Roofline Scaling Trajectories: A Method for Parallel Application and Architectural Performance Analysis

Khaled Z. Ibrahim, Samuel Williams, Leonid Oliker
Lawrence Berkeley National Laboratory
One Cyclotron Road, Berkeley, CA 94720, USA
{kzibrahim, swwilliams, loliker}@lbl.gov

***Abstract*—The end of Dennard scaling signaled a shift in HPC supercomputer architectures from systems built from single-core processor architectures to systems built from multicore and eventually manycore architectures. This transition substantially complicated performance optimization and analysis as new programming models were created, new scaling methodologies deployed, and on-chip contention became a bottleneck to performance. Existing distributed memory performance models like logP and logGP were unable to capture this contention. The Roofline model was created to address this contention and its interplay with locality. However, to date, the Roofline model has focused on full-node concurrency. In this paper, we extend the Roofline model to capture the effects of concurrency on data locality and on-chip contention. We demonstrate the value of this new technique by evaluating the NAS parallel benchmarks on both multicore and manycore architectures under both strong- and weak-scaling regimes. In order to quantify the interplay between programming model and locality, we evaluate scaling under both the OpenMP and flat MPI programming models.**

***Index Terms*—Roofline Model, Performance Analysis, Parallel Scaling, OpenMP.**

I. Introduction

Dennard scaling enabled energy-efficient scaling from one CMOS process generation to the next [1]. This scaling trend was exhausted by 2005 at which point computer architects were forced to reevaluate architectural paradigms and prioritize those techniques that maximized energy efficiency rather than single thread performance. This revolution in computer architecture gave birth to multicore and eventually manycore processors where dozens of cores are integrated on a single chip. Whereas multicore architectures tended to be based on large superscalar cores with large shared last-level caches, manycore architectures were further energy-optimized to maximize performance and concurrency through the use of simple cores, wide vector units, and a sea of private caches that motivated data parallelism and coarse-grained partitioning of computation.

The transition from supercomputers built from single-core nodes to multi- and manycore nodes significantly complicated performance optimization and analysis as new programming models like OpenMP were deployed, scaling regimes could be applied at the core, node, or system level, and on-chip contention emerged as a potential bottleneck to performance. Many programmers were incentivized to experiment with OpenMP only to find that performance paled when compared to flat MPI on a chip (one process per core). Without a performance analysis methodology capable of understanding the interplay between programming model, locality, concurrency, and architecture, they were generally unable to effectively remedy their performance gap.

In 2008, the Roofline model was created to understand the interplay between locality, bandwidth, and computation across a wide variety of architectures [2], [3]. However, prior Roofline research was typically used to understand the ultimate performance potential of a machine and not the interplay between concurrency and locality — a key metric in the application optimization and architecture co-design processes.

In this work, we introduce the roofline scaling trajectory technique to analyze the performance on multi/many-core architectures. We show that these trajectories help in assessing the application leverage of the cache hierarchy and how will the architecture is helping or impeding the scaling of an application. We present the application of the proposed method on the NAS Parallel Benchmarks [4]. The trajectory analysis enables applying the roofline method to workloads with integer or arbitrary user-defined operations.

The rest of this paper is organized as follows: We discuss the motivation for this work in § II. We introduce the roofline scaling trajectory method in § III. In § IV, we present the experimental setup and the workloads. In § V, we analyze the application of the proposed method on the NPB suite. We conclude in § VII after discussing related work in § VI.

II. Motivation

Performance analysis is notoriously difficult especially in the many-core era. It is becoming critical to driving the system efficiency because architectural efficiency from improvements in transistor miniaturization, lithographic advancement in CMOS technology, is slowing down.

Many-core architectures have multiple features that could influence the application performance. Among notable features are simple core design, reliance on distributed caches, and the adoption of memory technologies that are throughput optimized. Simpler core designs typically necessitate the reliance on SIMD/vectorization as a means for improving instruction level parallelism. The use of distributed cache/local storage enables servicing a larger number of memory requests concurrently. On the other hand, maintaining coherence between these distributed caches could result in reduced effective cache capacity, due to the replicated state of frequently read data, or longer latency to service memory write requests due to the multi-hop transactions in the directory-based cache coherence protocols that are typically used for distributed caches. Another challenge is feeding data to many-core architectures to make them busy. Some memory technologies are more optimized for throughput than latency. These memory technologies make it possible to handle many concurrent requests but not to improve on the latency to service the request and may increase performance variability as well.

Understanding the performance limits intuitively is a challenge with these architectural developments especially if we treat each code independently. In HPC, relatively few computational patterns dominate scientific computing. The computational dwarfs report from Berkeley [5] provides an excellent survey of these patterns. We can focus the analysis on few computational patterns and gain insight into a wide range of usage patterns.

III. Evaluation Method

In this paper, we introduce the roofline scaling trajectory technique for analyzing the performance and scalability of parallel applications. We also extend the use of roofline plots from being solely used for floating-point operations to any user-defined operation. Such an extension allows analysis of a broader set of irregular integer-based applications.

In the context of this work, our primary focus is assessing the change in application behavior while migrating from multi-core to many-core architectures (lightweight cores, distributed private caches, etc...). Rather than relying on theoretical limits and models for application and machine characterization, we rely on empirical measurements including the use of LIKWID to measure the DRAM performance counters [6], and the Intel SDE for instruction analysis [7].

