Optimizing for KNL Usage Modes When Data Doesn't Fit in MCDRAM

Neil Butcher
Dept. of Computer Science and Engineering, Univ. of Notre Dame
South Bend, Indiana
ndbutcher@nd.edu

Stephen L. Olivier
Center for Computing Research, Sandia National Laboratories
Albuquerque, New Mexico
slolivi@sandia.gov

Jonathan Berry
Center for Computing Research, Sandia National Laboratories
Albuquerque, New Mexico
jberry@sandia.gov

Simon D. Hammond
Center for Computing Research, Sandia National Laboratories
Albuquerque, New Mexico
sdhammo@sandia.gov

Peter M. Kogge
Dept. of Computer Science and Engineering, Univ. of Notre Dame
South Bend, Indiana
kogge@nd.edu

ABSTRACT

Technologies such as Multi-Channel DRAM (MCDRAM) or High Bandwidth Memory (HBM) provide significantly more bandwidth than conventional memory. This trend has raised questions about how applications should manage data transfers between levels. This paper focuses on evaluating different usage modes of the MCDRAM in Intel Knights Landing (KNL) manycore processors. We evaluate these usage modes with a sorting kernel and a sorting-based streaming benchmark. We develop a performance model for the benchmark and use experimental evidence to demonstrate the correctness of the model. The model projects near-optimal numbers of copy threads for memory bandwidth bound computations. We demonstrate on KNL up to a 1.9X speedup for sort when the problem does not fit in MCDRAM over an OpenMP GNU sort that does not use MCDRAM.

CCS CONCEPTS

• **Theory of computation → Parallel algorithms**; • **Computer systems organization → Multicore architectures**;

KEYWORDS

Multilevel Memory, Sorting, Intel Knights Landing

ACM Reference Format:
Neil Butcher, Stephen L. Olivier, Jonathan Berry, Simon D. Hammond, and Peter M. Kogge. 2018. Optimizing for KNL Usage Modes When Data Doesn't Fit in MCDRAM. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225116

1 INTRODUCTION

Data movement and retrieval are key bottlenecks in high-performance computing. Moving data is becoming relatively more costly than arithmetic and logic operations, in terms of performance and energy efficiency, as machines approach exascale. This divergence is due to the slow rate of improvement in memory bandwidth compared to processing power of manycore compute nodes, as well as almost no change in memory access latencies. More and more cores rely on the similar limited number of DRAM channels.

One architectural solution is to implement *multilevel memory (MLM)*. Emerging architectures are supplementing traditional DDR memory (typically packaged in separate DIMM modules) with high-bandwidth, "on-package" memory, logically positioned between the main memory and the processor core cache hierarchy, as shown in Figure 1. The Intel Knights Landing (KNL) processor [24] takes this approach, incorporating 16GB of high-bandwidth Multi-Channel DRAM (MCDRAM). MCDRAM is a 3D stack of memory where multiple wide memory ports take the place of a single high speed off-chip interface, offering higher bandwidth without a significant change in latency. A standardized variant of MCDRAM called High Bandwidth Memory (HBM) [3] is now in use with a variety of processing chip types, from FPGAs to GPUs, *e.g.*, the newest NVIDIA Volta GPU architecture. We use the term *multilevel memory* to refer to the general category of memory technologies that provide high-bandwidth capabilities.

1.1 MCDRAM Memory Modes

The Intel Knights Landing (KNL) machine used in this paper utilizes the Intel 7250 Xeon Phi processor with 68 four-way threaded cores (for a total of up to 272 threads) in a single cache coherent domain connected to 6 DDR4 channels and 8 MCDRAM stacks. The MCDRAM is configurable through the BIOS to allow three modes of operation: cache, flat, and hybrid.

In *hardware cache mode* the MCDRAM acts as a direct mapped cache with 64B lines (consistent with the cache line size of the L1 and L2 caches). The main advantage of hardware cache mode is the potential for performance gains with no direct effort on the part of the application developer. The main disadvantage is that MCDRAM, which offers no better latency than DDR, is not a natural choice for cache. To use the high bandwidth effectively, algorithms must stream through data rather than making random accesses. Furthermore, thrashing is a common problem with direct-mapped caches. Another potential downside of hardware cache mode is

ICPP 2018, August 13–16, 2018, Eugene, OR, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225116

that some portion of the memory is reserved to hold the tags of the cache, reducing the effective usable capacity.

In *flat mode* the MCDRAM serves as an extension to the addressable memory (*i.e.*, a "scratchpad"), and allocation into MCDRAM can be handled by special API calls such as 'hbw_malloc()' function[1] developed in the memkind library [8]. OpenMP support for complex memory such as MCDRAM is anticipated for the forthcoming 5.0 specification [19].

In *hybrid mode* some portion of the MCDRAM is used as an extension to main memory (requiring special allocations as in flat mode), and the remaining portion acts as a direct mapped cache. This configuration allows the programmer to explicitly allocate data structures that are highly reused to be stored explicitly into the flat memory space while allowing non-uniform access patterns that still exhibit some degree of locality to benefit from the cache portion. Since the total amount of MCDRAM is fixed, this mode makes less space available for caching than hardware cache mode and less space available for scratchpad use than flat mode.

Is MCDRAM just a very large cache, its own unique memory space or a combination of cache and addressable memory? At least for the last two options, explicit data movement may be required to transfer the data between main memory and the MCDRAM. Such an approach may not be feasible for large application suites with millions of lines of code. However, we recognize a fourth usage mode that leverages the KNL's cache mechanism while still adapting algorithms to MLM. We call this mode *implicit cache mode* because while the code is modified, no explicit data movement is required.

The ideal usage model for an application developer would be a hardware cache mode that exploits MCDRAM optimally. Although we confirm impressive performance in hardware cache mode in this paper, three factors make us skeptical that it is a general solution: (1) the lessons of the long-studied disciplines of external-memory and cache-friendly algorithms; (2) the overheads of treating MCDRAM as a cache, and, (3), the fact that unlike traditional CPU caches, MCDRAM has latency similar to that of DRAM.

