Explicit Data Layout Management for Autotuning Exploration on Complex Memory Topologies

Swann Perarnau, Brice Videau, Nicolas Denoyelle, Florence Monna, Kamil Iskra, Pete Beckman
Argonne National Laboratory
{swann, ndenoyelle, bvideau, fmonna}@anl.gov, {iskra, beckman}@mcs.anl.gov

***Abstract*—The memory topology of high-performance computing platforms is becoming more complex. Future exascale platforms in particular are expected to feature multiple types of memory technologies, and multiple accelerator devices per compute node.**

In this paper, we discuss the use of explicit management of the layout of data in memory across memory nodes and devices for performance exploration purposes. Indeed, many classic optimization techniques rely on reshaping or tiling input data in specific ways to achieve peak efficiency on a given architecture.

With autotuning of a linear algebra code as the end goal, we present `AML`: a framework to treat three memory management abstractions as first-class citizens: data layout in memory, tiling of data for parallelism, and data movement across memory types. By providing access to these abstractions as part of the performance exploration design space, our framework eases the design and validation of complex, efficient algorithms for heterogeneous platforms.

Using the Intel Knights Landing architecture in one of its most NUMA configurations as a proxy platform, we showcase our framework by exploring tiling and prefetching schemes for a DGEMM algorithm.

***Index Terms*—Deep memory, high-bandwidth memory, explicit memory management**

I. Introduction

As we approach exascale, high-performance computing (HPC) platforms are increasingly featuring complex memory topologies. Intel's Knights Landing [1] (KNL) processor was an early indicator of this trend, with a configuration that can result in a single socket being split into 8 NUMA domains: 4 quadrants for the on-chip network and 2 types of memory (high-bandwidth MCDRAM and regular DRAM) per quadrant. Exascale-class systems are also expected to feature a topology including multiple accelerator devices with their own memory banks along with their host multicore processor. Regardless of the specific technology, these systems will exhibit **deep memory**: multiple levels of cache-coherent, byte-addressable memory.

How these complex topologies should be managed by HPC applications is still an open research question. Vendors have spent considerable effort on hiding most of this complexity behind automatic mechanisms that expose a single coherent virtual address space with each compute element pulling data as close as possible when needed. This is a case for the *cache mode* of the KNL, which uses the high-bandwidth memory as a last-level direct-mapped cache, as well as for the *unified memory* feature on Nvidia GPUs, which allows the GPU to pull pages of data allocated on the CPU side as needed.

We believe that these automatic cache-like management schemes are leaving performance on the table and, more importantly, prevent the internal structure of applications from treating the placement, movement, and shape of data as key performance concerns. In this paper, we revisit typical memory optimization strategies in the autotuning of dense linear algebra kernels as first-class abstractions that can be part of an application design space for heterogeneous architectures. These abstractions are implemented in **AML**: a library providing *building blocks* for the creation of explicit, application-aware memory management policies. Our library allows application and runtime developers to quickly design memory management schemes adapted to complex memory hierarchies, while still being agnostic to the specific hardware topology exhibited by the architecture. As a memory management framework, AML provides facilities to build custom data layouts, adapt memory placement according to those layouts, and handle coarse-grained memory movement between memory layers, as required for prefetching mechanisms.

To motivate exposing those abstractions to application developers, we present an extensive performance exploration study combining autotuning and careful data movement orchestration to achieve efficiency on large input sizes for a DGEMM algorithm on a KNL system configured as a large NUMA topology. We demonstrate that our library enables us to implement efficient tiling schemes and dynamic memory movements to prefetch inputs into the more efficient memory.

This paper is organized as follows. Section II presents our motivation: the reproduction, using autotuning, of an optimization strategy for DGEMM with large inputs. Using this complex memory management optimization as a starting point, we present in Section III three abstractions necessary for building a complex data orchestration policy on a heterogeneous topology and discuss how they can be provided to application performance specialists as building blocks inside the same framework. We explore the full performance of our data orchestration scheme in Section IV. Section V discusses related work, and we conclude in Section VI with a summary and some observations about future platform requirements.

