed by multiple (perhaps iterative) floating-point operations in order to realize a floating-point divide. In order to quantify the impact of these, we will use several variants of the GPP kernel. The first is the stock GPP code in which the compiler emits multiple floating-point instructions per divide, and we empirically count the number of divides with SDE or `nvprof` ("Empirical GFLOPs" in Table I). In the second, although not numerically correct, we replace all source code floating-point divides with floating-point multiplies. Doing so ensures SDE counts all nominal "divides" as one floating-point instruction and provides the baseline as to what one would observe if simply equating "canonical" divides with multiplies ("Canonical GFLOPs" in Table I). Orthogonal to these variants, we vary the nw bound of the iw loop. Doing so allows us to increase the fraction of time in the inner loop and inhibit some compiler optimizations.

Although counting all FLOPs generated by the compiler for each divide is a step in the right direction, it may not be sufficient. On KNL, the compiler generates several mantissa- and exponent-related extract and insert instructions as well as comparisons that SDE does not count as FLOPs despite the fact they are executed in the vector units and displace other floating-point instructions. In the future, we will examine the impact of these floating-point instructions on sustained performance. However, in this paper, we will only examine the core "multiple", "add", "subtract", and "divide" floating-point instructions.

Table I shows both the canonical GFLOPs, where each divide is counted as 1 FLOP (second variant of GPP), as well as the empirical GFLOPs, where each divide is counted as however many FLOPs are actually executed by the architecture (first variant of GPP). For $nw = 1$, the difference between canonical GFLOPs and empirical GFLOPs is 14% on KNL and 28% on V100. However, as nw increases (and the code is increasingly dominated by the kernel), so too does the difference, reaching 35% on the V100. The implication is clear. Naively estimated FLOP counts derived from simply counting the source code FLOPs can be off by 35%. Moreover, what may have seemed like only 65% of peak performance is in reality 88% of peak performance.

Table I
TOTAL GFLOP COUNT FOR GPP ON KNL AND V100 AS A FUNCTION OF ARITHMETIC INTENSITY CONTROLLED BY nw. OBSERVE THAT PROPER ACCOUNTING FOR ALL FLOPS ASSOCIATED WITH DIVIDES (EMPIRICAL) IS IMPERATIVE.

Count	KNL			V100		
(GFLOPs)	nw=1	nw=3	nw=6	nw=1	nw=3	nw=6
Canonical	921.4	2354.7	4504.6	895.8	2329.1	4350.9
Empirical	1055.8	2834.5	5502.7	1151.6	3096.8	5886.5

Figures 3 and 4 visualize the impact of the FLOP un-

derestimates from Table I on GPP performance. In addition, they highlight the imperative of selecting the appropriate ceiling for performance analysis. The Bar labeled "1" highlights the large performance difference between FMA-enabled GPP performance and the bandwidth ceiling it's bound by. However, as SDE shows that only a small portion of the instructions are FMA's, comparisons against the FMA ceiling would be inappropriate. For the same reason, one should compare against neither the "Div FMA" nor the "Div No-FMA" ceilings (Bar 4 for "Div No-FMA" ceiling) as only a few of the floating-point instructions in GPP are divides. Rather, one should compare against the no-FMA ceiling.

With our focus on the no-FMA performance, Bar 2 (canonical performance vs. theoretical ceiling) could be considered, but given the discussion in Section II-C, the empirical no-FMA ceiling should be used instead. When one properly accounts for the divide-related FLOPs shown in Table I, GPP performance (red open triangle) moves diagonally to an increased arithmetic intensity with increased performance (blue open triangle). Although Bar 3 presents a more realistic estimate of the gap between GPP performance and its ceiling, The true performance of GPP is now captured by Bar 5 (empirical FLOPs and empirical ceilings) with a much smaller gap between the empirical performance and empirical ceiling.

In order to highlight the importance of comparing the correct performance metric against the correct ceiling, Table II presents application's architectural efficiency as calculated using Equation (1) for each of the different combinations in Figures 3 and 4. Clearly, Bars 1, 2, and 3 show very low performance portability (unnecessarily), while Bar 4 presents an impossible portability. Only Bar 5 (empirical FLOPs that account for those hidden FLOPs within a divide, coupled with empirical ceilings) delivers the appropriate architectural efficiency and ensures that any discussion of performance portability is not erroneously skewed.