A. Roofline Scaling Trajectories

Weak-scaling and strong-scaling are typically used to assess the efficiency of utilizing the hardware resources as we increase the level of concurrency. When weak-scaling, one aims for constant run time (linearly increasing flop rates) for problem sizes proportional to concurrency. Conversely, when strong-scaling, one hopes for run times inversely proportional to concurrency for fixed problem sizes. Unfortunately, it is often difficult to ascertain the root cause of any sub-linear scaling behavior in either regime. Often one employs a variety of tools to identify which components of the system or application are either not scaling or have become the bottleneck. These includes tools to analyze the memory hierarchy or the impact of communication on total run time. The roofline scaling trajectory focuses on visualizing the scaling behavior and identifying the effects of locality and limited cache capacity on the observed application performance. Additionally, these

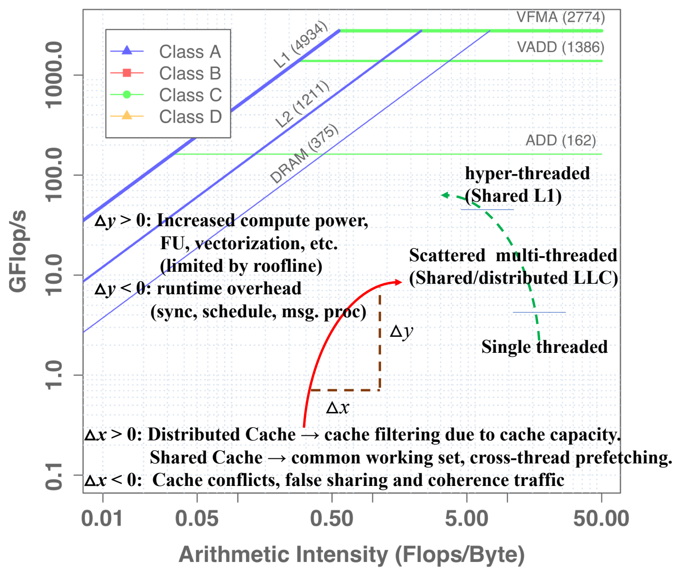

Fig. 1: Roofline Scaling Trajectory. As we double the computational resources we expect a corresponding doubling of performance (Δy), but the observed performance is typically influenced by the change in computational intensity (Δx).

plots could identify the potential advantage of leveraging a particular cache hierarchy for a given computational pattern and could also steer optimization efforts towards those with the highest possible gains.

We leverage the roofline plots [2], which are used to analyze the observed performance of an application against machine limits. The x-axis is the arithmetic (or computational[1]) intensity, computed as a ratio of floating point operations to transferred bytes from the memory system. The y-axis is observed performance. Machine limits, such as flop rate or memory bandwidth define the maximum attainable by an application, which is a function of the application's arithmetic intensity.

The roofline scaling trajectory tracks the scaling of an application. Scaling can be decomposed into transitions of Δx and Δy. As we double the level of concurrency, we ideally expect Δx to be 0 (no change in cache locality) and $\Delta y = 2\times$ (linear increase in performance). This idealized scaling behavior is also assumed to be unconstrained by the overheads of the language, synchronization, scheduling, message setup, etc. In reality, the nature of distributed vs. centralized cache hierarchies may constrain locality, and the limitations of finite memory and cache bandwidths may constrain the performance gain to less than $2\times$.

Although one could look at raw values extracted from performance counters to identify the causes of limited scalability, the process could be cumbersome and error-prone. Visualization allows efficient and potentially more intuitive

[1]We extend the arithmetic intensity use in the Roofline Model by using the general term computational intensity, where a computation could be any application defined operation.

analysis. The process requires building intuition about the scaling behavior possible moves and their interpretation.

We describe the architectural and scaling behaviors for each of the four possible directions an application may move with increased concurrency.

$\Delta x > 0$: When strong scaling, this transition is indicative of increased temporal locality (the number of floating-point operations has remained constant, so data movement must have decreased). This behavior can occur with distributed cache hierarchies as the aggregate cache capacity increases with increased concurrency. Eventually, a working set can fit in cache, and DRAM data movement is reduced. The key is to maintain such behavior is to avoid thrashing the working set. If the cache hierarchy involves some sharing, each processing element should limit the working-set in the last level cache to its fair share.

$\Delta x < 0$: This transition is an indication of a loss of temporal locality during the scaling experiments, which could happen for various causes. First, when strong-scaling on shared cache hierarchies, if the per-thread working set is not sufficiently small, the aggregate working set of all threads can eventually exceed the last-level cache capacity resulting in superfluous data movement. A similar effect can occur when weak scaling. Although the application strategy in dealing with the memory hierarchy is very influential on performance, the memory organization and whether it has a distributed or shared hierarchy could have a significant impact on the ability to leverage locality and as such the scaling behavior. For instance, a distributed cache is useful in reducing the interference or cache thrashing effect because it provides some locality isolation. On the other hand, distributed caches are challenged when attempting to capture a large shared working set as the same data may be replicated in multiple private caches. Effectively, the cache capacity could look much smaller than the aggregate capacities of individual caches.

$\Delta y > 0$: The increase in the y direction reflects the utilization of the excess compute resources. The transition could exceed the added compute resources if such transition carries an accompanying $\Delta x > 0$. In the Roofline Model, the Δy transitions are typically bound by one of the system limits (memory or compute), depending on the computational intensity.

$\Delta y < 0$: A decrease in the y direction indicates some runtime or architectural overhead that impedes increased performance. Such effects can arise from programming model (e.g. excessive synchronization overheads, substantial communication setup time, etc.), or contention when accessing architectural resources (e.g. L1/L2 or reorder buffer in hyper-threaded architectures).

Although each concurrency scaling transition could see independent effects on locality and performance (positive or negative), there is also a potential coupling between Δx and Δy transitions. If the application performance turns over at high concurrency ($\Delta y < 0$), our roofline trajectory could identify whether a $\Delta x < 0$ effect is causing the observed $\Delta y < 0$ effect on performance. This becomes more apparent when working close to the roofline at full concurrency.

For this paper, we will focus on the scaling trajectory considering only the DRAM arithmetic intensity (locality effects captured by the last level of cache). We designate pure Δy scaling ($\Delta x = 0$) as an ideal behavior that would require some performance isolation between concurrent activities in the system. We identify three sources of this performance isolation:

- Algorithms: This performance isolation is the hardest to achieve except in special cases, where the application developer decomposes the problem into smaller tasks that maximize the data reuse within the cache hierarchy. Lack of algorithmic isolation could cause severe degradation in computational intensity as we scale computation.
- Programming models: Distributed programming models such as MPI are known to force the programmer into the mindset of creating isolated tasks that communicate only at specific times explicitly using communication calls. This decomposition proves beneficial not only in a distributed computing environment but also in a shared memory environment where NUMA locality exists.
- Architectures: Cache hierarchy and memory could provide performance isolation through the use of distributed caches. As such, each core has its own private space and no cache thrashing is possible.