[1]http://memkind.github.io/memkind/

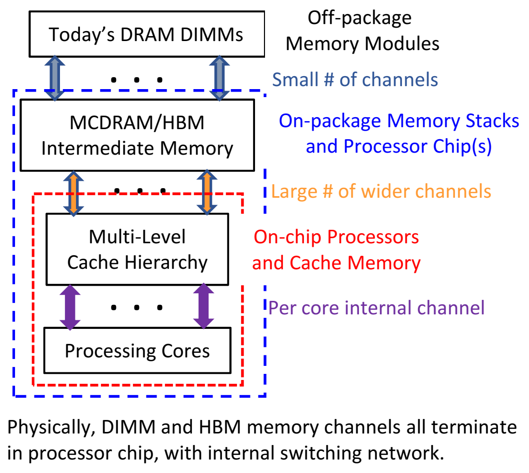

Today's DRAM DIMMs
Off-package Memory Modules
Small # of channels
MCDRAM/HBM Intermediate Memory
On-package Memory Stacks and Processor Chip(s)
Large # of wider channels
Multi-Level Cache Hierarchy
On-chip Processors and Cache Memory
Per core internal channel
Processing Cores
Physically, DIMM and HBM memory channels all terminate in processor chip, with internal switching network.

Figure 1: Logical Multi-Level Memory Architecture.

These factors suggest that targeted rewrites of kernels may be necessary to exploit MCDRAM optimally in flat mode. We demonstrate through a study of sorting that careful algorithmic work can optimize performance in all memory modes. These redesigned algorithms provide performance improvements well beyond that of conventional algorithms.

1.2 Contributions

The following points summarize the contributions of this paper.

- We corroborate the simulation results of [4] with real KNL runs.
- We give a muiltilevel memory sorting algorithm that leverages MCDRAM to achieve speedups over the best current algorithms run in hardware cache mode, and show that these gains can be preserved without explicitly copying between DDR and MCDRAM.
- We propose a model that predicts near-optimal numbers of copy threads to use with a simple MLM benchmark algorithm.

The organization of the rest of this paper is as follows. Section 2 discusses related work. Section 3 presents an approach to adapt kernels to multilevel memory. Sections 4 and 5 discuss a new sorting algorithm we call MLM-sort and streaming merge workloads, respectively. Section 6 concludes the paper.

2 RELATED WORK

The extensive literature relevant to multiple levels of memory in general, and sorting in particular, fall into several categories:

- *cache-friendly* algorithms that deal with two levels of memory: processor cache and main memory, where only main memory is explicitly addressable. This includes both *cache-oblivious* and *cache-aware* algorithms.
- *external memory* algorithms that deal with two different kinds of storage where both are addressable but with radically different access protocols: main memory and disk.
- *multilevel memory* algorithms that deal with a level of on-package memory and a level of main memory, e.g., DDR, where both are in some way in the same "address space."
- *performance analyses* of KNL that characterize profitability of MCDRAM for applications.

In this paper, we make extensive use of the multithreaded multiway merge and mergesort routines in the GNU parallel library [22, 23]. In our experience, these represent the current state of the art in multithreaded sorting.

2.1 Cache-Friendly Algorithms

A cache-aware algorithm arranges its memory references to increase cache hits. The algorithms described in later sections are multilevel memory versions of simple cache-aware algorithms. The "cache" in this case is MCDRAM on a KNL machine.

Previous work suggests that *cache-oblivious* versions of our algorithms might work just as well without being parameterized by the size of MCDRAM. The discipline of cache-oblivious algorithm design was introduced in the classical work of Frigo, et al. [12], which included a cache-oblivious sorting algorithm (“funnelsort”). The main idea is to let the algorithm partition data recursively. A carefully-engineered version of “Lazy Funnelsort” was eventually shown to outperform the best quicksort implementations [6, 7]. This was a complicated effort, but its success suggests that cache-oblivious versions of our simple algorithms might eventually perform as well without requiring tuning per machine. Finally, we note that there is work on *cache-adaptive* algorithms that tolerate changes to system resources during the run of the algorithm [5]. Funnelsort is shown to work optimally in their adaptive model. This would be useful in a future in which high-performance computing jobs must deal with fluctuating resource allocations.

2.2 External Memory Algorithms

External memory, or “out-of-core” algorithms were developed to perform computation that could not fit on the DRAM of the system, when the data is primarily stored in disk. These algorithms essentially use a chunking technique to perform the data movement. The theoretical model for external algorithm design is the “Disk Access Model” (DAM), first introduced by Aggarwal [1].

Out-of-core algorithms are designed to accommodate much larger latency on the slow storage. They sometimes exploit multiple I/O channels as we do. A major difference is that the data movement in our work is relatively fast, since latency is more nearly equivalent for DDR and MCDRAM (versus DDR and disk). Copying from disk to DDR tends to be much slower and time consuming. In mergesort-based out-of-core sorting algorithms the multiway-merge at the end is a slow and complex operation. Out-of-core sorting algorithms are designed to handle larger datasets that can only be stored on disk, whereas our in-memory sort can only sort datasets that fit into the DDR memory.[20]

2.3 Multilevel Memory Algorithms

Bender, *et al.* discussed the problem of algorithm design for general multi-level memory comprising a main memory and a high bandwidth ‘near memory’ [4]. They introduced a theoretical model similar to the DAM model, designed and analyzed a sorting algorithm using that model, and used simulation to predict the performance of a simplified version of that algorithm on KNL before the hardware was available. They predicted that using a secondary, higher bandwidth level of memory would improve sorting performance by 30%, and reduce the DDR memory traffic by 2.5x. Our results in Section 4.1 largely corroborate those predictions. They also expressed a simple technique suggested by Marc Snir for determining whether a computation is memory bandwidth-bound or not. For large core counts, they predicted that sorting can indeed be memory bandwidth-bound, and hence stands to gain from MCDRAM. [4]

2.4 KNL and MCDRAM Performance Analyses

MCDRAM and HBM represent only a subset of the MLM on modern machines. The Graphics Processing Units (GPUs) now widely used in high-performance computing usually require a fast “device memory” that supplements the main memory of the host processor. Data must be transferred to and from the host memory either implicitly (as in the case of NVIDIA’s unified memory) or explicitly via user-written functions. The MCDRAM found on KNL can support both, but it presents a serious challenge to application developers: how best to utilize the extra memory level. We focus on the fundamental choice of which developer usage model to use. This decision informs whether and how to rewrite kernels of application codes.

Previous work has also considered the impact of MCDRAM on the performance of applications on KNL. The work of Li et al. [16] examines the performance of a wide variety of scientific kernels on KNL using MCDRAM in flat, hardware cache, and hybrid modes. However, their use of the flat mode does not entail chunking data sets larger than the MCDRAM capacity. Instead, they use the setting exposed through the ‘numactl’ tool that simply allocates data in DDR memory once the MCDRAM is full. Doerfler et al. [11] apply the roofline performance model to flat and hardware cache modes. Denoyelle et al. [10] derive a general “cache-aware” roofline model and apply it to MCDRAM on KNL. The Interactive Code Adaptation Tool (ICAT) [2] uses profiling to recommend the MCDRAM mode to use when porting an application to KNL. Ramos and Hoefler [21] characterize KNL memory performance with a focus on distinguishing among common application bottlenecks: memory bandwidth, latency, and cache coherence traffic. Their model predicts that bitonic sort should not benefit from MCDRAM. This work operates off the conjecture that the problem size is small enough so that all data is stored in MCDRAM.