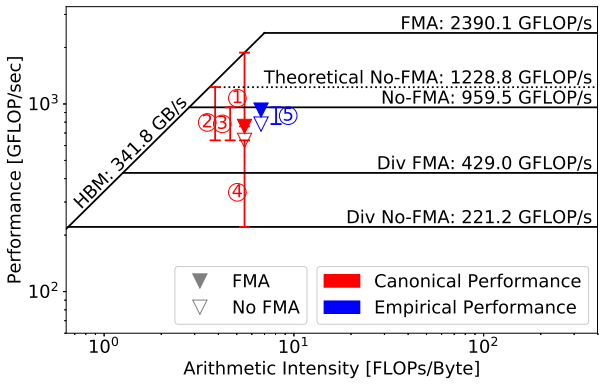

Figure 3. GPP performance for $nw = 6$ on KNL as a function of FMA code generation and how FLOPs are counted. Bars represent different pairings of performance and ceiling. Observe, only one pairing (5) is meaningful.

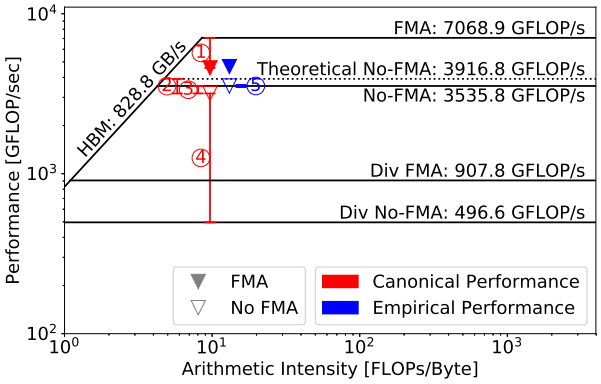

Figure 4. GPP performance for $nw = 6$ on V100 as a function of FMA code generation and how FLOPs are counted. Bars represent different pairings of performance and ceiling. Observe, only one pairing (5) is meaningful.

Table II
ARCHITECTURAL EFFICIENCY AND PERFORMANCE PORTABILITY OF GPP BASED ON DIFFERENT PERFORMANCE-TO-CEILING PAIRINGS (BARS 1-5)

Application Efficiency	Bar 1	Bar 2	Bar 3	Bar 4	Bar 5
KNL	40.41%	52.04%	66.65%	289.13%	81.42%
V100	64.89%	81.40%	89.79%	639.36%	99.96%
Performance Portability	49.81%	63.49%	76.51%	398.19%	89.74%

where...

	GPP Performance	Roofline Ceiling
Bar 1:	Canonical FMA	Empirical FMA
Bar 2:	Canonical no-FMA	Theoretical no-FMA
Bar 3:	Canonical no-FMA	Empirical no-FMA
Bar 4:	Canonical no-FMA	Empirical divide no-FMA
Bar 5:	Empirical no-FMA	Empirical no-FMA

C. Capturing Changes in Performance Bottleneck

Assured we can accurately account for the performance impact of floating-point divides in the context of the Roofline model and performance portability, we will now demonstrate that our methodology works across a range of arithmetic intensities. To that end, we simply increase the trip count of the iw loop in GPP by varying nw from 1 to 6. In theory, arithmetic intensity should increase (almost) linearly with the increasing nw.

In order to maximize the number of floating-point instructions in the dynamic mix and simplify analysis, we first disable the generation of FMA on KNL and the V100. Whereas on the V100, Figure 6 shows a strong transition (open symbols) as one linearly increases arithmetic intensity (note the log scale) from memory-bound performance to performance being bound by the no-FMA ceiling, Figure 5 shows that on KNL hitting the no-FMA ceiling is difficult, even with high arithmetic intensity and properly accounting for the floating-point operations. If one were to exploit FMA (solid symbols), then neither architecture sees the theoretical 2× speedup.

We believe much of these disparities in performance is attributable to the balance between instruction issue bandwidth

and floating-point throughput. For example, whereas KNL has two vector units, its front end can only fetch, decode, and issue at most two instructions (of any type) per cycle. So there is no spare instruction issue bandwidth for integer operations, compares, jumps (by definition, every loop has a jump), shuffles, and other non-floating point instructions. Every non-floating point instruction displaces a FMA, add, or multiply, thus making attaining the no-FMA ceiling impossible.