B. Non-Floating-Point Roofline Trajectories

Roofline analysis typically targets floating-point computations. This implies the use of floating-point operations as both the performance metric as well as a numerator of the arithmetic intensity ratio. However, many applications do not relying on measuring or performing floating-point operations, and thus one is motivated to use non-floating-point metrics (MIPS) or application-specific metrics. For instance, whereas one often uses traversed edges per seconds (TEPS) as a performance metric in graph analytics, one might use random numbers generated per second in other fields. For such applications, it is often difficult to define the roofline for operations based on the raw performance characterization of the underlying architecture and thus we may define performance and intensity relative to these metrics using application-specific internal performance metrics. Nevertheless, we still find applying the scaling trajectory analysis on roofline plot as an insightful tool for performance analysis, and will show that the scaling behavior of applications with non-floating-point metrics could have the same pitfalls and trends observed in floating-point intensive applications.

IV. Experimental Setup

A. Benchmarking Suite

In this paper, we used the NAS Parallel Benchmarks (NPB) [4] for evaluating scaling effects among architectural variants. These benchmarks represent a broad set of computational patterns. FT represents spectral methods, CG for sparse linear algebra, LU for solving a regular-sparse lower and upper triangular system, MG for multi-grid PDE solver

TABLE I: Systems used in this study.

	Cori I	Cori II
Processor	Intel Haswell	Intel KNL
Clock (GHz)	2.3	1.4
NUMA$\times$Cores	2×16	1×68
DP GFlop/s	1178	2611
D\$/core(KB)	64+256	64+1024 (per tile, 2 cores)
LL\$/chip(MB)	30	
Memory (GB)	128 DDR4	16 MCDRAM+ 96 DDR4
System and System Software		
Nodes	2,388	9,688
Interconnect	Dragonfly	
Compiler	Intel 18.0.1	

using a hierarchy of meshes, in addition to multiple structured grid methods in the mini-apps SP, BT. NPB also includes unstructured adaptive mesh benchmark, called UA. Similar to CG, its memory access pattern is irregular. In addition to the floating-point based scientific applications, NPB includes an integer sorting benchmark, IS, which is used in many scientific applications. For performance measurement, NPB measures the performance based on the rate of performing mega operations per seconds.

B. System Setup

We use NERSC's Cori supercomputer as our evaluation platform. Cori is a Cray XC40 system that employs both multi-core Intel Haswell and many-core Intel Xeon Phi "Knight's Landing" processors. The system leverages the Cray Aries network to connect 2,388 Haswell nodes and 9,688 KNL nodes. Table I gives brief description of the architectural features of the Cori nodes.

V. Trajectory Analysis for Computational Kernels

One of the most critical factors for performance is the efficiency in utilizing the cache (or memory) hierarchy. On roofline plots, the cache efficiency manifests as the computational intensity. The higher the efficiency of utilizing the cache, the higher the computational intensity. Obviously, the intensity is influenced by the kind of computation at hand and whether it is feasible to tile the computation to better leverage temporal locality. For a kernel that efficiently exploits the cache, we expect our Roofline trajectory to move vertically. Moreover, changing the problem size should have minimal impact on the computational intensity, unless the problem could fit entirely within the last-level cache.

Reduction in the arithmetic intensity ($\Delta x < 0$ transition) is not a desirable behavior, whether we do strong or weak scaling. A $\Delta x > 0$ transition while scaling could result from the increase of cache capacity. It could also result from better problem decomposition while scaling. While such a transition has favorable performance advantages, these advantages may be difficult to maintain asymptotically.

A. Strong-scaling Trajectories

Strong scaling involves changing the computational resources while fixing the problem size per node. A real test to the algorithm temporal-optimization is whether it could manifest a vertical trajectory on an architecture with a shared last-level cache.

A couple of kernels show such behavior, using their OpenMP implementation: MG and FT. For both, we notice that the trajectories are dominated by pure Δy transitions. Only when we start using hyper-threading, do we observe operational intensity degradation due to the increase in data movement. In Figure 2, we present the benchmarks that exhibits relatively regular behavior. For MG, the KNL and Haswell arithmetic intensities are close to each other due to the exploitation of tiling. The scaling is dominated by vertical transitions that continue until hitting the memory roofline.

For FT, we observe that the arithmetic intensity increases with the amount cached data. Given that the computation involves a transpose operation, we observe that the shared cache, in Haswell, helps in doing it efficiently. With the distributed cache in KNL, we observe a Δx transition, change in arithmetic intensity, with the increase in the data set size. The scaling for CG, with irregular access patterns, benefits from the shared cache if the data fits in cache. Concurrency in accessing the shared cache causes a diagonal scaling behavior ($\Delta x \neq 0$) if a significant portion of the active dataset fits in cache, *i.e.* Class A. With Distributed cache, we observe $\Delta x > 0$ while strong scaling and the performance improves as we increase the level of concurrency because we're increasing cache capacity, while not suffering from cache thrashing. Eventually, the scaling in both architectures converge to the same computational intensity for sufficiently large problems. Larger datasets cause a vertical scaling transitions that are bounded by the memory latency.

In Figure 3, we shows a different set of benchmarks with a stronger influence of the cache hierarchy. For mini-application, such as SP and BT, the scaling trajectory starts with an initial positive Δx due to the increase in cache capacity[2]. Increasing concurrency causes $\Delta x < 0$ scaling transition associated with $\Delta y > 0$. The challenge in these mini-applications is the difficulty in managing locality beyond a single kernel. Moving from one computational phase to the other would result in evicting the working set from the cache. For these kernels, the KNL distributed cache provides a lower initial computational intensity that keeps improving as we increase the concurrency level. Interference in the L1 cache reverses such scaling trend, once we start using hyper-threading. LU exhibits the most significant Δx transitions during strong scaling on the KNL due to the performance isolation in the distributed cache. LU uses scratch memory to condense scattered data from a global data structure. Some of data structures, with 5 elements in its innermost dimensions, exhibit false-sharing among threads creating either prefetching or interference effect, depending on the cache hierarchy.