3 REDESIGNING KERNELS FOR MLM

“Chunking” is a technique to take advantage of multilevel memory systems. The concept behind chunking is to migrate a portion of the data, a “chunk,” from the “far memory” into the “near” memory. The necessary computation is then performed on this chunk of data. Once the computation is complete, the current chunk is moved back from the near memory to the far memory and the next chunk moved into near memory. The program repeats these steps for each chunk. Once all the chunks have been operated upon there may still be a need for some merge step that aggregates the result. Ideally, moving the data closer to the processor should considerably speed up the computation even with the additional cost of explicitly copying data in and out.

In KNL flat mode, chunking has the added overhead of performing the data movement between MCDRAM and DDR. The common solution to this problem is referred to as “buffering”. Buffering allows the movement of chunks in and out of near memory to be pipelined. This is done by overlapping the memory transfers with the computation. The lack of user-programmable direct memory access (DMA) facilities on a KNL processor requires that one or more of the application’s threads perform the transfers. The implementation of buffering for KNL thus typically requires allocating three separate thread pools, a large pool for performing the computation, then another pool to perform the “copy-in” and finally, a third pool to perform the “copy-out.” This arrangement has the disadvantage of higher contention for resources. Using more threads to perform the copying operations results in fewer available threads to perform the compute. Using a buffered technique also has the disadvantage that 2/3 of the MCDRAM will be used by the copy threads. The copy threads use both MCDRAM and DDR bandwidth, as well as on-die resources such as network-on-chip bandwidth, so, the use of too many copy threads may lead to contention with compute

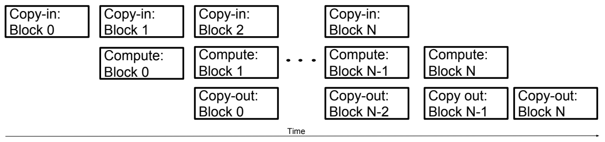


Figure 2: Chunking and buffering.

threads and result in lower aggregate performance. Another problem that arises from using buffering is there are three active buffers in MCDRAM at any given time (one for each of the thread pools). This limits the maximum chunk size which can have a significant impact on performance.

3.1 MLM Optimization in MCDRAM Modes

Kernels adapted for MLM can target any of the KNL MCDRAM usage modes, but the effort required and the impact on performance may vary. Flat mode offers the most potential for optimization since the user has complete control of the MCDRAM. However, it also requires the most effort since explicit data movement is required. Hardware cache mode still may be preferable for developers of large applications. Adapting a small portion of code in a large application to use chunking may not warrant a transition to flat mode. This is especially true if the rest of the application could utilize the hardware cache well. Hybrid mode attempts to serve as a compromise between flat mode and hardware cache mode, but optimal use still requires explicit data movement. We outline the use of a chunking approach on each of these modes, in addition to the implicit cache mode that we propose as as an alternative.

Fig. 2 illustrates how MCDRAM is often used for a pipelined chunking algorithm in flat mode. The operations are performed by three separate thread pools, copy-in, copy-out, and compute. In the first step a single chunk of data is copied into the MCDRAM. In the second step a second chunk of data is copied in while computation occurs on the first chunk. The third and later steps then also include a copy-out of the data computed in the prior step. Each subsequent step repeats all three stages concurrently until the end of the data set is encountered. At the penultimate step, no copy-in is needed. In the final step, neither copy-in nor computation occur. If there is no interaction between phases, the time for each step is determined by the longest of the components. Ideally the copy time would overlap completely with the compute time. The data movement in terms of MCDRAM is shown in in Figure 3.

The hybrid mode performs the chunking in the same manner as flat mode. The hybrid mode is more limited in the maximum chunk size because of the smaller addressable MCDRAM memory. MCDRAM cache is often polluted by the copy-in and copy-out data, making it less effective. The data movement of the hybrid chunked algorithm is shown in Fig. 4.

We refer to the use of chunking while the MCDRAM is configured in cache mode as implicit cache mode. In implicit cache mode all available threads are dedicated to performing the compute. A single chunk is computed upon at a time. Initial accesses cause the MCDRAM caching mechanism to bring data into the MCDRAM implicitly. The downside of relying on the caching mechanism is that the bandwidth suffers at the start of each chunk. Data has to be transfered into the MCDRAM cache by the system on initial accesses, i.e., cold misses, which are expensive to service. The data movement using implicit cache mode is shown in Figure 5.

Bender, *et al.* developed preliminary guidance for application developers who must decide whether to rewrite certain computational

