Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL

Sabela Ramos
Scalable Parallel Computing Lab
Department of Computer Science
ETH Zürich
E-mail: rsabela@inf.ethz.ch

Torsten Hoefler
Scalable Parallel Computing Lab
Department of Computer Science
ETH Zürich
E-mail: htor@inf.ethz.ch

***Abstract*—Increasingly complex memory systems and on-chip interconnects are developed to mitigate the data movement bottlenecks in manycore processors. One example of such a complex system is the Xeon Phi KNL CPU with three different types of memory, fifteen memory configuration options, and a complex on-chip mesh network connecting up to 72 cores. Users require a detailed understanding of the performance characteristics of the different options to utilize the system efficiently. Unfortunately, peak performance is rarely achievable and achievable performance is hardly documented. We address this with capability models of the memory subsystem, derived by systematic measurements, to guide users to navigate the complex optimization space. As a case study, we provide an extensive model of all memory configuration options for Xeon Phi KNL. We demonstrate how our capability model can be used to automatically derive new close-to-optimal algorithms for various communication functions yielding improvements 5x and 24x over Intel's tuned OpenMP and MPI implementations, respectively. Furthermore, we demonstrate how to use the models to assess how efficiently a bitonic sort application utilizes the memory resources. Interestingly, our capability models predict and explain that the high bandwidth MCDRAM does not improve the bitonic sort performance over DRAM.**

***Keywords*-Cache coherence; memory hierarchy; manycore architectures; performance modeling.**

I. Motivation

Manycore processors, such as Intel's Xeon Phi KNL or Oracle's SPARC, provide growing compute bandwidth with up to 72 cores on a single chip. To keep up with the increasing core-count, coherent manycore processors are configured as multiple NUMA nodes with complex memory hierarchies, including several levels of private and shared caches. While cache coherence hides the complexity of the system from the programmer, it also hides opportunities for performance improvement, making it difficult to exploit the full capabilities that these processors provide. Furthermore, emerging memory technologies such as NVRAM, 3D-stacked, and on-package DRAM complicate the memory subsystem further.

Users who want to reason about system performance are often faced with lacking documentation or peak performance numbers that are hard to achieve in practice. We propose to utilize systematic microbenchmarking to develop detailed capability models that capture the complex performance properties of the memory subsystem. Capability models express the features of an architecture analytically such that they can be used together with application requirement models to reason about performance rigorously [1].

To exemplify the methodology, we develop an extensive memory capability model for the recently released Intel Xeon Phi KNL architecture. The chip provides up to 72 cores grouped in tiles, four threads per core, two levels of cache, six memory DRAM modules using two different technologies, and an on-die mesh interconnect that keeps the full system coherent. Moreover, it provides three memory models and five configuration modes, making a total of fifteen configurations that may affect memory and the cache coherence protocol. The documentation itself states only peak memory bandwidths independent of the configuration.

To demonstrate the effectiveness of our detailed capability models, we show how to use them to (1) automatically design ("model-tune") non-trivial communication algorithms, and (2) assess how efficiently a sorting application utilizes the memory subsystem. We derive close-to-optimal communication algorithms for KNL. For example, Figure 1 shows the model-tuned reduction tree that performs 3 times better than Intel's OpenMP on KNL (cache mode). It is unlikely that this tree would have been found with traditional algorithm design techniques. The model also allows us to estimate how efficiently the memory subsystem is used in a "memory-bound" bitonic sort application. The model enables us to determine ranges of threads and input sizes where this implementation is efficient and where not. Furthermore, the model explains why the higher-bandwidth but higher-latency MCDRAM does not improve performance of this application over DRAM.

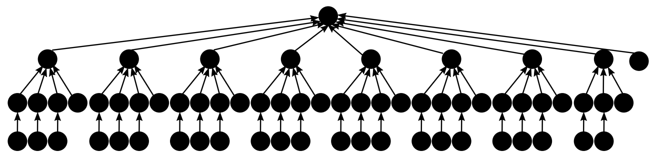


Figure 1: Model-tuned reduction tree for 64 cores on KNL (cache mode).

The main contributions of this paper are:

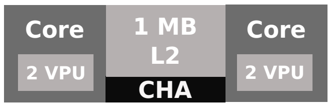
1) We develop a benchmarking methodology to derive

and parametrize capability models of the memory system of manycore CPUs.

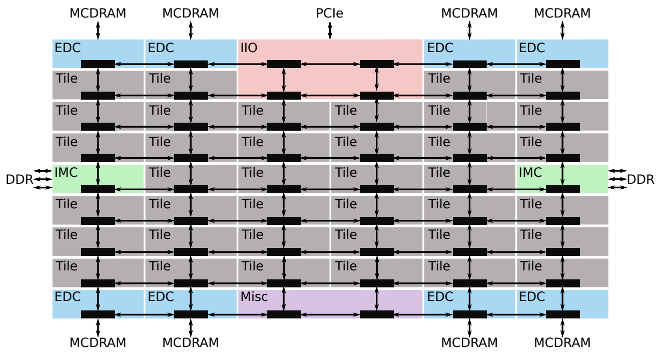
2) We show how capability models can be used to model-tune efficient communication algorithms.
3) We exemplify situations in which the cache-coherence protocol is the main bottleneck and in which it is the memory bandwidth.
4) We present an extensive performance analysis of the multiple modes, configurations, and memory hierarchy of the recently released Intel Xeon Phi KNL.

II. The Intel Knights Landing Architecture

We have used the Intel Xeon Phi KNL to exemplify the usefulness of our methodology. KNL is the new x86-based manycore processor released by Intel [2][3][4]. One of the major changes regarding its predecessor (KNC) is that KNL is shipped not only as a PCIe accelerator, but also as a standalone processor, providing a peak performance of 6 Tflops of single precision and 3 Tflops of double precision per KNL. Moreover, it is binary compatible with Haswell (except for the TSX instructions), supporting Windows Server and Linux.



(a) Dual-core Tile



(b) Mesh Interconnect.