In Figure 4, we show the same set of applications implemented using MPI. We notice improvement of the peak

[2]The initial $\Delta x > 0$ is due to scattering threads across sockets. With dual socket systems, the full cache capacity is fully utilized only if we have at least two threads.

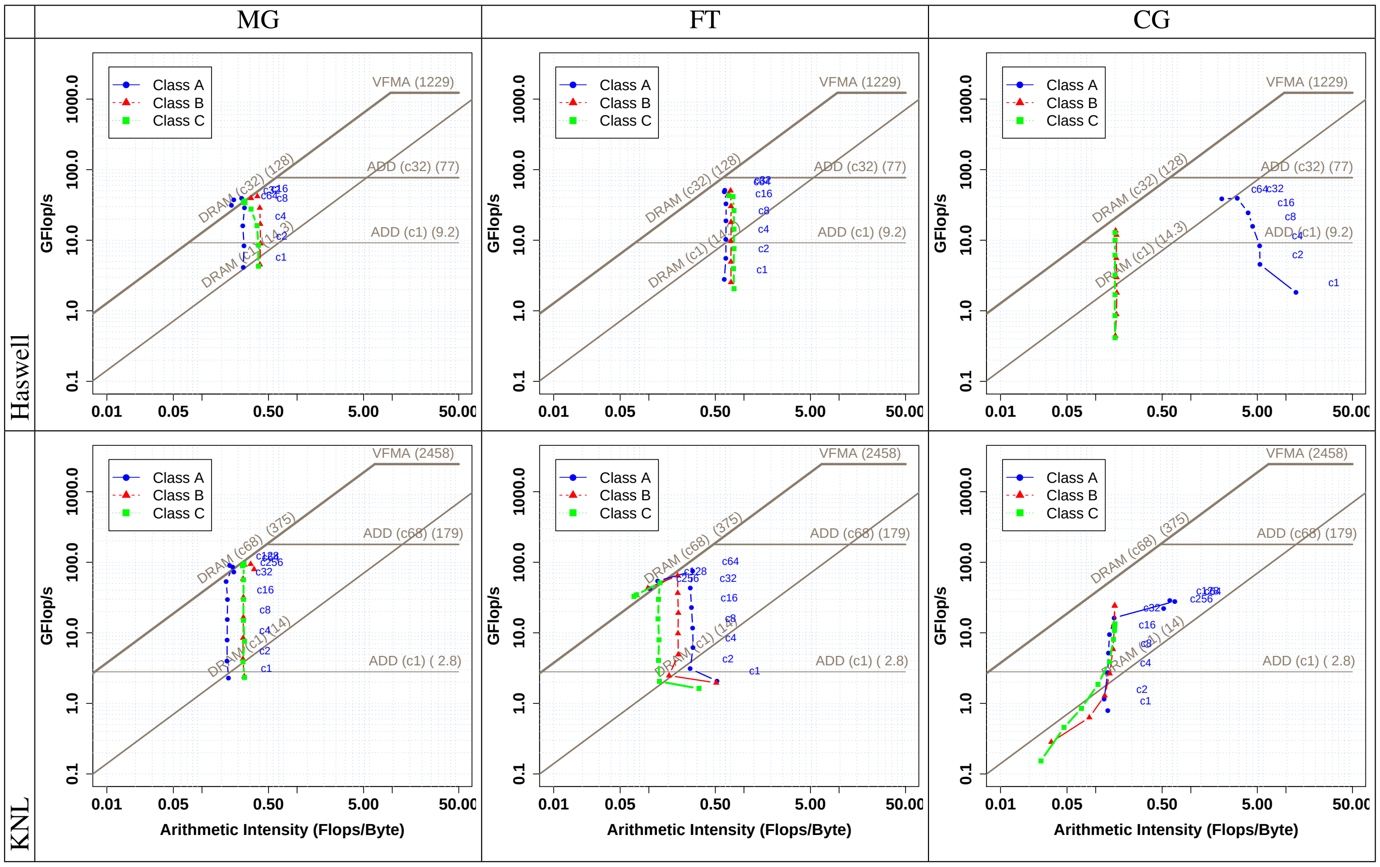

Fig. 2: Scaling trajectories for OpenMP-based kernels MG, FT, and CG on KNL vs. Haswell. MG and FT exhibit regular scaling behavior. CG benefits from the shared cache if the data fit within the LLC.

performance, especially with the shared cache of Haswell. In this case, the programming model isolation, *i.e.* the use of MPI that eliminates the implicit communication between computing processors, partially helps in addressing the scaling problem, but the trends of computational intensity change with the concurrency level is not eliminated. MPI can help in partitioning the problem, which reduces the dataset interference by each compute processor, but it cannot eliminate the loss of temporal locality due to successive kernel invocations.

The cache hierarchy plays a major role in the Δx transitions in the scaling trajectory, which influence the Δy transitions for this set of applications. Distributed caches provide the performance isolation that facilitates reducing the Δx transitions.

In Figure 5, we show the application of our roofline trajectory methodology for applications with user-defined operation. In such case, we can determine only the memory rooflines, but the computational capabilities would need to be user-defined. Both IS and UA generate irregular accesses. Following other floating-point benchmarks, UA experiences scaling that involves $\Delta x < 0$ as we increase the concurrency on Haswell. Although sorting algorithms, such as IS, typically generate irregular access, we do not notice much Δx change while scaling due to the use of bucket sorting. The cache filters most of the irregular accesses and the traffic to memory remains stable while strong scaling. We also notice severe performance degradation once we start exploiting hyper-threading. Significant performance drop is observed on the KNL architecture.

B. *Weak-scaling Trajectories*

Weak scaling involves changing the problem size as we increase the computation resources. NPB increases problem sizes by $4\times$ as we change the problem size from one class to the other up to class C. Weak scaling plot reduces the impact of runtime overheads on performance.