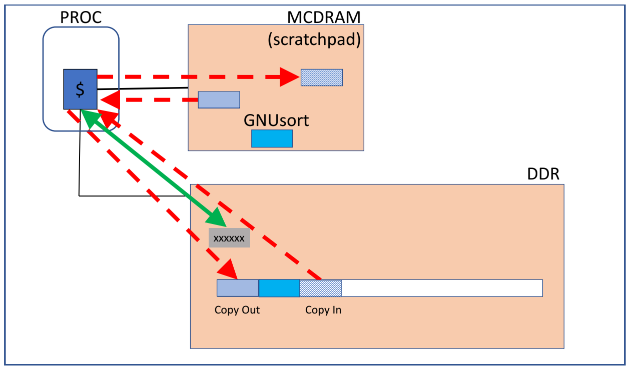


Figure 3: Chunked flat using sort as an example

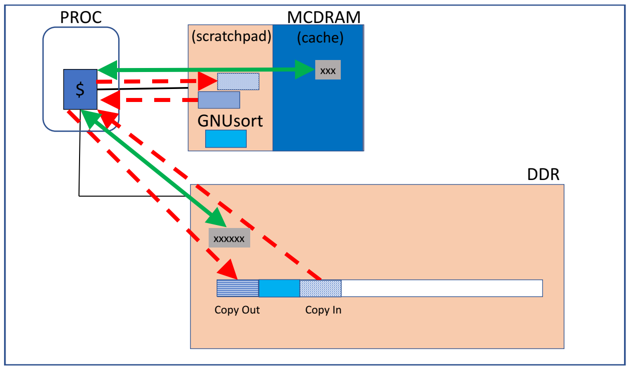


Figure 4: Chunked hybrid mode using sort as an example

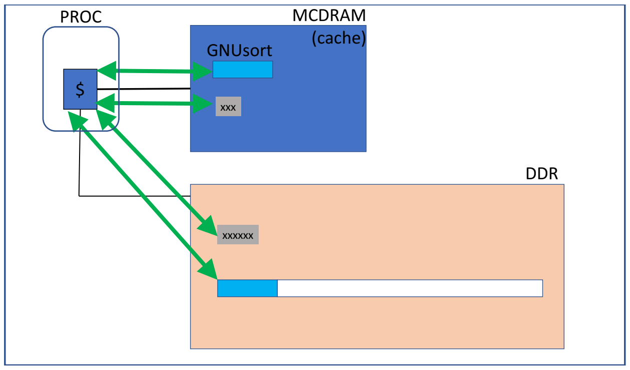


Figure 5: Implicit chunking using sort as an example. The red arrows denote user-controlled scratchpad transfers; the green arrows denote system-controlled transfers. In “implicit mode” the algorithm designed for flat mode is run in hardware cache mode.

kernels in order to optimize performance on systems with such high-bandwidth intermediate level memory [4]. Using a divide-and-conquer approach in which the chunks effectively fit into the high-bandwidth memory, the study showed that significant speedup is possible in theory and in simulation. This paper corroborates this result and demonstrates for a real system that such transfers can improve performance.

3.2 Model for Buffering MLM Algorithms

Choosing the ideal number of copy threads is typically not obvious without a great deal of experimentation, particularly in cases where execution behavior is affected by problem size or input data complexity. In such cases, choosing the number of copy threads is often critical to optimizing performance but would require significant user benchmarking.

To better understand how to choose the ideal number of copy threads we derive a model based on the following simplifying assumptions: the MCDRAM is configured in flat mode using up to three thread pools, one pool to perform the computation, one to copy data into MCDRAM, and one to copy data back from MCDRAM. The copy-in and copy-out pools are equal in size and have equivalent workloads. Compute threads use strictly MCDRAM, while the copy threads use both DDR and MCDRAM. Thus, our model must consider the bandwidth limitations of DDR and MCDRAM and the bandwidth each thread pool can supply.

Starting from a high level, consider that the total time, T_{total} for the entire execution is limited by the copy time, T_{copy}, and the compute time, T_{comp}. Hence the following equation.

$$T_{\text{total}} = \text{Max}(T_{\text{copy}}, T_{\text{comp}}) \tag{1}$$

Let T_{copy} be the data set size in GB. The copy time, T_{copy}, is the time to transfer B_{copy} twice (into MCDRAM and back out to DDR). With p_{in} threads used to copy data in and p_{out} threads used to copy data out, the following equation gives the copy time.

$$T_{\text{copy}} = \frac{2 * B_{\text{copy}}}{(p_{\text{in}} + p_{\text{out}}) * C_{\text{copy}}} \tag{2}$$

The coefficient C_{copy} represents the copy rate in GB/s for a single thread. The value of this coefficient depends on whether the maximum DDR bandwidth, DDR_{max}, has been reached. For a small number of copy threads, each additional thread contributes to an increased aggregate copy rate. Let S_{copy} be that maximal per-thread contribution. If the number of threads is large enough to saturate the DDR bandwidth, each additional thread merely reduces the per thread share of the available bandwidth rather than increasing the aggregate copy rate. The following equation expresses this conditional function.

$$C_{\text{copy}} = \begin{cases} S_{\text{copy}}, & \text{if } (p_{\text{in}} + p_{\text{out}}) * S_{\text{copy}} \le \text{DDR}_{\text{max}} \\ \\ \frac{\text{DDR}_{\text{max}}}{p_{\text{in}} + p_{\text{out}}}, & \text{otherwise} \end{cases} \tag{3}$$

The compute time reflects a streaming algorithm requiring the data set to be both read and written over some number of passes. If p_{comp} threads are used, each computing at a per-thread rate of C_{comp}, the following equation gives the compute time.

$$T_{\text{comp}} = \frac{2 * B_{\text{copy}} * \text{Passes}}{p_{\text{comp}} * C_{\text{comp}}} \tag{4}$$

The remaining task is to derive the coefficient C_{comp}. Again, bandwidth limitations apply. Each additional compute thread adds to the aggregate compute rate, until the MCDRAM bandwidth is saturated. Let S_{comp} be that maximal per-thread contribution. However, both the compute threads and the copy threads must share the MCDRAM bandwidth. When the number of combined compute and copy threads is large enough to demand more than the available MCDRAM bandwidth, C_{comp} is limited to a per-thread share of that available bandwidth. The following equation expresses this conditional function.

$$C_{\text{comp}} = \begin{cases} S_{\text{comp}}, \\ \text{if } (p_{\text{comp}} * S_{\text{comp}}) + (p_{\text{in}} + p_{\text{out}}) * S_{\text{copy}}) \\ \le \text{MCDRAM}_{\text{max}}) \\ \\ \frac{\text{MCDRAM}_{\text{max}} - (p_{\text{in}} + p_{\text{out}}) * C_{\text{copy}}}{p_{\text{comp}}}, \text{ otherwise} \end{cases} \tag{5}$$

Note that this model neglects to consider the cases in which some of the thread pools are unoccupied. While the first chunk is being copied in, all threads are designated to perform the copy. Once the first chunk is in MCDRAM, the copy out threads can either join the copy-in or compute thread pools. Unless the number of chunks is small this simplification has a negligible effect on the performance of the model.

We shall return to these equations in Section 5. There we demonstrate their use to determine the optimal number of threads for executions in MCDRAM flat mode on KNL. Next, however, we examine a concrete example of an algorithm adapted to multilevel memory: sort.

4 SORTING USING MULTILEVEL MEMORY

Sort is an extensively studied problem in computer science with diverse applications [9, 14, 15]. For large problems it is often performed at scale on distributed clusters, *e.g.*, running Hadoop, where performance is greatly affected by the number of secondary storage units available. This approach is fast and efficient, but in this work we focus on how quickly a single KNL node can sort in memory without swapping to disk. A previous winning entry in the 'Terasort' competition demonstrated early use of GPUs for general purpose computation [13]. Similarly our work uses the KNL MCDRAM memory to provide speedups.

We first consider a basic algorithm that simplifies the sorting approach of Bender, *et al.*: Divide the data into chunks of equal size, sort each chunk, and then perform a multi-way merge between all the chunks [4].