II. Motivation

Our motivating problem is the fine-grained optimization of a DGEMM kernel ($C = A * B + C$) for large matrix sizes, utilizing a state-of-the-art strategy [2]. This strategy is a bottom-up, architecture-aware approach for maximizing performance: for each level of the memory hierarchy of the target architecture, use subkernels and reorganize data to achieve the best compute intensity (ratio of memory loads to compute instructions). Such a strategy involves finely tuned reorganization of data across cache levels in a non portable way. For productivity and portability reasons, we would like to generalize and simplify the memory management of this kind of approach.

The DGEMM kernel proposed in [2] is tailored to target the Knights Landing architecture, which is notoriously difficult to tune for. In order to reach peak performance, every level of the memory hierarchy needs to be accounted for. The KNL node used in this study has 64 cores with 32 vector registers 512 bits wide (8 double-precision floats), 32 KiB of L1 cache and 1 MiB of L2 cache shared between each pair of cores. The node also has 16 GiB of MCDRAM memory, and although it can be used as a cache, it will exposed as a NUMA memory node in this study. Our strategy thus involves 4 levels of kernel optimization.

Inner Kernel: The inner kernel is optimized for register-level data access patterns. Register reuse is achieved by careful vectorization and limited use of load and store instructions. Specifically, the inner kernel works on small tiles of A, B, and C with sizes $[kb, mr]$, $[kb, nr]$, and $[mr, nr]$, respectively. We denote these tiles $\widehat{A}$, $\widehat{B}$, and $\widehat{C}$ (note that $\widehat{A}$ is transposed). Tile sizes are chosen so that nr is a multiple of the vector length (8 in our case), while mr is small enough that the whole tile $\widehat{C}$ and a row of $\widehat{B}$ fit inside the registers simultaneously. Thus, for this architecture, the possible values of $[mr, nr]$ are $(31, 8)$, $(15, 16)$ and $(7, 32)$. The last parameter, kb, should be large enough to amortize data transfer overheads on $\widehat{C}$ but also small enough for $\widehat{A}$ to fit inside half of the L1 cache.

Intermediary Kernel: The intermediary kernel works on sets of tiles of A, B, and C with sizes of $[kb, mb]$, $[kb, n]$, and $[mb, n]$, respectively. We denote these tilesets $\widetilde{A}$, $\widetilde{B}$, and $\widetilde{C}$. $\widetilde{A}$ is stored as mb/mr (or $nblocka$) consecutive tiles of size $[kb, mr]$ ($\widehat{A}$), $\widetilde{C}$ is stored as $(mb/mr) * (n/nr)$ (or $nblocka * nblockn$) consecutive tiles of size $[mr, nr]$ ($\widehat{C}$), and $\widetilde{B}$ is stored as n/nr (or $nblockn$) tiles of size $[kb, nr]$ ($\widehat{B}$). These sizes are chosen to take advantage of L2 caches, and in particular mb is chosen so that $\widetilde{A}$ fits inside half of the L2 cache available to a core.

Outer Kernel: The outer kernel updates a block C of size $[m, n]$, using a block A of size $[k, m]$ and a block B of size $[k, n]$. The layouts used here are as follows: C is stored as m/mb (or $nblockm$) blocks of $\widetilde{C}$, A is stored as $(k/kb) * (m/mb)$ (or $nblockk * nblockm$) blocks of $\widetilde{A}$, and B is stored as (k/kb) (or $nblockk$) blocks of $\widetilde{B}$. This is where our approach differs from [2]. In the original algorithm, the transposition of $\widehat{A}$ is performed in the intermediary kernel, while $\widehat{B}$ is transposed in the outer kernel. We extracted those transformations to take place before the outer kernel. This approach results in another level of blocking in the algorithm: the outer kernel is operating on blocks of A, B, and C with a shape of $[nblockk, nblockm, nblocka, kb, mr]$, $[nblockk, kb, nblockn, nr]$, and $[nblockm, nblockn, nblocka, mr, nr]$, respectively. We denote them $\dot{A}$, $\dot{B}$, and $\dot{C}$. The sizes m, n, and k are chosen so that the blocks are approximately square and large enough to amortize the transform cost. The pseudocode for this entire algorithm is given in Listing 1.

```
void inner_kernel(
  double ah[KB][MR],
  double bh[KB][NR],
  double ch[MR][NR]) {
  /* ch += ah * bh */
}

void intermediary_kernel(
  int nblockn,
  double at[NBLOCKA][KB][MR],
  double bt[nblockn][KB][NR],
  double ct[nblockn][NBLOCKA][MR][NR]) {
#pragma omp parallel for num_threads(NUM_INNER_TH)
  for (int jr = 0; jr < nblockn; jr++) {
    for (int ir = 0; ir < NBLOCKA; ir++) {
      inner_kernel( &at[ir][0][0],
                    &bt[jr][0][0],
                    &ct[jr][ir][0][0]);
    }
  }
}

void outer_kernel(
  int nblockm, int nblockn, int nblockk,
  double ad[nblockk][nblockm][NBLOCKA][KB][MR],
  double bd[nblockk][nblockn][KB][NR],
  double cd[nblockm][nblockn][NBLOCKA][MR][NR]) {
  for (int p = 0; p < nblockk; p++) {
#pragma omp parallel for num_threads(NUM_OUTER_TH)
    for (int i = 0; i < nblockm; i++) {
      intermediary_kernel(
        nblockn,
       &ad[p][i][0][0][0]
       &bd[p][0][0][0],
       &cd[i][0][0][0][0]);
    }
  }
}
```