Using our roofline trajectory methodology to assess weak scaling could provide insights into the benchmark dependence on temporal locality. As shown in Figure 6, LU has a $\Delta x > 0$ transitions during weak-scaling on the KNL cache hierarchy. Haswell's trajectory follows the opposite trend $\Delta x < 0$. This behavior indicates that LU has a common working set between the compute threads. This attribute is difficult to capture using code inspection because such common dataset stems from false sharing. Interference could reduce the effectiveness of keeping this working set in the cache. Only a few applications have weak scaling without a change in Δx, *e.g.* IS.

The weak-scaling analysis on KNL in particular could provide insights into what kind of cache interference is happening; whether the interference is between the threads involved in the computation, or the cache eviction happens due to changing phases. On KNL, a $\Delta x < 0$ when threads are scattered indicates self-interference, *i.e.* cache evictions between different phases of computations. We observed such behavior

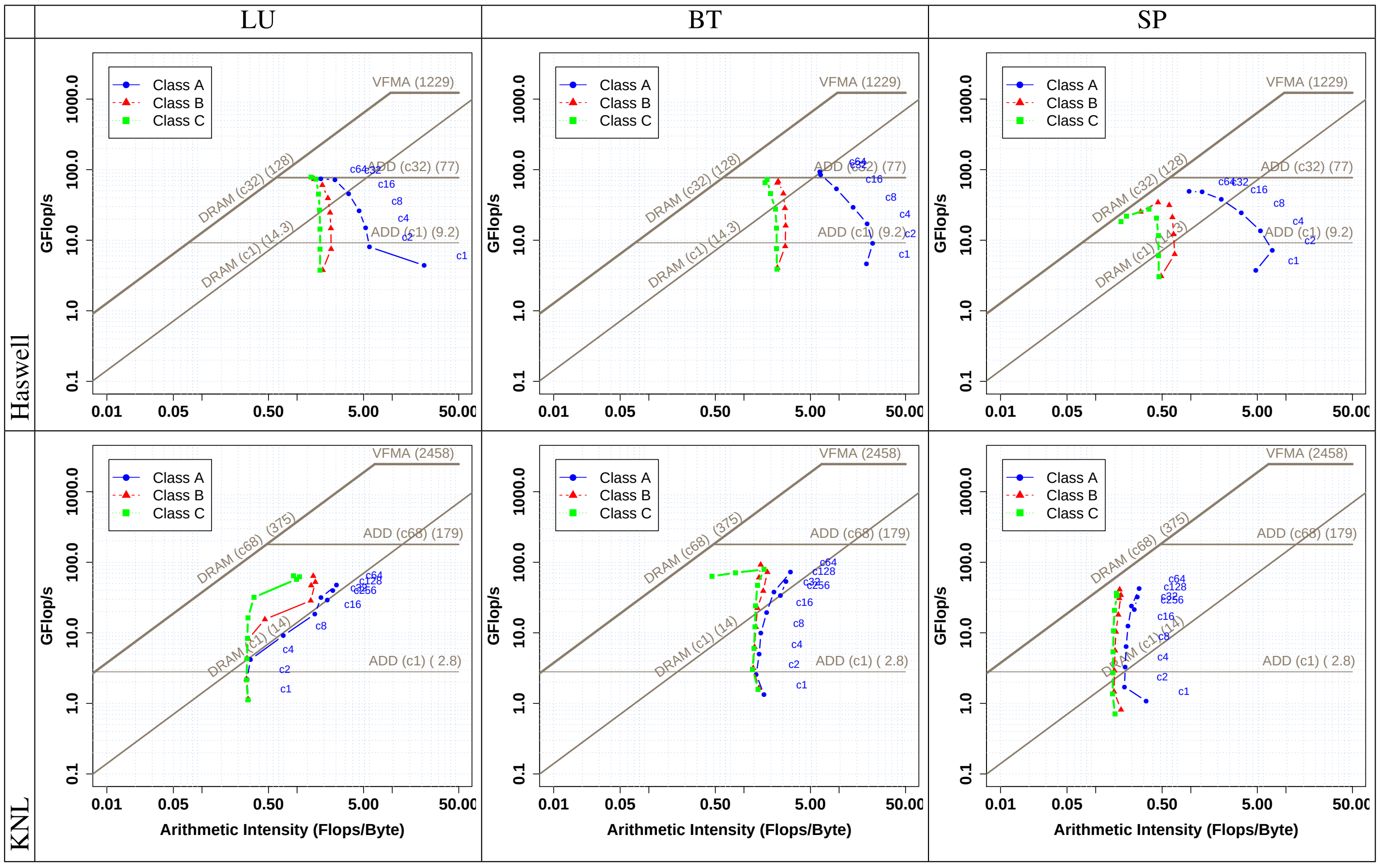

Fig. 3: Scaling trajectories for OpenMP-based FT kernel and mini-applications BT, SP on KNL vs. Haswell. These benchmarks show sensitivity to cache hierarchy and exhibit significant Δx transitions.

with FT. For instance, SP self-evict part of its working set in weak-scaling experiments, while BT does not experience such behavior. A $\Delta x > 0$ indicates a shared working set between the threads, where one thread effectively prefetches data for other threads.

C. Limits of Vertical Transitions

We have observed that most applications are not exceeding the computational roofline of scalar operations. While this may not be a limitation if the application is memory bound, some applications are not memory bound on either the Haswell or KNL systems. We also notice that the performance relative to peak scalar performance is lower on KNL compared with Haswell. Not leveraging vectorization is a severe limitation especially on the KNL architecture where 16× of the full computational power is at stake.