The chunked-sorting phase of this basic algorithm is modeled in Section 3. To sort N elements with a chunk size of w, divide the data into chunks of size N/w. Sort each chunk using the best available multithreaded sorting algorithm (GNU parallel sort at the time of this writing [22]), and merge the sorted chunks using the best available multi-way merge (again GNU parallel [22]). The final multi-way merge does not use the chunking mechanisms or even explicitly take advantage of the MCDRAM. The multi-way merge is a small portion of the computation time and exploits prefetching well for the caches on the KNL cores. Therefore, we find that refactoring the multi-way merge to utilize the MCDRAM is unnecessary.

Bender, *et al.* used simulations to predict roughly a 30% advantage for this MCDRAM-aware chunking algorithm on KNL [4]. Although GNU parallel sort, when run in hardware cache mode, was expected to realize some portion of this 30% performance boost as well, Bender, *et al.* did not simulate this mode.

We implemented the basic algorithm described above and initially confirmed that the predicted KNL speedups of roughly 30% can be realized. However, we found that this algorithm yields no advantage over GNU parallel sort run in hardware cache mode.

In order to better demonstrate the potential of chunking algorithms for multi-level memory resources such as KNL nodes, we designed a new algorithm which we refer to as “MLM-sort.” Unlike the basic algorithm, MLM-sort does not rely on thread-scalability of multithreaded algorithms (to hundreds of cores) when sorting in high-bandwidth memory. Instead, MLM-sort relies on the best available *serial* sorting algorithm for the sorting of chunks. At a high level, MLM-sort is identical to the basic chunking algorithm: it divides the input array into MCDRAM-sized “megachunks,” sorts each megachunk, then uses multi-way merge to finish the global sort. The difference is that before sorting, MLM-sort divides each MCDRAM-resident megachunk into maximally-sized chunks such that each thread gets one chunk. Each thread sorts its chunk in serial, then a parallel multi-way merge (from MCDRAM to DDR) finishes the sort of the megachunk.

Section 4.1 shows our experiments with MLM-sort and GNU parallel sort. MLM-sort offers significant speedup over GNU parallel sort run in hardware cache mode. Furthermore, almost all of this performance advantage is retained by a variant we call “MLM-implicit.” This version runs MLM-sort with no explicit copying to or from MCDRAM, instead relying on hardware cache mode.

Even though MLM-sort executes on a single manycore compute node, it is natural to think of it as primarily a *distributed* rather than a multithreaded algorithm. The serial sorts are like single-node computations within a distributed-memory algorithm. Communication logic in the latter (typically stitching together subproblem results) corresponds to the multithreaded multi-merge steps in MLM-sort. If the subproblems are large enough so that communication does not dominate, then we leverage perfect speedup through hundreds of cores on the subproblem computations.

Secondly, MLM-implicit is directly motived by complicated Department of Energy (DOE) application workloads in which KNLs will always be booted into hardware cache mode. All but the most specialized computations will rely on hardware cache mode for any performance boosts due to MCDRAM.

MLM-implicit allows megachunk sizes greater than MCDRAM. Initially, we assumed that it would never make sense to specify such large chunks. Doing so would seem to encourage thrashing in high-bandwidth memory. However, the serial sorting operation run by each thread is itself a divide-and-conquer process (a quicksort variant for std::sort). The superior performance of MLM-implicit in most of our experiments is likely explained as follows. Each serial divide-and-conquer sort focuses on one subproblem at a time, and during most of the run these are small enough such that every thread can have its active set in MCDRAM.

The success of MLM-sort and MLM-implicit is encouraging, and we suggest exploring distributed-style design options for single-node computations on manycore

4.1 Performance Comparison for Sort

In Figure 6 we present speedups over GNU parallel sort in DDR (“GNU-flat”) for “GNU-cache” (GNU parallel sort in hardware cache mode, “MLM-ddr” (MLM in DDR only), and our main results: “MLM-sort” and “MLM-implicit” (described above). study considered various chunk sizes, numbers of OpenMP threads, and MCDRAM modes (flat, hardware cache, hybrid, “implicit” cache). However, we present only the highest performing options for MLM-sort and MLM-implicit. For MLM-sort, this is: 256 threads, and megachunk size of 1.5 billion elements for the runs with six billion elements. For all other problem sizes we use megachunk sizes of one billion elements. For MLM-implicit, we use megachunk size equal to problem size. Each result is the average of ten runs, and the raw data is shown in Table 1.

Note that reversed input arrays have structure that our MLM-sort variants exploit more effectively than the stock GNU algorithms. Also, note that MLM-ddr makes no use of the MCDRAM; our other MLM-sort variants exploit all of the memory (DDR and MCDRAM).

These results provide answers to some key questions:

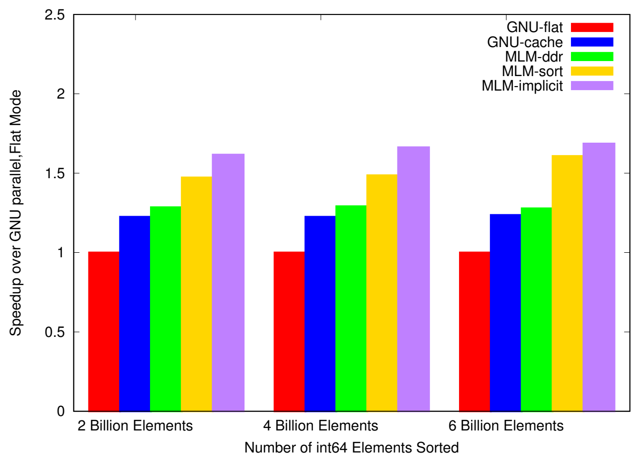
- If algorithms designed for use of the MCDRAM in flat mode, *e.g.*, processing the data in chunks, are used with the MCDRAM configured in cache mode, *significant performance gains* over an unchunked implementation can be obtained.
- For data sets that exceed the MCDRAM capacity, algorithms that explicitly place data into the MCDRAM in flat mode *can improve on the performance* observed from simply using the provided system-managed cache mode.

We observed superior performance with the machine in hardware cache mode using MLM-implicitwith megachunk size equal to the overall problem size. It is likely that this performance was possible only because the serial divide-and-conquer algorithms solve our subproblems. The chunked MLM-sort algorithm performed best when the machine was used in flat mode. For some other algorithms, this mode may offer the best peformance. In fact, we see this in sorting reversed arrays of six billion elements, though is it not clear why MLM-implicit lagged in that case.