Figure 2: Knights Landing Architecture

A. Cores and tiles

The new Intel Xeon Phi KNL provides up to 72 x86 fully-compliant cores (called Knight cores) with up to four threads (HyperThreads) per core. The Knight core is an out-of-order version of the Silvermont used in the Atom C2000 series, with a twice deeper pipeline, and running at 1.3 GHz. Each core has a private 32 KB L1 data and 32 KB L1 instruction cache. The data cache has 8-way associativity, two 64 B load ports and one store port. Moreover, each core is equipped with two vector processor units that support AVX-512F (AVX3.1) with Intel AVX-512 Conflict Detection Instructions (CDI), Intel AVX-512 Exponential and Reciprocal Instructions (ERI), Intel AVX-512 Prefetch Instructions (PFI), and hardware scatter/gather support.

Cores are arranged in tiles (Figure 2a). Each tile holds two cores and a shared 1 MB L2 cache (private to the tile) with 16-way associativity that provides 1 line read and half-line write per cycle. Together with the cores and the cache, there is a Cache/Home Agent (CHA) that acts as distributed tag directory to keep coherence of the L2 caches across tiles using a MESIF protocol. It is also the connection point of the tile. There are 38 tiles but at least two of them are disabled in all models currently shipping.

B. Tiles and mesh

Tiles are connected into a 2D mesh that provides cache coherence between the L2 caches. Besides the tiles, the mesh incorporates the memory controllers and the I/O connections. The mesh is called a “mesh of rings” in which each row and column is a half ring. At the end of the die, the rings loop around, but not as a torus structure (when a message goes off the ring, it gets injected back in the opposite direction). Each ring stop (e.g., one tile) sees both directions of two discrete rings, one in dimension X and one in dimension Y, with a total of four connections (each ring passes through each stop twice but in opposite directions). Each packet injected to the ring moves first in the Y dimension and then across the X dimension. A stop holds the packet until there is a gap on a ring.

C. Memory

KNL presents an heterogeneous memory hierarchy with ‘near’ (MCDRAM) and ’far’ (DDR4) memory. The ’far memory’ consists of 2400 Mhz DDR4 slots, accessible through two memory controllers, each one with three DDR4 channels, i.e., 6 memory channels total with up to 64 GB per channel (384 GB total) and a peak bandwidth of 90 GB/s. The ‘near’ memory consists of 16 GB (8 chunks of 2 GB each) of integrated Micron Multi Channel DRAM (MCDRAM) based on the Hybrid Memory Cube technologies by Micron with 2.5D stacking and providing 5x bandwidth over DDR (400 - 500 GB/s).

The near memory can be configured in three different modes:

Flat: both memories form a single address space and DDR and MCDRAM appear as separate NUMA nodes.

Cache: the near memory is configured as “fast cache” for the far memory. It is a direct mapped memory based on physical addresses with 64 B lines. Data read from DDR is sent to MCDRAM and the requesting tile simultaneously. It is a “memory-side” cache and acts like a high-bandwidth buffer on the memory side (e.g., memory declared as uncacheable may be allocated in the MCDRAM cache). MCDRAM as cache is inclusive of all modified lines in L2 (write-backs are made directly to MCDRAM). Before a line is evicted from MCDRAM, there is a snoop to check if a modified copy exists in L2. If so, it downgrades it to shared by forcing a write-back and it is not evicted from cache.

Hybrid: the near memory is part cache (4 or 8 GB) and part flat (12 or 8 GB).

D. Cluster modes

Following the Xeon trend to expose the NUMA domains within Xeon sockets [5], KNL provides five levels of NUMA exposure varying how the cache lines are assigned to the tag directories that manage the cache coherence protocol[1]:

A2A (All-to-all): the cache line addresses are uniformly hashed across all the distributed tag directories. This mode is similar to the KNC Xeon Phi cache coherence [6].

Quadrant: the chip is divided into four quadrants, and each cache line is assigned to a directory residing in the same quadrant as the memory from where the cache line is fetched. It takes advantage of certain locality within the mesh while maintaining software transparency.

Hemisphere: the same as quadrant but with two hemispheres.

SNC4 (Sub-NUMA Clustering 4): it is equivalent to the quadrant mode but without transparency. Each quadrant (cluster) is exposed as a separate NUMA domain to the OS, analogous to a 4-socket Intel Xeon. It is similar to the Cluster-On-Die mode present in modern Intel Xeon processors.

SNC2: like SNC4 but exposing two NUMA domains.

Figure 3 shows an example of how the different assignment of lines to the tag directories may impact on an L2 miss in the all-to-all, quadrant, and SNC4 modes.

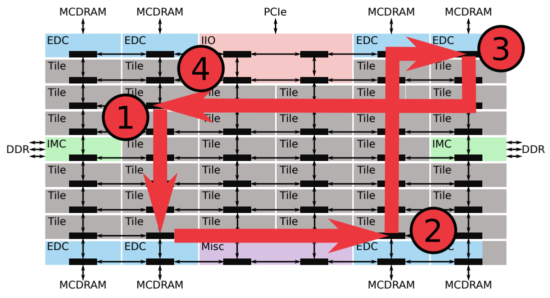
In all-to-all, quadrant, and hemisphere modes, memory addresses are uniformly distributed across the memory channels, although the distribution pattern is internally different due to the different affinity configurations. In flat mode, contiguous ranges are assigned to DDR and MCDRAM respectively, with the MCDRAM range above the DDR range.

In SNC modes, contiguous ranges of memory are assigned to each cluster or NUMA node. In flat mode, the memory region mapped to each cluster is divided in two contiguous portions that are interleaved over the MCDRAM and DDR of the cluster. Note that there are only two memory controllers for DDR and, when using SNC4, the DDR memory address range assigned to a quadrant is interleaved among the three DDR channels of the closest DDR memory controller.

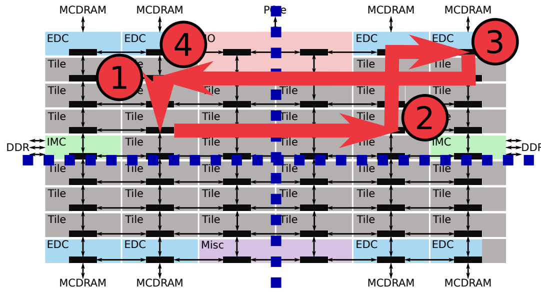
E. I/O and self-hosting

The KNL can be used as a self boot socket or as a PCIe card. The socket version may be equipped with either (1) 36-lanes Gen3 PCIe (root port), or (2) two Omni-Path Fabric Ports (100 Gb/s/dir) and 4-lanes Gen3 PCIe (root port). Instead of Omni-Path, the unit can be equipped with InfiniBand. There is no support for dual-socket KNLs and there is no cache coherence support through the Omni-Path Fabric.