Listing 1. Pseudocode for the DGEMM outer kernel.

Top Kernel: The top kernel is responsible for orchestrating the transformation of the input matrices and the launch of the outer kernel. Matrix $\bar{C}$ is of size $[M, N]$, matrix $\bar{A}$ of size $[M, K]$, and matrix $\bar{B}$ of size $[K, N]$. The matrices are arranged in blocks of size $[m, n]$ (C), $[m, k]$ (A), and $[k, n]$ (B), respectively. Those blocks are transferred from main memory to MCDRAM and transformed on the fly into blocks $\dot{C}$, $\dot{A}$, and $\dot{B}$, respectively. Since MCDRAM can hold several of those blocks simultaneously, computation and transfer can overlap, and some blocks can be reused, depending on the order of evaluation.

Challenge: This multilayered algorithm has one major drawback: its memory management is not an explicit parameter of the algorithm. Indeed, to port this algorithm to a different architecture, one must rediscover the appropriate shapes and sizes of the data for each topology level. To automate this work, memory management must be made explicit and composable enough that an application performance specialist can automate the search for ideal parameters, using autotuning for example. Moreover, the layouts are complex enough that facilities to track and reason about these geometries are required.

III. Abstractions for Efficient In-Memory Data Orchestration

Our motivating problem points towards the need for a principled approach to three memory abstractions: layout, tiling, and movement across a topology. In order to improve on existing algorithms and adapt our work to future platforms, these three abstractions need to become available as first-class constructs that we can use for implementation and for further experimentation.

Such goal is also in line with the PADAL whitepaper [3], which identified that as we approach exascale, high-performance computing platforms will move toward cheap and massively parallel compute power while data movement will dominate energy and performance costs. To facilitate the development of applications on those platforms, the authors called for the community to establish locality-preserving abstractions. Data layout, tiling, and topology were identified as critical components of such an effort.

We present here how these three abstractions can be made available inside a single framework, as building blocks for further performance optimization studies. Our framework should have the following goals:

- *Composable*: application developers and performance experts should be able to pick and choose which building blocks to use depending on their specific needs.
- *Flexible*: one should be able to customize, replace, or change the configuration of each building block as much as possible.
- *Declarative*: to the extent possible, each building block should provide the means for users to describe how it is used by the application, without having to change the existing programming model.
- *Hardware-oblivious*: given the right initialization parameters, the resulting memory management policies should work on any hardware configuration.

Given these goals, we detail the scope and intent for each abstraction and highlight how they interact with one another.

Data Layouts: This abstraction is in charge of representing how the bytes of a data structure are organized in a linear virtual address space. In this paper, we focus on layouts typically encountered when dealing with dense linear algebra algorithms: multidimensional arrays of a single element type, with optional strides. This abstraction provides methods to find a single element and, more important, slicing and reshaping. We identify slicing as the act of returning a subset of the layout; reshaping provides the user with a view of the layout using different dimensions (without changing the underlying data organization).

Tiling Data: Tiling, or blocking, is a common optimization strategy to improve the data locality of an algorithm. It can be summarized as the action of grouping data elements to be placed or worked on together. Based on our motivating problem, tiling is the abstraction in charge of providing an indexation and generation mechanism over slices of a layout. Indeed, for a blocked DGEMM algorithm it is critical to be able to reason about the layout of entire input matrices as tiles for which coordinates are available so that the tiles can be iterated on correctly.