By contrast, each warp scheduler on the V100 (there are four per SM) can dispatch 32 threads per cycle to (four groups of) eight FP64 cores, eight load/store cores, 16 INT cores, and 16 FP32 cores. In other words, there is far more instruction issue bandwidth than FP64 throughput, with the surplus usable for loads, stores, or integer operations. As a result, it is much easier for Volta to deliver performance close to the no-FMA ceiling. The observed performance plateau when FMA is enabled suggests the generated code is not 100% FMAs, but rather is a mix of FMA, multiplies, and adds. This was confirmed with SDE code inspection. Our future work will examine methodologies and Roofline visualization techniques to highlight the effect of finite instruction issue bandwidth on performance on both CPUs and GPUs.

A more nuanced, yet critical facet of performance portability can be inferred from the AI, at which a transition from bandwidth-limited performance to compute-limited performance occurs. As one increases arithmetic intensity by increasing nw, we observe that the V100 transitions at $nw = 2$. Conversely, at this arithmetic intensity, KNL is still bandwidth limited. This difference highlights the imperative of using Roofline to understand performance portability, as one cannot simply compare fraction of compute peak on both architectures any more than one can compare fraction of memory bandwidth on both architectures. Rather, one must compare the V100 to the no-FMA ceiling and KNL to the HBM ceiling, and in general one must be cognizant of the ultimate (Roofline) bound on an architecture-by-architecture and kernel-by-kernel basis.

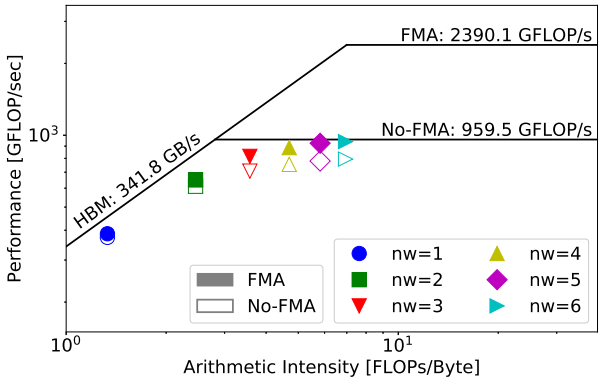

Figure 5. GPP performance on KNL as a function of nw. Observe the transition from bandwidth bound to a plateau below the no-FMA ceiling.

Table III presents architectural efficiency and performance portability (Equation 1) of the six variants of GPP (increased nw provides increased AI) with or without the use of FMA. In all cases, we use accurate measures of FLOPs and empirical ceilings. Across all values of nw, without FMA, performance is either bound by memory bandwidth or the no-FMA ceiling on both KNL and V100 (strongly). As a result, architectural efficiency is generally high on both machines and performance portability is consistently greater than 80%. Conversely, when FMA is enabled, the benefit at high nw is far less than 2× on either platform. As a result, architectural efficiency suffers moderately on V100 and strongly on KNL. This divergence in architectural efficiency leads to a reduction in the performance portability metric to just under 50% at $nw = 6$, and from an application standpoint, portability has been lost. However, if one were to incorporate non-floating-point vector operations into the Roofline-based architectural efficiency metric, some efficiency might be regained.

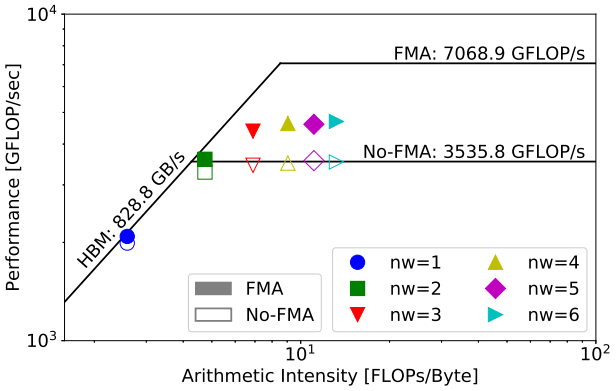

Figure 6. GPP performance on V100 as a function of nw. Observe the transition from bandwidth bound to either the no-FMA ceiling, or a plateau between the no-FMA and FMA ceilings.

Table III
ARCHITECTURAL EFFICIENCY AND PERFORMANCE PORTABILITY OF GPP VARIANTS OF nw (DIFFERENT AI) AND EXPLOITATION OF FMA.