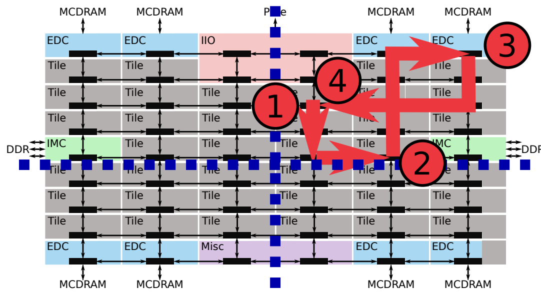
[1]All cluster modes maintain the full system coherent.



(a) All-to-all mode.



(b) Quadrant mode.



(c) SNC4 mode.

Figure 3: Effect of the cluster modes on an L2 read miss. Steps are: (1) L2 miss, (2) request to the distributed directory and directory miss, (3) forward to memory, (4) reply from memory.

III. Memory capabilities Benchmarking

In order to expose the real performance capabilities of the architecture at different levels and configurations, we need to design a benchmark suite to obtain the relevant information about the system. We then can use this information to derive performance models. We focus on cache coherence and memory accesses using a custom benchmark suite and a tool to measure cache-to-cache transfers.

A. Tools and configuration

We used BenchIT [7][8] and the Xeon Phi Benchmarks [9] to measure the cost of cache-to-cache transfers considering cache states and cache line location in coherent processors.

The latter uses ping-pong and one-directional communications (one thread allocates the data and other(s) thread(s) accesses, with no polling). We added ad-hoc benchmarks for synchronization (polling on flags that are cached and not randomized) and memory bandwidth based on STREAM [10] that measure the access bandwidth to random buffers selected from a larger one. The XeonPhi Benchmarks measure the cost of each iteration within each thread. We use the maximum value measured per iteration. The data used in each iteration is randomly selected from a larger buffer. Threads are synchronized with window intervals based on the use of the TSC counter [11]. Before initializing the windows, the TSC skew among cores is calculated. We compile our benchmarks with Intel ICC 16.0.2 and Intel MPI 5.1.3.181. We use an Intel Xeon Phi KNL 7210 with 64 cores at 1.30 GHz, 16 GB MCDRAM and 96 GB DDR4 (2133 MT/s).

B. *Overview of the results*

Tables I and II show a summary of the numbers that characterize the architecture in the different modes. We report medians in all experiments, and, for bandwidth, we report the maximum median achieved across a set of experiments (varying the message size, and the number and schedule of threads). For Copy and Triad, we also report the results obtained with the STREAM benchmark [10].

We discuss these results in detail in the following sections but, in general, we observe that the performance difference between modes appears mainly in terms of achievable memory bandwidth. Moreover, the performance differences when accessing to different quadrants/hemispheres is only visible with some specific latency benchmarks (pointer-chasing) and some configurations of the bandwidth benchmarks. However, in order to reason about performance and modeling, we can use the same performance model and adjust the parameters when necessary.

In comparison with the previous generation of Xeon Phi (KNC), KNL shows much better performance in terms of latency and bandwidth. The main improvement is the single thread performance: KNL does not rely anymore on having more than one thread per core to hide memory access latency. And, although the vector instructions do improve KNL performance, it is not so dependent on them (or on memory alignment) as KNC was.

We also found some limitations that impact our results.

- Due to the yield, some of the tiles are disabled. Hence, it is not possible to know what is the location of a given tile within the mesh (we only know which tile is in which quadrant or hemisphere when using SNC modes).
- Some configuration modes are still experimental, especially the SNC2 mode, which may improve in the future.
- We measure a resolution of 10 nanoseconds in the instruction that reads the TSC counter.

IV. Cache-to-cache transfers

In order to characterize cache-to-cache transfers, we use and extend a methodology [12] that bases on using cache-to-cache benchmarking to construct a simple performance model and optimize communication algorithms. We show how to apply this methodology to Intel Xeon Phi KNL after making an in depth performance analysis, and we use our results to model-tune three communication algorithms.

A. *Benchmarking and modeling*

We characterize the cost of transferring data across the mesh using a set of benchmarks that measure features that we have identified to be key to model communications in cache-coherent systems: cache-to-cache latency, bandwidth, contention (latency when multiple threads are accessing the same data), and congestion (latency when multiple cache-to-cache transfers are happening simultaneously through the mesh). Each feature is a piece of a model that we can use to optimize shared memory algorithms.

1) One cache line transfers: We use BenchIT [8] to measure the cost of cache line transfers between two threads, varying thread location and cache-state. BenchIT provides one number that represents the median[2] of the 5000 averages of 1024 passes of 32 pointer-chasing accesses. The latency results (c.f. Tables I and II) do not show significant differences among the different configuration modes.

We distinguish between accesses to local cache (L1), the cache of the same tile (L2), and the cache of a remote tile (also L2). Although in the accesses to remote tiles the states M (modified) and E (exclusive) perform similarly, within a tile we observe the extra cost of the write-back in the M state. For remote accesses, we observe small differences (5-15%) between the S (shared) and F (forward) state. Moreover, there are between 5-10% differences between the quadrants in the cluster modes (SNC2 and SNC4).

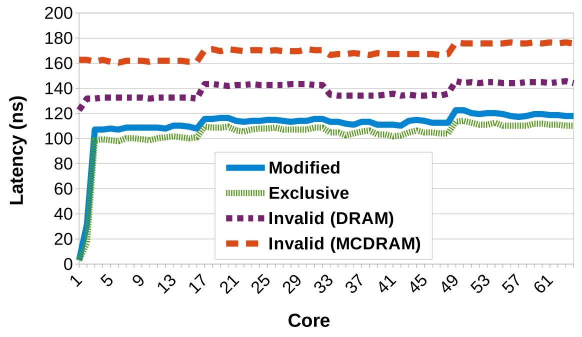


Figure 4: Latency of cache line transfers between core 0 and every other core in SNC4-flat mode for M, E and I states.