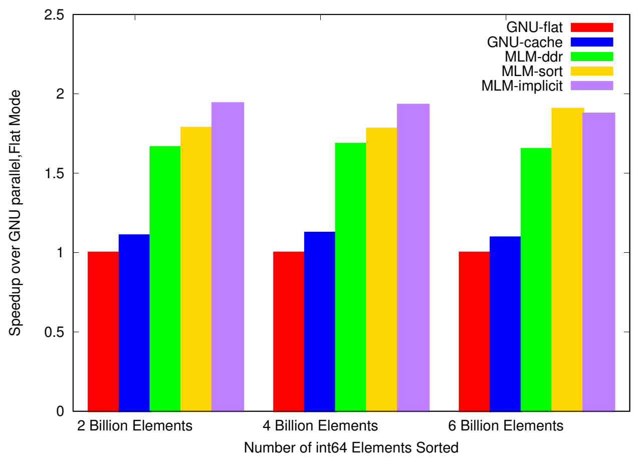
Whether chunking is implicit or not, we find that rewriting the sorting kernel for high-bandwith memory rather than running existing library routines in hardware cache mode yields significant speedups. We expect that this will hold for many bandwidth-bound algorithms.

4.2 The Impact of Chunk Size

To execute a chunked algorithm efficiently, the chunk size needs to be as large as the near memory will allow. While running in flat, implicit or hybrid the chunk size is ultimately limited by the size of the MCDRAM. The choice of chunk size also impacts the final merge step, and its performs best with only a small number of chunks to be merged. Fig. 7 shows the performance of the different configuration with a fixed problem size and thread count but varying chunk size.



(a)



(b)

Figure 6: (a) Performance of sort in various MCDRAM configurations on randomized input arrays. (b) Performance of sort in various MCDRAM configurations on reverse sorted input arrays.

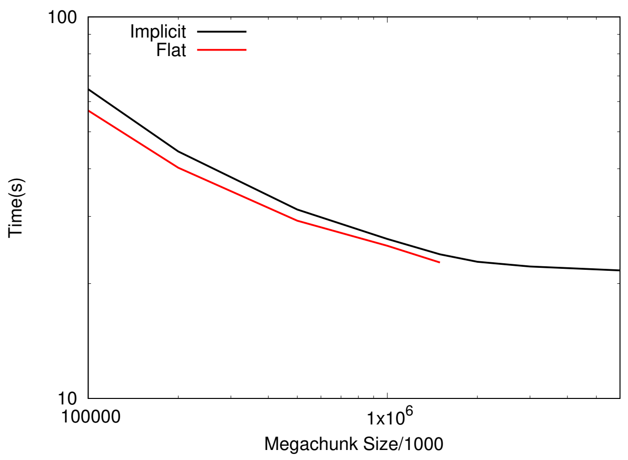


Figure 7: Performance of the chunked sort implementation (6 billion int64 elements) using different MCDRAM configurations and varying the chunk size. Note that MLM-implicit can continue improving as megachunk size exceeds MCDRAM.

The chart shows that in both flat and implicit the larger the chunk size the better the performance. We have run in hybrid mode using our sorting algorithm, but do not report specific results here. The hybrid mode shows near identical performance to flat, given a chunk size. Since we prefer large chunk sizes, and the chunk size in hybrid cannot be as large as the chunk size in flat mode, we obtain our best results in either flat or implicit mode.

5 ANALYSIS OF A STREAMING BENCHMARK

Although our MLM-sort algorithm differs from the basic triple-buffered approach described at the beginning of Section 4, we note that the chunking and buffering approach of the latter is likely to be a common approach to multilevel-memory algorithm design. We now evaluate the model introduced in Section 3.2 to provide guidance on the sizes of compute and copy thread pools in that context.

In general, as the computation time gets larger the need for copy threads is decreased. This assertion contrasts with prior work that found a need for multiple copy threads to optimize MCDRAM buffering for benchmarks with short compute phases [18].

To test our hypothesis, we developed a simple "merge" benchmark and analyzed it through empirical evaluation. The concept is simple: using the generic multilevel approach from Section 3, we designate the compute stage to be a simple merge. The data in each chunk is evenly dispersed among the threads. Each thread chops its portion in half and performs a merge on each of the two halves. We allow the merge to be repeated as a parameter. We refer to the number of times merge is performed as the number of 'repeats'. The data is brought in/out of only MCDRAM once, but the amount of work in the compute stage is dictated by the number of repeats. The repeat parameter allows adjustment of the amount of work to be done in the compute stage while keeping the amount of work to be done in the copy stage to be constant.

The merge benchmark enables a more complete understanding of the relationship between amount of work available and the effectiveness of adding additional copy threads. As we expressed in

Elements	Input Order	Algorithm	Mean(s)	Std. Dev.(s)
2,000,000,000	random	GNU-flat	11.92	0.1662
2,000,000,000	random	GNU-cache	9.73	0.1777
2,000,000,000	random	MLM-ddr	9.28	0.0043
2,000,000,000	random	MLM-sort	**8.09**	0.0072
2,000,000,000	random	MLM-implicit	**7.37**	0.0186
4,000,000,000	random	GNU-flat	24.21	0.1638
4,000,000,000	random	GNU-cache	19.76	0.1892
4,000,000,000	random	MLM-ddr	18.74	0.0113
4,000,000,000	random	MLM-sort	**16.28**	0.0080
4,000,000,000	random	MLM-implicit	**14.56**	0.2288
6,000,000,000	random	GNU-flat	36.52	0.2565
6,000,000,000	random	GNU-cache	29.53	0.3412
6,000,000,000	random	MLM-ddr	18.74	0.0113
6,000,000,000	random	MLM-sort	**22.71**	0.0099
6,000,000,000	random	MLM-implicit	**21.66**	0.3154
2,000,000,000	reverse	GNU-flat	7.97	0.2446
2,000,000,000	reverse	GNU-cache	7.19	0.2069
2,000,000,000	reverse	MLM-ddr	4.79	0.0049
2,000,000,000	reverse	MLM-sort	**4.46**	0.0128
2,000,000,000	reverse	MLM-implicit	**4.10**	0.0183
4,000,000,000	reverse	GNU-flat	16.06	0.3832
4,000,000,000	reverse	GNU-cache	14.27	0.1739
4,000,000,000	reverse	MLM-ddr	9.53	0.0130
4,000,000,000	reverse	MLM-sort	**9.02**	0.0129
4,000,000,000	reverse	MLM-implicit	**8.31**	0.0098
6,000,000,000	reverse	GNU-flat	23.94	0.5884
6,000,000,000	reverse	GNU-cache	21.85	0.3622
6,000,000,000	reverse	MLM-ddr	14.48	0.0200
6,000,000,000	reverse	MLM-sort	**12.56**	0.0086
6,000,000,000	reverse	MLM-implicit	**12.76**	0.0159

Table 1: Raw sorting performance (averages of 10 runs each). The best runs in flat mode (MLM-sort) and hardware cache mode (MLM-implicit) are highlighted.

our model, the overall time to execute a step of the algorithm is the maximum of the compute time and the copy time. An efficient implementation would use the minimum number of copy threads needed to keep copy time as close as possible to the compute time. Thus, the expected behavior of the benchmark is that as the computational workload increases, fewer copy threads are required to overlap computation and data transfer.