FMA GPP performance against the FMA ceiling

Application Efficiency	nw=1	nw=2	nw=3	nw=4	nw=5	nw=6
KNL	84.98%	77.50%	66.77%	55.28%	46.56%	39.65%
V100	97.36%	91.50%	76.70%	65.44%	65.07%	66.38%
Performance Portability	90.76%	83.92%	71.39%	59.93%	54.28%	49.65%

No-FMA GPP performance against the no-FMA ceiling

Application Efficiency	nw=1	nw=2	nw=3	nw=4	nw=5	nw=6
KNL	82.06%	72.95%	73.74%	78.72%	81.28%	82.81%
V100	92.88%	92.88%	97.43%	98.91%	1.00	99.73%
Performance Portability	87.14%	81.72%	83.95%	87.67%	89.93%	90.49%

D. Accounting for Strided Memory Access

As originally envisioned, Roofline was focused on streaming (in particular unit-stride) memory access patterns. As such, it

nominally uses STREAM or some variant like ERT to calculate attainable bandwidth. Nevertheless, time and again, users have asked if, when, and how Roofline will support strided memory access patterns. In this section, we will demonstrate that Roofline already effectively accounts for small (up to two memory transactions) stride memory access.

We modified GPP's ig loop to implement a strided memory access pattern with a stride known at compile time, and an additional loop was inserted to ensure the computation was mathematically equivalent. To further expand the range of computations, we once again vary nw from 1 to 6 in order to vary arithmetic intensity. As usual, we simply record data movement to and from HBM using LIKWID and `nvprof`.

As we increase the stride, Figure 7 and Figure 8 show that on both KNL and V100, arithmetic intensity decreases. Across a range of strides and values of nw (range of arithmetic intensities), GPP performance on both KNL and V100 gracefully transitions from compute bound to HBM bandwidth bound.

On KNL, the cache line size is 64B. As such, stride-2 and stride-4 imply accessing two elements or one element per cache line, but accessing every consecutive cache line. This results in a straightforward $2\times$ and $4\times$ loss in spatial locality for stride-2 and stride-4 respectively. The loss in spatial locality directly translates into the observed loss in arithmetic intensity. However, as one proceeds to stride-8 (128B) and stride-16 (256B), one accesses every other and every fourth cache line on KNL. Ideally, one would hope the intervening cache lines are not loaded. Nevertheless, we observe a continued decrease in arithmetic intensity (increased data movement) for increasing stride that we attributed to hardware prefetchers loading neighboring cache lines.

Behavior on V100 is more complex. The compiler can generate different loads that can result in data being cached either in both the L1 and L2, or solely in the L2. Moreover, depending on how the data is cached, the memory fetch generated on a miss can be either 32B or 128B [29]. Figure 8 shows this effect as we vary stride and nw.

Regardless of nw, caching behavior, or memory fetch size, stride-2 (32B) results a $2\times$ loss in spatial locality with an expected and observed $2\times$ decrease in arithmetic intensity. Although we see the $2\times$ loss in arithmetic intensity as we proceed to stride-4 (64B) across all values of nw, we observe that stride-8 (128B) behaves very differently for larger values of nw. For the higher values of nw, stride-8 delivers roughly the same arithmetic intensity as stride-4 indicating only 64B is moved from memory. Conversely, for $nw = 1$, we observe a continued decrease in arithmetic intensity, although not quite the $2\times$ that would imply 128B transfers. As nw and $stride$ are known at compile time, it is possible the compiler generated different code for each variant that resulted in different L2 cache behavior.

In the future, we will examine larger strides (e.g. stride by 4KB) in order to quantify how hardware stream prefetchers and GPU compilers/caches behave differently in the context of Roofline. Specifically, with sufficiently large strides (e.g. 1 TLB miss per access), one could define multiple bandwidth ceilings.

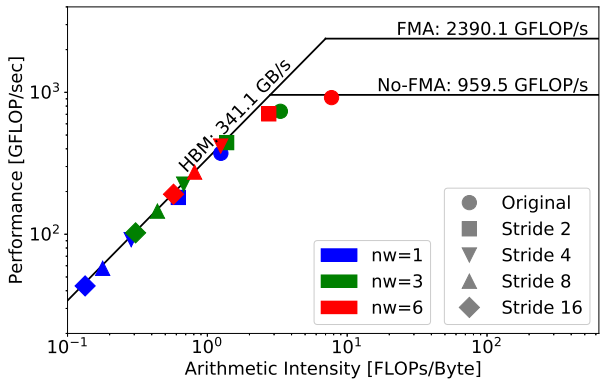

Figure 7. GPP performance on KNL as a function of stride and nw when FMA is enabled. Observe strided memory access simply changes arithmetic intensity (data movement) without changing bandwidth.