Figure 4 shows the latency distribution across cores for states M, E and I in SNC4-flat.

[2]We modified BenchIT to provide the median instead of the minimum.

Table I: Cache-to-cache benchmark results (Section IV). We report medians that are within the 10% of the 95% confidence intervals.

		Software NUMA		Software UMA		
		SNC4	SNC2	QUAD	HEM	A2A
Latency [ns] (Copy/BenchIT)	Local (L1)	3.8	3.8	3.8	3.8	3.8
	Tile (L2)	34 (M)	34 (M)	34 (M)	34 (M)	34 (M)
		17 (E)	18 (E)	18 (E)	18 (E)	18 (E)
		14 (S,F)	14 (S,F)	14 (S,F)	14 (S,F)	14 (S,F)
	Remote	107-122 (M)	111-125 (M)	119 (M)	120 (M)	122 (M)
		98-114 (E)	104-117 (E)	116 (E)	116 (E)	116 (E)
		96-118 (S,F)	104-118 (S,F)	107-117 (S,F)	107-117 (S,F)	109-117 (S,F)
Bandwidth [GB/s] (Read)		2.5	2.5	2.5	2.5	2.5
Bandwidth [GB/s] (Copy)	Tile	6.7 (M)	6.7 (M)	7.5 (M)	7.4 (M)	7.5 (M)
		7.6 (E)	6.7 (E)	9.2 (E)	9.2 (E)	9.2 (E)
	Remote	7.7	6.7	7.5	7.5	7.5
Congestion (P2P pairs)		None				
Contention [ns] (1:N copy)	α	200	200	200	200	200
Linear, $\mathcal{T}_C(N) = \alpha + \beta \cdot N$	β	34	34	34	34	34

Table II: Memory benchmark results (Section V). We report medians and, for bandwidth, we also report the peak as obtained with STREAM. The medians reported are within the 10% of the 95% confidence intervals.

			Software NUMA		Software UMA		
			SNC4	SNC2	QUAD	HEM	A2A
Flat Mode	Latency [ns] (BenchIT)	DRAM	130-140	134-146	140	140	139
		MCDRAM	160-175	160-170	167	167	168
	Bandwidth [GB/s] (Copy NT / STREAM Copy)	DRAM	69 / 77	69 / 77	70 / 77	71 / 77	71 / 77
		MCDRAM	342 / 418	333 / 388	333 / 415	315 / 372	306 / 359
	Bandwidth [GB/s] (Read)	DRAM	71	71	77	77	77
		MCDRAM	243	288	314	314	314
	Bandwidth [GB/s] (Write)	DRAM	33	34	36	36	36
		MCDRAM	147	163	171	165	161
	Bandwidth [GB/s] (Triad NT / STREAM Triad)	DRAM	71 / 82	71 / 82	74 / 82	73 / 82	73 / 82
		MCDRAM	371 / 448	347 / 441	340 / 441	332 / 434	325 / 427
Cache Mode	Latency [ns] (BenchIT)		158-178	161-171	166	168	172
	Bandwidth [GB/s] (Copy NT / STREAM Copy)		150 / 252	130 / 252	175 / 255	134 / 237	132 / 233
	Bandwidth [GB/s] (Read)		87	95	124	128	118
	Bandwidth [GB/s] (Write)		56	56	72	72	68
	Bandwidth [GB/s] (Triad NT / STREAM Triad)		296 / 292	246 / 294	296 / 309	273 / 274	264 / 269

2) Contention: We run a custom benchmark in which one thread running in core 0 owns a one-line buffer, and other N threads access this line simultaneously and copy it into a local buffer. Results show high contention (similarly to KNC) that can be estimated with a $\alpha + \beta \cdot N$ model using linear regression. We populate the model with the results obtained with the different modes and cache line states, as well as different thread schedules: each new thread runs in a different tile vs. each new thread runs in a different core that can be in the same tile. In Table I we show the parameters for the latter schedule. We did not observe significant differences between the cluster and memory modes (although SNC2 is in experimental state and variance is higher than in the other cluster modes).

3) Congestion: We run a benchmark in which pairs of threads located in different pairs of cores communicate simultaneously using a ping-pong pattern. We experimented with multiple thread schedules and did not observe any increase in latency. Note that we do not know the exact location of the tiles in the mesh and we cannot produce layouts that stress specific rows or columns in the mesh.

4) Multi-line: We measure the latency and bandwidth when one thread copies a multi-line message that lies in a remote cache, into a local buffer. As well as the latency and bandwidth of reading a remote message into local registers. Table I reports the maximum median observed with sizes ranging from 64 B (1 cache line) to 256 KB buffers. We use vector instructions because performance is higher when compared with non-vectorized accesses (read bandwidth goes from 1 GB/s to 2.5 GB/s, and copy from 6 GB/s to 9 GB/s, except for SNC2, where it is still 6.7 GB/s). Note that this is single-thread and not aggregate bandwidth, which could be obtained directly since we did not observe congestion in the mesh.

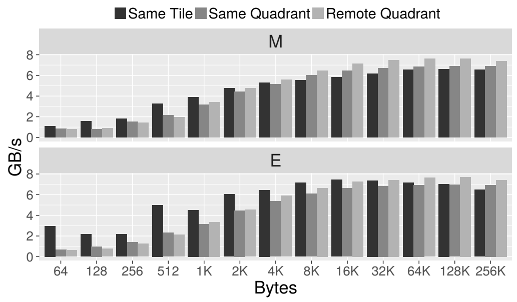


Figure 5: Bandwidth of cache-to-cache copies in SNC4-cache mode.

Figure 5 shows, for SNC4-cache mode, the bandwidth when copying data (in M and E states) into a local buffer in E state. The remote buffer is in another core in the local tile, and in two remote tiles: one in the same quadrant, and another in a remote quadrant. In this copy benchmark, we observe again the extra cost of writing-back when the remote buffer lies in the same tile. Although, in general, local accesses have higher bandwidth than the remote ones when data fits in L1. The only difference between cluster modes appears in the copy benchmark in the SNC modes: the bandwidth of local accesses is lower than in other modes, and there are small differences between the two remote clusters. Also note that this difference only appears when we use vector instructions, and SNC2 mode is still experimental.