The simplicity of the merge benchmark makes it an ideal target to demonstrate the effectiveness of the model discussed in Section 3.2. We obtained values for these parameters from system measurements and problem characteristics. The values for the parameters and their descriptions are shown in Table 2. To ensure the correctness of the coefficients, we used actual execution times from the benchmark as a validation of our model.

Empirical results from the benchmark are shown in Figure 8(b). As predicted, increasing the number of repeats drives down the optimal number of threads to perform the compute. These results only contain data from runs that use 1, 2, 4, 8, 16 or 32 copy threads. Often the optimal number of threads is somewhere nearby the observed optimal.

Parameter	Value	Description
B_{copy}	14.9 GB	Data size
DDR_{max}	90 GB/s	Maximum DDR bandwidth, as measured by the STREAM benchmark [17].
$MCDRAM_{\text{max}}$	400 GB/s	Maximum MCDRAM bandwidth, as measured by the STREAM benchmark [17].
S_{copy}	4.8 GB/s	Per-thread data transfer speed between MCDRAM and DDR when not bandwidth-limited
S_{comp}	6.78 GB/s	Per-thread computation speed when not bandwidth-limited

Table 2: Parameters for the model introduced in Section 3, based measurements of the system and the merge benchmark problem.

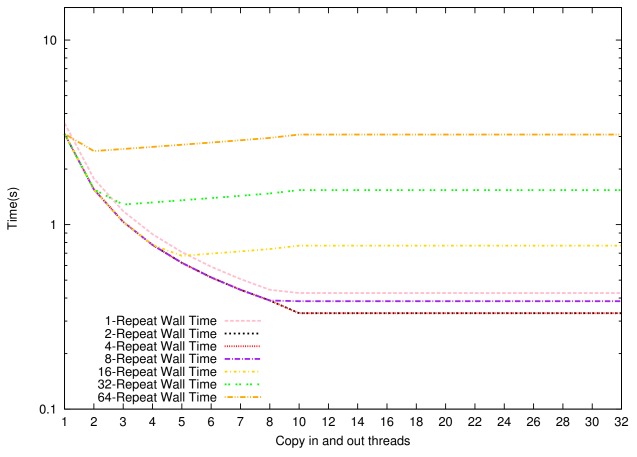
Optimal Number of Copy Threads For Merge Benchmark		
Number of Repeats	Model	Empirical (Powers of 2)
1	10	16
2	10	16
4	10	8
8	8	4
16	3	2
32	2	2
64	1	1

Table 3: Optimal number of copy threads for both the model and empirical results.

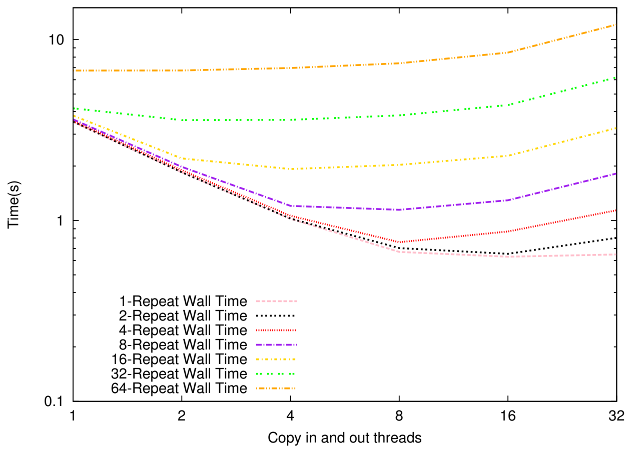
The performance predictions of the model are shown in Fig 8(a). The purpose of the model is to estimate the optimal number of compute and copy threads. Table 3 shows both the optimal numbers of copy threads from the empirical results and the numbers that the model predicts to be optimal. The numbers do not match exactly, and in particular note that we only tested powers of two number of threads in the empirical evaluation. Generally these results show that our model's predictions provide appropriate decisions of the number of copy threads to use.

6 CONCLUSION

A primary challenge when porting applications to Knight's Landing and similar architectures is to decide if the application can significantly benefit from using multilevel memory. Applications tend to use a mix of different algorithms. Some algorithms are memory bandwidth bound, while others are compute bound or memory latency bound. This unpredictability may prompt many application developers to rely solely on the cache mode of KNL or similar architectures, or to employ a hybrid mode with some high-bandwidth memory used as a cache. Algorithms to utilize flat mode can be difficult to design, especially when explicit copying between memory levels is performed. Issues such as buffer sizes, the ratio of copy to compute threads, and overlapping of the operations all must be worked through.



(a)



(b)

Figure 8: (a) Estimated execution times for the merge benchmark based on the model for buffering MLM algorithms from Section 3.2. The minimum time for each number of repeats yields a prediction for the optimal number of copy-in threads (with the same number of copy-out threads. (b) Empirical Execution Time for the merge benchmark for 1 to 64 repeats with varying copy threads.

Our work demonstrates the effectiveness of using chunking and buffering for MCDRAM in the context of a largely bandwidth-constrained sort kernel. These methods provide a performance benefit for all configurations studied, with the greatest benefit being observed in flat mode. We observe performance speedup of approximately 1.6-1.9X (depending on input order) times that of using the non-chunking GNU sort without MCDRAM.

What is striking is that optimal or near-optimal performance (usually dominating the best flat mode results, see Figure 6 and Table 1) can be achieved in hardware cache mode using a simple chunking algorithm. The chunking algorithm removes explicit data transfers, and simply accesses the original data from higher level memory. This observation is particularly useful to developers of large applications wishing to adapt their codes to use the KNL.

Fig. 7 indicates that at least for sorting, chunk sizes of 1-1.5GB are sufficient to provide near-minimal execution times. Knowing this number for real applications provides insight into how to manage MCDRAM for complex problems that employ multiple kernels where other data should remain in MCDRAM. We leave as future work the question of buffering in our MLM-sort algorithm. At the moment, we require all threads during the multiway merges of chunks into megachunks. However, a slightly different approach might allow hiding the copy-in latency of the next megachunk.