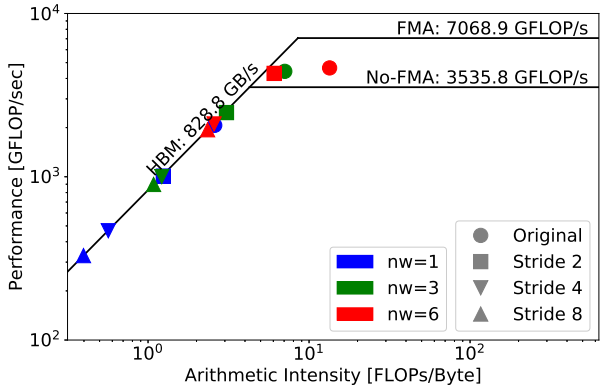

Figure 8. GPP performance on V100 as a function of stride and nw when FMA is enabled. Observe that just as on KNL, strided memory access simply changes arithmetic intensity (data movement) without changing bandwidth.

Table IV shows the architectural efficiency and performance portability for the GPP stride variants for a fixed $nw = 6$ (higher arithmetic intensity). Beginning with the original baseline (stride-1), we see that both KNL and V100 attain relatively low architectural efficiency and performance portability for this high arithmetic intensity kernel. As previously noted, this is more an artifact of computation than architecture (not all FLOPs in GPP are amenable to FMA). Conversely, as one increases stride (thereby increasing data movement and decreasing AI), we see a steady improvement in architectural efficiency on both KNL and V100, to the point where both machines exceed 98%. In such a regime, we attain high performance portability (both machines are similarly bound by the bandwidth Roofline).

IV. Conclusion and Future Work

In this paper, we discuss a number of important practical considerations in applying the Roofline methodology to quan-

Table IV
ARCHITECTURAL EFFICIENCY AND PERFORMANCE PORTABILITY OF GPP VARIANTS OF STRIDE SIZES, WITH FMA ENABLED, FOR $nw = 6$.

Application Efficiency	Original	Stride 2	Stride 4	Stride 8	Stride 16
KNL	38.40%	75.24%	98.39%	99.20%	98.00%
V100	65.64%	85.43%	98.81%	99.89%	-
Performance Portability	48.46%	80.01%	98.60%	99.55%	-

tifying application's architectural efficiency - a key ingredient in quantifying performance portability.

We extended the Roofline methodology to accurately account for hidden FLOPs associated with non-multiply/add instructions such as floating-point divide instructions on both KNL and V100 architectures. When properly accounted, a kernel's Roofline coordinate shifts (in a potentially architecturally dependent way) diagonally (increased raw performance and increased AI). Doing so can shift the kernel away from a bandwidth ceiling and towards a compute ceiling. Visualization of this motion ensures software developers are cognizant of how close the Roofline they may actually be.

In addition, we addressed one of the persistent questions associated with Roofline, namely how it accounts for strided or other memory access patterns. We showed that Roofline as originally defined, combined with an appropriate approach for empirically measuring data movement, easily captures moderate stride memory access and the performance remains consistent with the Roofline bound on both KNL and V100.

We found that these extensions of Roofline, in conjunction with empirical benchmarking of machine bandwidths and FLOPs (using ERT), were essential in using Roofline as the basis for a discussion of performance portability.

For future work, we may extend the accurate accounting for divides to other complex floating-point operations such as exponentials, logarithms, and trigonometric functions. We may also incorporate the effect of non-floating-point vector μOPs (compares, shuffles, gathers, stores, integer vector, etc...) as well as finite instruction issue bandwidth into our empirical Roofline methodology. Balancing this endeavor's goal of more accurately modeling performance must be matched against the potential for erroneously concluding high architectural efficiency equates with high performance portability.

As Moore's law fades, vendors are increasingly motivated to specialize core and instruction-set architectures in order to maximize performance. For AVX-512 and V100, this has produced complex instructions like VNNI and Tensor Cores. Just as FMA added substantial complexity and required examination of the resultant dynamic instruction mix in order to quantify its effect, we expect similar analysis to be required for these new and future instruction and core types.