In order to obtain a latency model for N cache lines, we fit a linear regression model ($\alpha + \beta N$).

B. Optimization of communication algorithms

The benchmarks described in the previous sections can be utilized to build a capability model for coherent caches of manycore CPUs. We now use the measured parameters to model-tune three communication algorithms: broadcast, reduce, and dissemination barrier.

Because we cannot predict which thread wins and how often a cache line is moved when at least one thread polls the same variable, we model the best and worst case performance for each algorithm using a so-called min-max model [12]. However, we optimize for the best case because the worst rarely happens in practice. We use one thread per core and distinguish between inter- and intra-tile communications. Hence, we apply some optimizations intended for multi-socket or hierarchical machines. On the inter-tile level, the high contention and the mostly homogeneous mesh make the optimization adapt to tradeoffs between parallelism and contention.

1) Considerations for broadcast and reduce: These operations originate or finish in one thread. Using this thread as root, we configure the tiles in a generic tree in which node i has an arbitrary number of children (k_i) [12]. The cost of the inter-tile broadcast tree for n tiles is represented in Equation (1). We express the cost recursively in terms of transmitting the message to the immediate descendants T_{lev}, plus the cost of the most expensive subtree. R_I is the cost of reading one line from memory, R_R from a remote cache, and R_L from local cache. To transfer the data to the descendants, the parent (node 0) copies the data to a shared structure and sets a flag (in the same cache line, $R_I + R_L$), that the children read before copying the data (causing contention, $\mathcal{T}_C(i)$). Finally, the children write a flag (sequentially) that the parent reads to know that the message has been copied ($R_I + k_0 R_R$).

$$\begin{aligned} \underset{k_i}{\text{minimize}}\ T_{bc}(tree) &= T_{lev}(k_0) + \max_{i=1,\dots,k_0}(T_{bc}(subtree_i)) \\ T_{lev}(k_0) &= R_I + R_L + \mathcal{T}_C(k_0) + R_I + k_0 R_R \\ \text{subject to}\ T_{bc}(leaf) = 0 &\quad \textstyle\sum_{i=0}^{n} k_i = n-1,\ k_i \geq 0 \end{aligned} \tag{1}$$

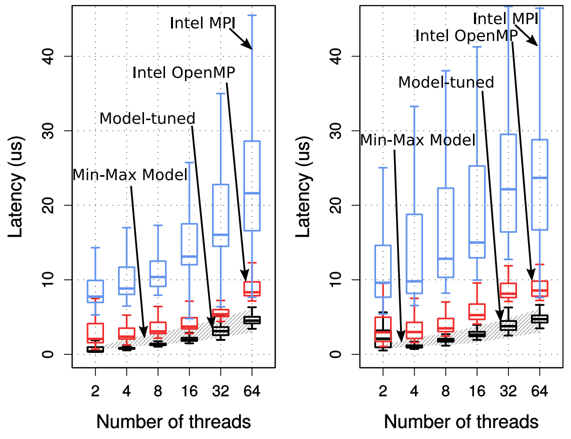
The model for reduce includes some extra buffering to hold the data collected from the descendants and the cost of "reducing" these values. When there is more than one thread per tile, we make a flat tree within the tile. This follows the principle of isolation of expensive (inter-tile) and cheap (intra-tile) polling, so that we limit the performance variation that occurs when one poller fetches a line holding a flag, and the writer has to invalidate it before updating it. The optimization procedure leads to non-trivial trees as shown in Figure 1

2) Considerations for barrier: We use a generic dissemination barrier, based on multiple rounds (r) in which every thread communicates with a given number of threads (m). We use the performance values to configure the number of rounds as shown in Equation (2), where R_I is the cost of reading one line from memory, R_R the cost of reading it from a remote cache, and n is the number of threads involved.

$$\begin{aligned} \underset{r,m}{\text{minimize}} \quad T_{diss,min}(r,m) &= r(R_I + mR_R) \\ \text{subject to} \quad r = \lceil \log_{m+1}(n) \rceil &\quad (m+1)^r \geq n \end{aligned} \tag{2}$$

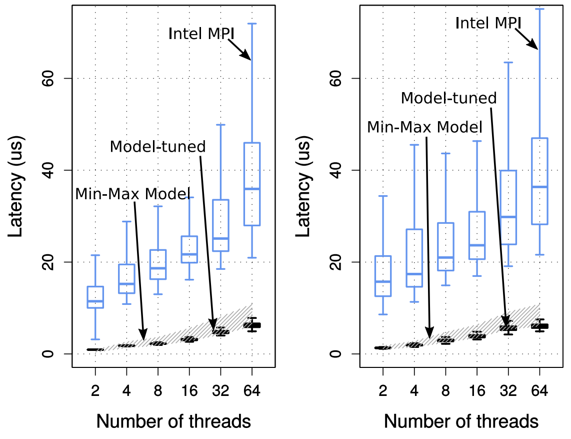
We use a global dissemination so, in each round there is at least one thread communicating with a remote tile, hence we consider R_R as the cost or communicating with a remote tile. According to our model, the reduction in interferences when combining inter-tile dissemination with intra-tile barriers does not compensate for the addition of two extra stages (we need an intra-tile gather, followed by the inter-tile dissemination, and then an intra-tile broadcast).

3) Performance results: We compare our algorithms with Intel MPI and Intel OpenMP. We run 1000 iterations per benchmark and we pin threads to cores with two different schedules: scatter (first one thread per tile, and then per core) and filling tiles (one thread per core). Figures 6, 7, and 8 show the results for SNC4-flat in MCDRAM (the differences between configuration modes are usually below 10%). The results are represented with boxplots and the min-max model is represented by the black shadow. The reduce and broadcast models overestimate the cost when the number of threads is 32 or 64. But it is able to capture the performance trends and variability. Our model-tuned algorithms provide speedups of up to 7x (barrier) and 5x (reduce) over OpenMP, and up to 24x (barrier), 13x (broadcast) and 14x (reduce) over Intel's MPI for KNL. We note that most MPI implementations utilize different address spaces and are thus at a disadvantage. Yet, this is not fundamental because, on manycore, one could simply map all process address spaces into the virtual memory of each process [13].



(a) Filling Tiles. (b) Scatter.