Another critical factor is how effective is the compiler in vectorizing the application code, or how amenable is the application to vectorization. In Figure 7, we report the percentage of scalar floating-point operations, which were not vectorized, as a percentage of the total floating-point operations. The reported values are based on Intel SDE profiling. The impact of having a large percentage of scalar operations is significant on architectures where vector processing units are responsible for executing the scalar operations. On KNL, only one of the two VPUs can execute scalar and legacy vector operations [8], such as SSE. We notice that well-vectorized benchmarks, such as MG and FT, are memory bound. As such, they did not benefit much from their efficient vectorization. Benchmarks that are compute bound, such as LU and BT, have a large percentage of scalar flops. Given the generated code may not fully utilize functional units in performing application floating point operations, we need to consider the efficiency of utilizing vectorization units. We define vectorization efficiency as

$$vec_{eff} = \frac{(total\_flops - scalar\_flops)}{(vector\_uops \times max\_flops\_per\_uop)}$$

The numerator is a measure of the application vectorized instructions. The denominator is proportional to the vector micro-ops (μops), involving various vectorization widths, which flow through the VPU units. These μops include, in addition to floating-point computations, integer, shuffle, broadcast, convert, etc. For the studied application, non floating-point instructions are generated automatically by the compiler, but their occupancy of the VPU units reduces the effective throughput observed by the application. We used Intel SDE [7] for measuring the application level scalar and vector instructions, while we used performance counters measurements reported by LIKWID to measure the μops.

This vectorization efficiency metric averages the efficiency of exploiting SSE, AVX2, AVX512 (with and without masking), across different regions of the code. The effectiveness of vectorization depends on many factors including data alignment, loop iteration count, the complexity of the control flow

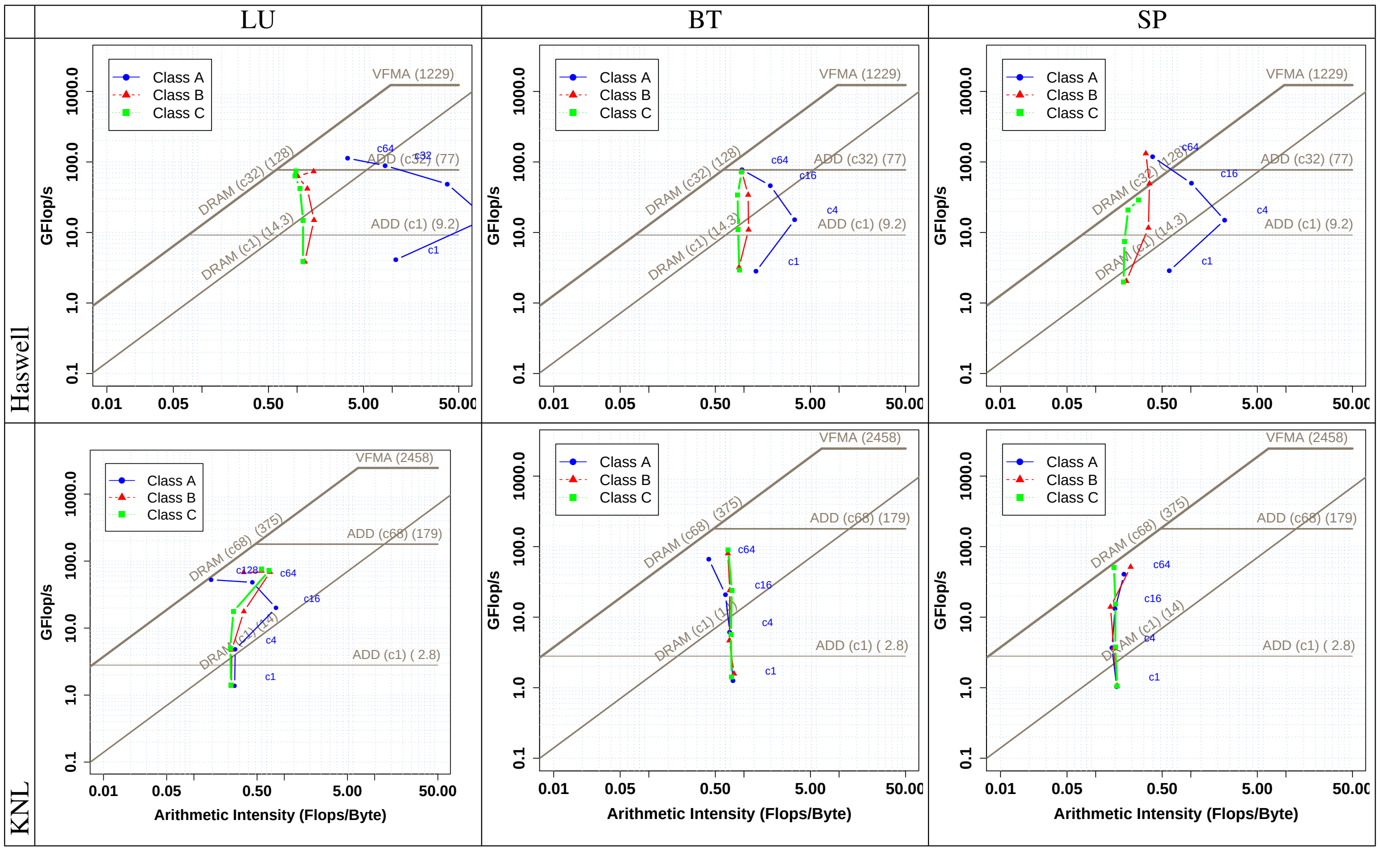

Fig. 4: Scaling trajectories for MPI-based kernels LU, BT, and SP on KNL vs. Haswell, running on a single node. We observe improvement in performance isolation on shared cache but we generally notice similar scaling trends to those with OpenMP.

instructions, etc. Given that most of the explored codes are written in Fortran with compile-time known loop bounds, the compiler's ability to vectorize these codes is usually high. In fact, we measured higher vectorization success rate on these codes compared with a variant of the code written in C. The flops per instruction are 8 (or 16 for fused instructions[3]) double precision flops per vector instruction. The results, shown in Figure 8, use an estimate of 8 flops per instructions. We observed the highest efficiency for FT and MG approaching up to 54%. For LU, BT, and SP the vectorization efficiency is 22-25%. In general, the vector floating-point μops are less than 50% of the total μops. For CG, we measured the lowest Floating-point μops, roughly 19%. The remaining μops are used to gather and align the data for the efficient vector fused-multiply-add computations. Such low average efficiency in vectorization correlates with the hovering below the rooflines for multiple applications.