We also analyzed a simple merge benchmark to explore the interplay between the amount of computation in each step of a buffering implementation and the allocation of threads for computation and copying. The model constructed here is based on the rate at which an individual thread can make demands on the different levels of memory. This provides a convenient basis for estimating in flat mode the optimal number of copy threads to perform transfers.

Finally, this work considers different MCDRAM usage models in a single KNL node for bandwidth-driven kernels. Future work will extend this to multiple KNL nodes. Another level of memory is also conceivable, *e.g.*, high capacity storage based on non-volatile memory such as 3D-XPoint. The larger memory capacity of such architectures will accommodate a much larger problem size, but now there may be double levels of chunking to consider. In addition, we intend to examine more complex benchmarks and applications that exhibit non-uniform data access patterns for which a chunking approach is not obvious. Lastly, using a variation of the model, we will explore alternative configurations that may be possible in future technologies, in hopes of suggesting more optimal design points for both hardware and applications.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Nuclear Security Administration Predictive Science Academic Alliance Program and by the Laboratory Directed Research and Development program at Sandia National Laboratories, in part by the Univ. of Notre Dame, and in part by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002377, as part of the Center for Shock-Wave Processing of Advanced Reactive Materials, Univ. of Notre Dame.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Dept. of Energy's National Nuclear

Security Administration under contract DE-NA-0003525. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

REFERENCES

[1] Alok Aggarwal, Jeffrey Vitter, et al. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[2] Ritu Arora and Lars Koesterke. 2017. Interactive Code Adaptation Tool for Modernizing Applications for Intel Knights Landing Processors. In *Proceedings of Practice and Experience in Advanced Research Computing 2017 (PEARC17)*. ACM, New York, NY, USA, Article 28, 8 pages.

[3] JEDEC Solid State Technology Association. 2015. JEDEC Standard High Bandwidth Memory (HBM) DRAM Specification, Standard JESD235A". (2015).

[4] Michael A. Bender, Jonathan W. Berry, Simon D. Hammond, K. Scott Hemmert, Samuel McCauley, Branden Moore, Benjamin Moseley, Cynthia A. Phillips, David Resnick, and Arun Rodrigues. 2017. Two-level main memory co-design: Multi-threaded algorithmic primitives, analysis, and simulation. *J. Parallel and Distrib. Comput.* 102 (2017), 213 – 228.

[5] Michael A Bender, Roozbeh Ebrahimi, Jeremy T Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. 2014. Cache-adaptive algorithms. In *Proceedings of the 25th ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 958–971.

[6] Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. 2002. Cache oblivious search trees via binary trees of small height. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 39–48.

[7] Gerth Stølting Brodal, Rolf Fagerberg, and Kristoffer Vinther. 2008. Engineering a cache-oblivious sorting algorithm. *Journal of Experimental Algorithmics (JEA)* 12 (2008), 2–2.

[8] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. 2015. *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*. Technical Report SAND2015-1862C. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States).

[9] Jonathan M. Cohen, Sarah Tariq, and Simon Green. 2010. Interactive Fluid-particle Simulation Using Translating Eulerian Grids. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '10)*. ACM, New York, NY, USA, 15–22.

[10] Nicolas Denoyelle, Brice Goglin, Aleksandar Ilic, Emmanuel Jeannot, and Leonel Sousa. 2017. Modeling Large Compute Nodes with Heterogeneous Memories with Cache-Aware Roofline Model. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS 2017)*, Stephen Jarvis, Steven Wright, and Simon Hammond (Eds.). Springer, Cham, 91–113.

[11] Douglas Doerfler, Jack Deslippe, Samuel Williams, Leonid Oliker, Brandon Cook, Thorsten Kurth, Mathieu Lobet, Tareq M. Malas, Jean-Luc Vay, and Henri Vincenti. 2016. Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor.. In *ISC Workshops (Lecture Notes in Computer Science)*, Michela Taufer, Bernd Mohr, and Julian M. Kunkel (Eds.), Vol. 9945. Springer, Cham, 339–353.

[12] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. 1999. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*. IEEE, Washington, DC, USA, 285–297.

[13] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 325–336.

[14] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4, Article 21 (Dec. 2009), 39 pages.

[15] Yuji Kohara, Kiyotaka Akiyama, and Katsumi Isono. 1987. The physical map of the whole E. coli chromosome: Application of a new strategy for rapid analysis and sorting of a large genomic library. *Cell* 50, 3 (1987), 495 – 508.

[16] Ang Li, Weifeng Liu, Mads R. B. Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 2017. Exploring and Analyzing the Real Impact of Modern On-package Memory on HPC Scientific Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 26, 14 pages.

[17] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[18] Stephen L. Olivier, Simon D. Hammond, and Alejandro Duran. 2017. Double buffering for MCDRAM on Second Generation Xeon Phi Processors with OpenMP. In *Proceedings of the 13th Internanational Workshop on OpenMP (IWOMP 2017): Scaling OpenMP for Exascale Performance and Portability (Lecture Notes in Computer Science)*, Bronis R. de Supinski, Stephen L. Olivier, Christian Terboven, Barbara M. Chapman, and Matthias S. Müller (Eds.), Vol. 10468. Springer, Cham, 311–324.

[19] OpenMP Architecture Review Board. 2017. OpenMP Technical Report 6: Version 5.0 Preview 2. http://www.openmp.org/wp-content/uploads/openmp-TR6.pdf. (Nov. 2017).

[20] S. Rajasekaran. 2001. A Framework for Simple Sorting Algorithms on Parallel Disk Systems. *Theory of Computing Systems* 34, 2 (01 Apr 2001), 101–114.

[21] Sabela Ramos and Torsten Hoefler. 2017. Capability Models for Manycore Memory Systems: A Case-Study with Xeon Phi KNL. In *2017 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2017*. IEEE, Washington, DC, USA, 297–306.

[22] Johannes Singler and Benjamin Konsik. 2008. The GNU Libstdc++ Parallel Mode: Software Engineering Considerations. In *Proceedings of the 1st International Workshop on Multicore Software Engineering (IWMSE '08)*. ACM, New York, NY, USA, 15–22. https://doi.org/10.1145/1370082.1370089

[23] Johannes Singler, Peter Sanders, and Felix Putze. 2007. MCSTL: The Multi-core Standard Template Library. In *Euro-Par 2007 Parallel Processing*, Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 682–694.

[24] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. 2016. Knights landing: Second-generation Intel Xeon Phi product. *IEEE Micro* 36, 2 (2016), 34–46.