Figure 6: Barrier performance in SNC4-flat (MCDRAM).

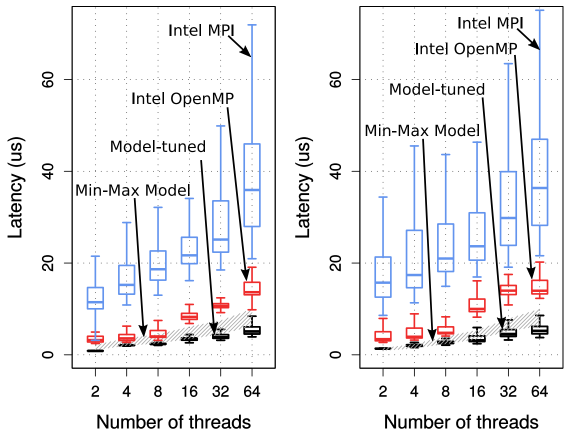


(a) Filling Tiles. (b) Scatter.

Figure 7: Broadcast performance in SNC4-flat (MCDRAM).

V. Memory Bandwidth

The first part of the capability model describes the cache-coherence subsystem. Now, we proceed to design models for hybrid memory architectures that combine DRAM with



(a) Filling Tiles. (b) Scatter.

Figure 8: Reduce performance in SNC4-flat (MCDRAM).

either high bandwidth memory (like MCDRAM) or non-volatile memory (NVRAM). In this scenario, we aim to apply a similar methodology than for cache, selecting a set of representative benchmarks in order to quantify and characterize memory performance. Then we use this characterization to estimate the performance of a sorting algorithm.

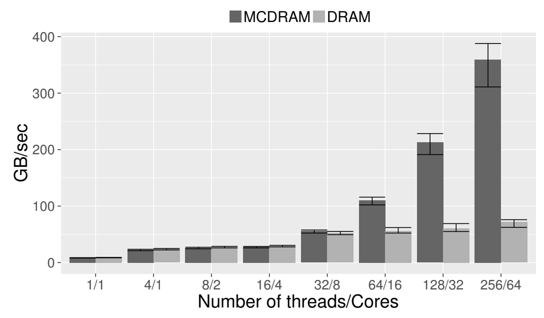
A. Benchmarking

When tackling memory performance, we need to consider large amounts of data that will not fit in cache, hence, we do not need such fine-grained modeling as we did for cache-to-cache communications. We analyze the achievable performance with four types of sequential accesses varying the number of reads and writes performed per iteration: copy ($a[i] = b[i]$), read ($a = b[i]$), write ($b[i] = a$), and triad ($a[i] = b[i] + s \cdot c[i]$). We use vector instructions with non-temporal hints when possible (i.e., when we can avoid cache and write directly to memory) because we need them to come closer to the bandwidth peak. We run the benchmark for 1000 iterations, using random buffers selected from a larger one. Note that we are not using NUMA-aware allocation in the SNC modes (only selecting MCDRAM or DRAM in the flat modes).

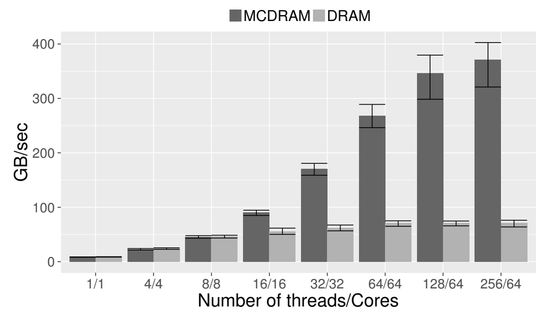
Table II shows the maximum median observed per mode and benchmark in a set of experiments (varying number of threads and schedule). We report the medians because they are the expected performance and we use them to populate our model. For copy and triad, we also report the peak obtained with STREAM. Figure 9 shows the results of our triad benchmark for SNC4-flat.

In these benchmarks we do observe differences between the flat and the cache mode. In every iteration of the benchmark, we select a buffer randomly. In the cache mode, when accessing random buffers, we cannot ensure if it is cached in MCDRAM or not, hence, we obtain much more variability in the results. Besides, the peak performance is lower since the memory has to check first if the requested data is in DRAM or MCDRAM. When using DRAM, copy, read, and triad are around 70-80 GB/s. We found that is it necessary to use both reads and writes simultaneously (e.g., the triad pattern), as well as non-temporal hints (used in triad and copy) to come close to the peak bandwidth of MCDRAM.

We also analyzed two schedules: filling cores with up to four threads (compact) and using 1/2/4 threads per core (scatter). For MCDRAM and compact schedules, we usually need 256 threads to obtain the best results, whereas with scatter we can reach the highest median once we are using all cores (64 or 128 threads). For DRAM, we only need 16 cores to saturate the bandwidth.



(a) Filling Cores.



(b) Filling Tiles.

Figure 9: Memory bandwidth achieved with our triad benchmark in SNC4-flat mode.

B. Application: Sorting

To exemplify the use of the memory data to model an application, we implement a parallel integer merge sort. We parallelize the sorting and implement the merge with a bitonic network of width 16 (for integers) to take advantage of vector instructions [14] (hence, we always fetch full lines). We use ping-pong buffers to limit the amount of necessary memory.

1) Memory access model: In each merge operation we read two lists of $\frac{n}{2}$ lines and produce a list of n lines. When starting the algorithm, we read two lines, we apply the bitonic network, and we write one line. Then, we do $n-1$ repetitions in which we read 1 line, apply the network (or not, if we only have elements left in one list), and write one line. Hence, we have a total of n writes and n reads per merge. When all elements fit in L1 cache, we only fetch data from memory in the first stage, in the rest of them data is in L1. The cost is shown in Equation (3).

$$C_{L1}(n) = [\log_2(n) - 1]2n \cdot \text{cost}_{L1} + 2n \cdot \text{cost}_{mem} \quad (3)$$

We can apply a similar reasoning to obtain the model in Equation (4) when data fits in L2, where n_{L1} is the size of the largest output list that fits in L1. We can process the first elements (n_{L1}) in L1 and then we will be accessing L2.

$$\begin{aligned} C_{L2}(n) = {} & \frac{n}{n_{L1}} C_{L1}(n_{L1}) + \\ & + [\log_2(n) - \log_2(n_{L1})]2n \cdot \text{cost}_{L2} \end{aligned} \quad (4)$$