Perhaps most valuable is applying this methodology to not only other computational motifs, but also to full applications. Ensuring our methodology is both correct on, scalable to, and still intuitive for full applications ensures the widest possible audience can reap the benefit and either improve implementation, develop better balanced architectures, or motivate new applied mathematics that avoids bottlenecks encountered by today's numerical methods on today's architectures.

V. ACKNOWLEDGEMENT

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. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research also used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

We would also like to acknowledge the contributions John Pennycook and Jason Sewall from Intel Corporation made to this paper, their valuable input on performance portability and insightful discussions on how Roofline can be effectively incorporated into the performance portability analysis.

REFERENCES

[1] DOE Office of Science. [Online]. Available: http://performanceportability.org

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[3] R. Hornung and J. Keasler, "The RAJA portability layer: Overview and status," *Lawrence Livermore National Laboratory, Livermore, USA*, 2014.

[4] J. Larkin, "Performance portability through descriptive parallelism," *Presentation at DOE Centers of Excellence Performance Portability Meeting*, 2016.

[5] S. J. Pennycook, J. D. Sewall, and V. Lee, "A metric for performance portability," *arXiv:1611.07409*, 2016.

[6] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[7] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware Roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.

[8] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, and H. Vincenti, "Applying the Roofline performance model to the Intel Xeon Phi Knights Landing processor," *International Conference on High Performance Computing*, pp. 339–353, 2016.

[9] T. Koskela, Z. Matveev, C. Yang, A. Adedoyin, R. Belenov, P. Thierry, Z. Zhao, R. Gayatri, H. Shan, and L. Oliker, "A novel multi-level integrated Roofline model approach for performance characterization," *International Conference on High Performance Computing*, pp. 226–245, 2018.

[10] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, 2017.

[11] H. Dreuning, R. Heirman, and A. L. Varbanescu, "A beginner's guide to estimating and improving performance portability," *3rd International Workshop on Performance Portable Programming Models for Accelerators (P^3MA) at the International Supercomputing Conference (ISC)*, 2018.

[12] Intel Knights Landing Processor. [Online]. Available: https://ark.intel.com/products/94034/Intel-Xeon-Phi-Processor-7230-16GB-1_30-GHz-64-core

[13] NVIDIA V100 GPU Whitepaper. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[14] Intel Xeon Phi™ Processor: Your path to deeper insight. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-processor-product-brief.pdf
[15] Empirical Roofline Toolkit. [Online]. Available: https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/
[16] General Plasmon Pole Kernel. [Online]. Available: https://github.com/cyanguwa/BerkeleyGW-GPP
[17] BerkeleyGW Code. [Online]. Available: https://berkeleygw.org
[18] J. Soininen, J. Rehr, and E. L. Shirley, "Electron self-energy calculation using a general multi-pole approximation," *Journal of Physics: Condensed Matter*, vol. 15, no. 17, p. 2573, 2003.
[19] C. Yang, B. Friesen, T. Kurth, B. Cook, and S. Williams, "Toward automated application profiling on Cray systems," *Cray User Group (CUG)*, 2018.
[20] Intel Software Development Emulator (SDE). [Online]. Available: https://software.intel.com/en-us/articles/intel-software-development-emulator
[21] Calculating "FLOP" using Intel SDE. [Online]. Available: https://software.intel.com/en-us/articles/calculating-flop-using-intel-software-development-emulator-intel-sde
[22] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," *39th International Conference on Parallel Processing Workshop (ICPPW)*, pp. 207–216, 2010.
[23] LIKWID Toolkit. [Online]. Available: https://github.com/RRZE-HPC/likwid/wiki
[24] NVIDIA Profiler nvprof. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html
[25] S. Williams, J. Deslippe, C. Yang, and P. Basu, "Performance tuning of scientific codes with the Roofline model," *Exascale Computing Project (ECP) 2nd Annual Meeting*, 2018. [Online]. Available: https://crd.lbl.gov/assets/Uploads/ECP18-Roofline-1-intro.pdf
[26] Intel VTune Amplifier Tool. [Online]. Available: https://software.intel.com/en-us/vtune
[27] SDE Parsing Script. [Online]. Available: https://bitbucket.org/dwdoerf/stream-ai-example/src/master/
[28] Intel Intrinsics Guide. [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/
[29] CUDA C Programming Guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html