Another performance limiting factor on KNL is the runtime overheads. To quantify such overheads, we used Coral Clomp [9] benchmark for assessing OpenMP performance. As shown in Figure 9, the barrier synchronization latency using 64 threads is $4us$. This latency roughly doubles at 256 threads. Generally, we observe the runtime latency to be proportional to the log of threads on KNL. The runtime performance is also generally better on Haswell compared with KNL for the same level of concurrency, which is expected because Haswell has faster cores. On Haswell, the latency increase rate with concurrency is lower than the log of thread concurrency. This behavior suggests that scaling is probably benefiting from the cache hierarchy on Haswell better than KNL.

[3]AVX512 instruction could perform up to 16 double precision floating-point instructions using fused multiply-add instructions.

D. Architectural Exploration for Performance

The presented trends show the challenge in the transition from one architecture to the other, even if they share more or less the same ISA. The quest for performance could follow multiple fronts. First, some applications should put more efforts into refactoring the code to better leverage the cache hierarchy at hand. For applications composed of multiple computational phases, this could be a difficult task to achieve.

The second front would be to explore the architectural impact on the various application computational pattern. In this work, we explored two design points, one is characterized by sharing the LLC, and the other with minimal sharing of LLC. While we understand that distributed caches are more scalable in handling many-core architecture, the level of sharing LLC (i.e., the number of cores per tile could be changed). Intel Haswell could be viewed as 2-tile (socket) architecture with 16 cores per tile (socket), while KNL has a 36 tile architecture with 2 cores per tile. Between these two design points, there are other under-explored design variants.

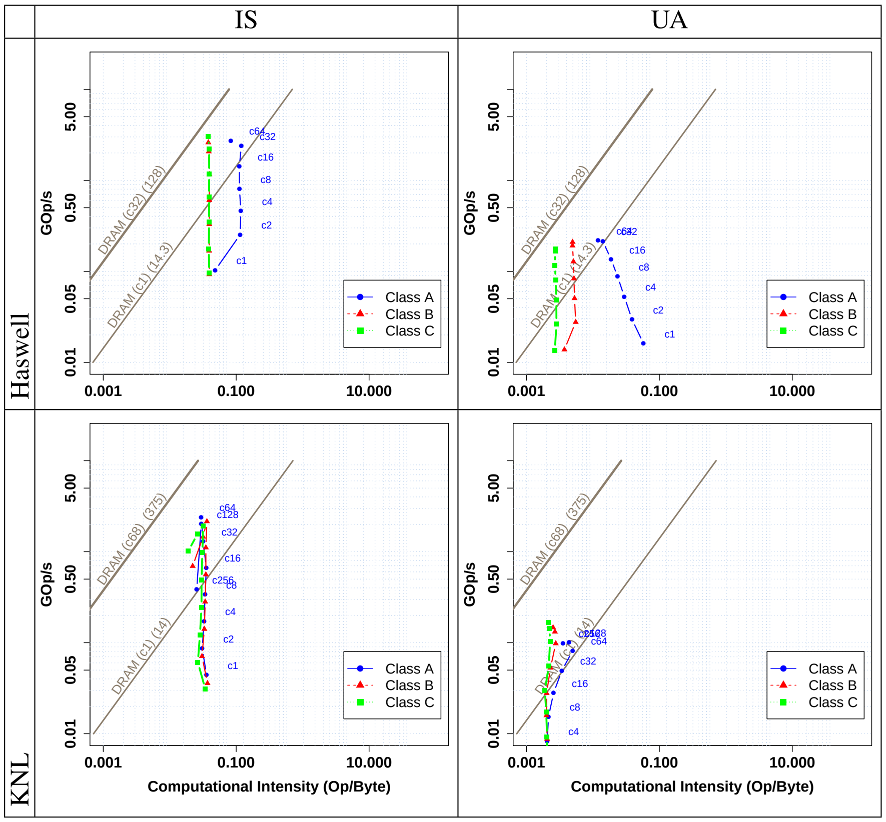

Fig. 5: Scaling trajectories for OpenMP-based user-defined op kernels: IS and UA on KNL vs. Haswell. UA's irregular access pattern benefits from Haswell's shared cache.

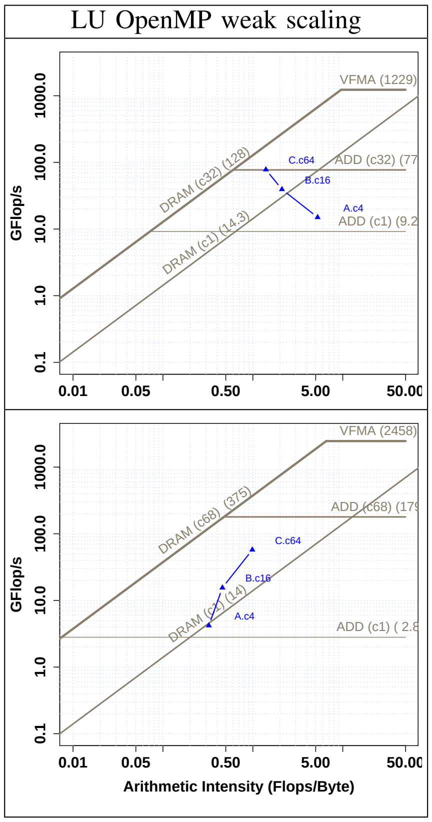

Fig. 6: KNL provides better weak scaling than Haswell, but starts from a lower computational intensity.

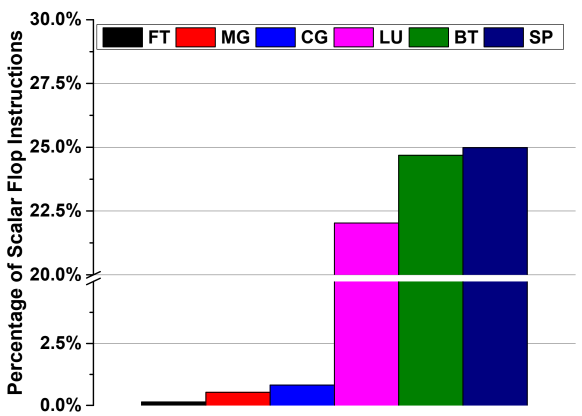

Fig. 7: Scalar operation percentage for NAS NPB. Some applications, such as BT and SP, have a significant fractions of their floating-point operations not vectorized.