Finally, we can derive Equation 5 to cover larger sizes that do not fit in L2, where n_{L2} is the size of the largest output list that fits in L2. The available amount of L1 or L2 depends on how many threads are running in the same core or tile.

$$\begin{aligned} C_{mem}(n) = {} & \frac{n}{n_{L2}} C_{L2}(n_{L2}) + \\ & + [\log_2(n) - \log_2(n_{L2})]2n \cdot \text{cost}_{mem} \end{aligned} \quad (5)$$

We added the cost of synchronization among threads between each merge using the cache model: one thread writes a flag once it finishes its merge and, the thread that uses this merged data as an input in the next stage, reads this flag ($R_L + R_R$). Moreover, we added the cost of the bitonic sort taking into account the cost of the AVX-512 instructions and the number of vector instructions per bitonic network [2].

When reading data from cache (cost_{L1} and cost_{L2}) we use the latencies from Table I. However, when accessing memory (cost_{mem}), we can use the latency or the inverse of the bandwidth depending on the layout of the data. When the input lists are ordered, we read one full input list first and the other afterwards. But if the data is random, we interleave reads from each list. Since we cannot predict the layout of the input data, and given that thread interaction is very limited, instead of a min-max model based on thread interaction, we use the latency for the cost_{mem} of the worst case, and the inverse of the bandwidth for the cost_{mem} of the best case. When considering bandwidth, we take into account not only the size of the message that is being accessed, but also the number of threads that are accessing data, as well as their location. Since, as shown in Table II and Figure 9, they both affect the achievable bandwidth.

However, besides the difference in achievable bandwidth between MCDRAM and DRAM, and although our latency model predicts a higher cost for MCDRAM (given its higher access latency), our bandwidth memory model does not predict any performance benefit for this algorithm when using MCDRAM. This is because the memory access pattern of this merge sort does not involve all cores accessing memory except for the first stages, in which the size of the merged arrays is small (when sorting 1 GB with 256 threads, all threads are sorting for output sizes of up to 4 MB). Then, the number of threads is halved until only one thread is working (and the achievable bandwidth for a single-thread is around 8 GB/s in both memories). This highlights the need for performance models in order to quantify the benefits that a given algorithm implementation can obtain from the different memory modules.

2) Full model: Our memory model works well when the memory access cost dominates (the size of the sorted vector is larger than 16 MB). However, for smaller messages, the overhead introduced due to thread management, recursion, and false sharing is higher. In order to assess the efficiency of our implementation and analyze when the cost is dominated by memory access bandwidth, we developed an overhead model by applying linear regression to the cost of sorting 1 KB messages with multiple number of threads, after subtracting the cost predicted by the memory model. Then, we use this overhead for all the message sizes, combined with the memory model.

3) Results: Figure 10 presents the results (for SNC4-flat and MCDRAM) of running our parallel merge sort with a random input, as well as the memory models (using latency and using the inverse of the bandwidth), and the full model that combines one of the memory models and the overhead model. We mark with a vertical black line when the overhead is over 10% of the memory model, meaning that we are no longer bounded by the memory bandwidth *achievable by this algorithm*, but, instead, we are introducing extra overhead and not using our resources efficiently.

When the data size is small (1 KB), the two memory models are very similar and the cost is dominated by the overhead when using more than two threads. For intermediate sizes (e.g., 4 MB), for less than eight threads the cost is dominated by memory accesses. After that, the efficiency of our algorithm decreases, as the overhead is larger. For large sizes, the cost is bounded by memory accesses, and our implementation makes an efficient use of the resources.

As predicted by the model, the difference between MC-DRAM and DRAM is negligible, despite the higher bandwidth of MCDRAM, due to synchronization and latency overheads.

VI. Related Work

The detailed study of the hardware characteristics of complex manycore architectures is key to understand their advantages, in CPU systems [15], and accelerators [6] [16] [17].

Nevertheless, we need to translate the hardware capabilities into usable models and guidelines that the programmers can use. Petrovic et al. [18] discuss how to implement a broadcast basing on the architecture of Intel SCC, with no coherence. In previous work, we tackled the development of an analytical performance model for cache-coherent systems [12]. In this work, we extend it and apply it to a mesh-based manycore as well as extend the analysis to hybrid memories and multiple configuration modes.

The most common model used to analyze the memory bandwidth needs of an application in a given system is the roofline model. Doerfler et al. [19] apply this model to the Xeon Phi KNL, and it has been used to compare different architectures [20]. However, it does not provide a framework to optimize algorithms. We make an in depth analysis of the memory capabilities and show how we can use it to predict the performance and the efficiency of a merge sort algorithm.

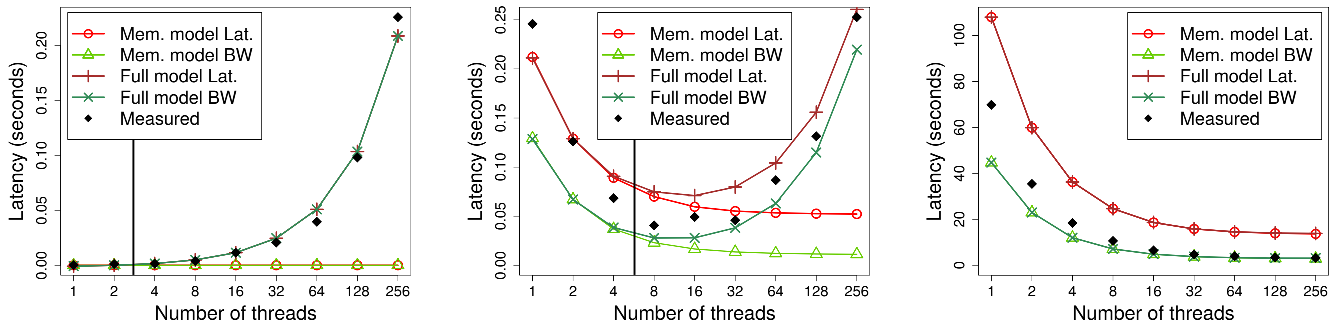
VII. Discussion and Conclusions

Heterogeneous memory systems make it difficult to reason about their performance. Moreover, documentation usually only reports peak performance which does not provide a realistic model for the capabilities of a system. Achieving this performance in real applications might be very difficult (or even impossible). Furthermore, there are multiple variables whose impact is not clear unless it is measured, like use of vectorization, number and type of memory accesses (reads and writes), thread scheduling, memory pinning, or NUMA-aware allocation. For example, in KNL, we show that, even though we can touch peak memory bandwidth by carefully tuning a memory benchmark, on average and with randomized addresses, the median of the results is far from peak. Moreover, depending on the application characteristics, we may need to measure different capabilities, often with different granularities (e.g., cache vs. memory).

With the Intel Xeon Phi KNL as a case-study, we use benchmarking to show how the cache-coherence protocol is designed for homogeneity, trading programmability for performance. Even the differences between the multiple mesh configuration modes are not that relevant, especially on a fine-grained level. And, although we cannot investigate these differences further because tile location in the mesh is unknown, the impact of this location does not seem to be relevant in the modeling of communication algorithms.

Regarding the memory modes, although the cache mode eases memory allocation, the access latency is higher, and the amount of data that does not fit in the MCDRAM cache (or is not reused) limits the achievable memory bandwidth. However, when using a flat mode, we need performance models in order to decide which data has to be allocated in which memory. And, in both cases (flat and cache), we show how a performance model can guide us in assessing how efficient is our application in terms of resource usage.

To sum up, we derived systematic benchmarking methodologies to select relevant parameters for capability models of memory subsystems. These models enable a flurry of applications to reason rigorously performance. We showed two use-cases to design algorithms and to assess how effective memory bandwidth is usable by a sorting application. We believe that a structured model-driven approach to performance engineering of memory bound applications can lead to further interesting insights into the relation between applications and complex memory systems.



(a) Sorting 1 KB of integers. (b) Sorting 4 MB of integers. (c) Sorting 1 GB of integers.

Figure 10: Performance of the sorting algorithm when using a compact schedule (filling cores) in SNC4-flat mode (MCDRAM).

ACKNOWLEDGMENTS

The authors would like to thank Jeff Hammond, James H. Cownie, and Hans Pabst from Intel for their support, as well as Hussein Harake and the CSCS for the access to their Xeon Phi KNL system. Sabela Ramos is supported by an ETH Posdoctoral Fellowship.

REFERENCES

[1] T. Hoefler, W. Gropp, M. Snir, and W. Kramer, "Performance Modeling for Systematic Performance Tuning," in *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'11), SotP Session*, Seattle, WA, USA, 2011, pp. 6:1–6:12.

[2] J. Reinders, J. Jeffers, and A. Sodani, *Intel Xeon Phi Processor. High Performance Programming. Knights Landing Edition*. Boston, MA, USA: Morgan Kaufmann Publishers Inc., 2016.

[3] E. Gardner, "What public disclosures has Intel made about Knights Landing?" https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing, 2016.

[4] A. Sodani, "Knights Landing: 2nd Generation Intel Xeon Phi Processor," in *Hot Chips: A Symposium on High Performance Chips (HC27)*, Cupertino, CA, USA, 2015.

[5] D. Hackenberg, D. Molka, and W. E. Nagel, "Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture," in *Proc. 44th Intl. Conf. on Parallel Processing (ICPP'15)*, Beijing, China, 2015, pp. 739–748.

[6] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi," in *Proc. 22nd Intl. Symposium on High-performance Parallel and Distributed Computing (HPDC'13)*, New York, NY, USA, 2013, pp. 97–108.

[7] D. Molka, D. Hackenberg, R. Schoene, and M. S. Mueller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, Raleigh, NC, USA, 2009, pp. 261–270.

[8] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems," in *Proc. 42nd IEEE/ACM Intl. Symp. on Microarchitecture (MICRO'42)*, New York, NY, USA, 2009, pp. 413–422.

[9] S. Ramos and T. Hoefler, "Benchmark Suite for Modeling Intel Xeon Phi," http://gac.des.udc.es /~sramos/xeon_phi_bench/xeon_phi_bench.html.

[10] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, 1995.

[11] G. Paoloni, "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures," Intel White Paper 324264-001, 2010.

[12] S. Ramos and T. Hoefler, "Cache Line Aware Algorithm Design for Cache-Coherent Architectures," *IEEE Transactions on Parallel & Distributed Systems (TPDS)*, vol. 27, no. 10, pp. 2824–2837, 2016.

[13] S. Li, T. Hoefler, and M. Snir, "NUMA-Aware Shared Memory Collective Communication for MPI," in *Proc. 22nd Intl. Symposium on High-performance Parallel and Distributed Computing (HPDC'13)*, 2013, pp. 85–96.

[14] N. Satish, C. Kim *et al.*, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," in *Proc. ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'10)*, Indianapolis, IN, USA, 2010, pp. 351–362.

[15] S. Saini, J. Chang, and H. Jin, "Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications," in *Proc. 4th Intl. WS. on Perf. Modeling, Bench. and Sim. of HPC Systems (PMBS'13)*, Denver, CO, USA, 2013.

[16] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. IEEE Intl. Symposium on Performance Analysis of Systems & Software (ISPASS'10)*, White Plains, NY, USA, 2010, pp. 235–246.

[17] X. Mei, K. Zhao, C. Liu, and X. Chu, "Benchmarking the Memory Hierarchy of Modern GPUs," in *Proc. 11th IFIP WG 10.3 Intl. Conf. on Network and parallel Computing (NPC'14)*, Ilan, Taiwan, 2014, pp. 144–156.

[18] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper, "High-performance RMA-based Broadcast on the Intel SCC," in *Proc. 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12)*, Pittsburgh, PA, USA, 2012, pp. 121–130.

[19] D. Doerfler, J. Deslippe *et al.*, "Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor," in *IXPUG Workshop: Application Performance on Intel Xeon Phi – Being Prepared for KNL and Beyond*, Frankfurt, Germany, 2016.

[20] S. Muralidharan, K. OBrien, and C. Lalanne, "A Semi-Automated Tool Flow for Roofline Anaylsis of OpenCL Kernels on Accelerators," in *Proc. 1st Intl. Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'15)*, Austin, TX, USA, 2015.