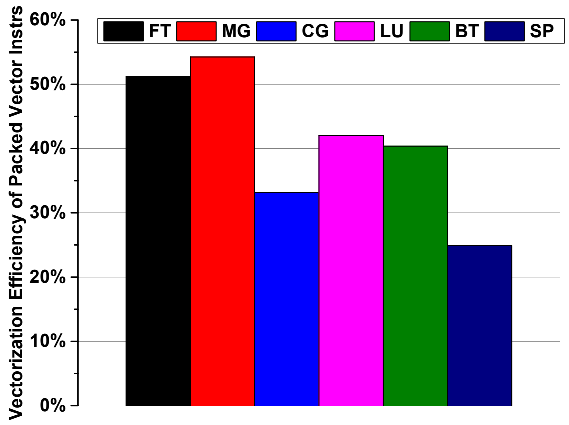

Fig. 8: Average vectorization efficiency on KNL architectures as a percentage of fully utilized AVX512 instructions (assuming a max of 8 DP flops per instruction).

Moreover, the level of sharing for L1 is 4 in KNL, while it is 2 for Haswell. Studying the impact of varying cache hierarchy with the visualization of roofline scaling trajectories could help on developing better designs. Applications that benefit from performance isolation would require architectural features that allow fair resource management of caches. Exploration of such techniques is beyond the scope of this paper.

VI. Related Work

In the distributed memory regime, logP and logGP were commonly used to model the latency, overheads, and bandwidths of the network communication layer that constrain inter-node performance and scalability [10], [11]. Although applicable to understanding MPI-dominate flat MPI execution models, we have chosen to focus on codes dominated by on-

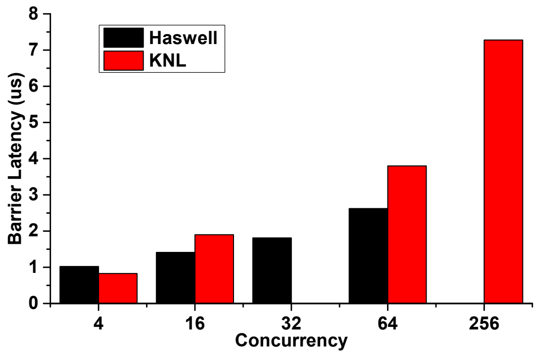

Fig. 9: Barrier latency on Haswell and KNL. KNL generally has higher latency for synchronization for the same level of concurrency, in addition to having higher concurrency.

node computation where inter-process communication is kept to a minimum.

The Roofline methodology has been applied to multiple levels of the cache hierarchy in order to estimate the average memory bandwidth [2], [12]. However, those experiments where conducted at fixed concurrency while our experiments are designed to visualize the interplay between concurrency, application parallelization strategy, and architecture on performance. The Cache-Aware Roofline Model (CARM) is a similar technique used to infer locality by comparing average memory bandwidth to the Roofline ceilings [12]. However, there are a two major differences with our work. First, CARM nominally is run at full concurrency whereas we observe the performance and locality trends as we scale concurrency. Second, when calculating arithmetic intensity, CARM only uses the number of loads and stores presented to the L1 cache where as we use the number of bytes to DRAM after filtering by all cache levels. As a result, CARM cannot observe any contention in the cache hierarchy that gives rise to capacity or conflict misses and superfluous DRAM data movement.

VII. Conclusions

In this paper, we present the Roofline Scaling Trajectory technique for analysis. We also extend the use of the Roofline Model to include user-defined operations. The presented method helps in identifying the performance dependency of an application on temporal architectural structures such as the cache hierarchy. As such, the presented method could serve both performance analysis for applications as well as architectural evaluations of cache hierarchies.

Applying our analysis to the NAS Parallel Benchmarks, we showed the correlation of performance limitations with their scaling trajectories on the Roofline. For instance, applications with changing computational intensity, such as BT, SP, and LU, scaling are likely to face difficulty in achieving roofline limits. They are the most sensitive to the migration from between architectures using centralized and distributed cache hierarchies.

Studying the same application on multiple cache hierarchies could help distinguishing loss of temporal locality due to concurrency induced contention from self-interference between phases of computations for the same thread. This analysis technique is likely to be instrumental as we move to future architectures with a wide variety of memory hierarchies. Although we presented the technique applied to a single level of the memory hierarchy, extending it to other levels of the memory hierarchy, including between memory and I/O, should be straightforward. We believe our technique provides a simple, yet powerful, visualization analysis for temporal behavior during scaling.

Acknowledgment

This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department of Energy Office of Science's Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231.

References

[1] R. Dennard, F. Gaensslen, H. Yu, V. Rideout, E. Bassous, and A. Leblanc, "Design of ion-implanted mosfets with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] S. Williams, A. Watterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the ACM*, April 2009.

[3] S. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2008.

[4] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," *Technical Report NAS-95-010, NASA Ames Research Center*, 1995.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.

[6] LIKWID, "LIKWID," https://github.com/RRZE-HPC/likwid.

[7] Intel Software Development Emulator, "Intel Software Development Emulator," https://software.intel.com/en-us/articles/intel-software-development-emulator.

[8] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights landing: Second-generation intel xeon phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar.-Apr. 2016.

[9] G. Bronevetsky, J. Gyllenhaal, and B. R. de Supinski, "Clomp: Accurately characterizing openmp application overheads," *International Journal of Parallel Programming*, vol. 37, no. 3, pp. 250–265, Jun 2009.

[10] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken, "Logp: A practical model of parallel computation," *Commun. ACM*, vol. 39, no. 11, pp. 78–85, Nov. 1996.

[11] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model&mdash;one step closer towards a realistic model for parallel computation," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, NY, USA: ACM, 1995, pp. 95–105.

[12] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Comput. Archit. Lett.*, vol. 13, no. 1, pp. 21–24, Jan. 2014. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2013.6