Table I
Summary of Data Management Abstractions

Abstraction	Intent	Methods
Layout	data organization in memory	`deref`, `reshape`
Tiling	generate/index slices	`index`, `slice`
Movement	orchestrate movement	`copy`, `transform`

Moving Data: The core abstraction behind the movement of data inside a topology can be identified by two main functions: the means to perform the movement itself and the type of movement being performed. Indeed, depending on the architecture and the specific programming model, moving data between memories can be done by a regular `memcpy`, by moving physical pages of data around, by calling into device-specific APIs (`cudaMemcpy`), by queuing copy operations (`clEnqueueCopyBuffer` in OpenCL), or even by using parallel runtimes (OpenMP 5.0). On the other hand, the data movement operation itself might include more than just copying and moving data around, for example, transposing matrices for better efficiency or packing/filtering the data to improve locality.

These abstractions, summarized in Table I, were implemented in **AML**, a lightweight framework written in C99 using pthreads for its asynchronous movement facilities.

IV. Performance Exploration of DGEMM on a Large Heterogeneous Topology

We now present how our memory management framework can be used to generalize the DGEMM optimization presented earlier, on Intel's Knights Landing architecture.

A. Experimental Setup

All the experiments presented in this section run on an Intel Knights Landing processor model 7210, comprising 16 GiB of high-bandwidth, on-package MCDRAM and 192 GiB of DDR. This platform has 64 cores available, running at 1.3 GHz, and hyperthreading is deactivated. Unless otherwise specified, the node is configured at boot time to run in Flat/Quad mode, meaning that the MCDRAM is exposed to the system as a single NUMA node and that the distributed cache directory is configured to improve latency on the cache misses.

The node is running CentOS 7.5, kernel version 3.10. The frequency governor has been set to `performance`. The benchmarks are compiled with icc 17.0.1; architecture-specific optimizations (`-xHost`) are active. All experiments are run using 60 cores for computation (with OpenMP) and 4 cores for data movement. The inner kernel is generated ahead of time by using the BOAST autotuning framework [4].

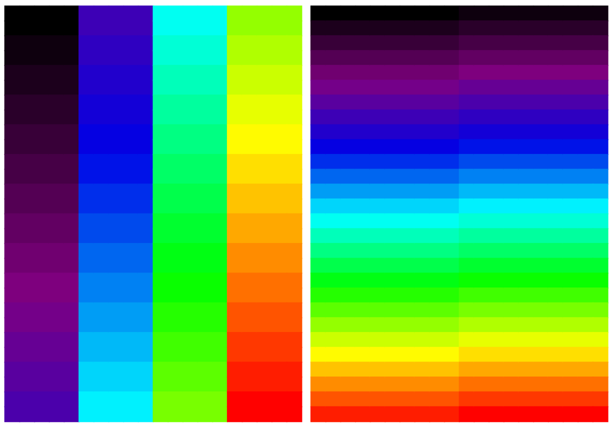

Figure 1. Example of a $\dot{A}$ before and after transposition. Elements of each $\widehat{A}$ have the same color.

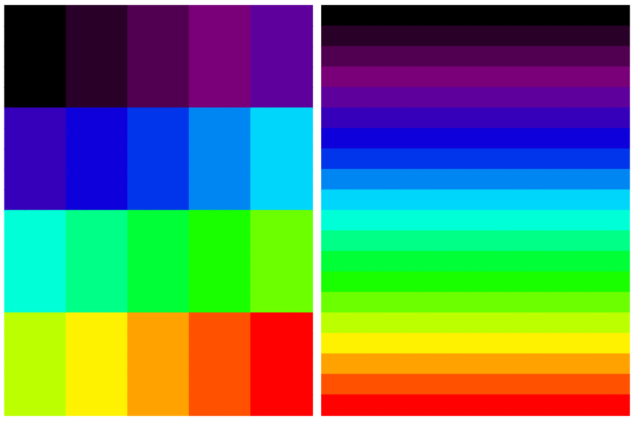

Figure 2. Example of a $\dot{B}$ before and after transposition. Elements of each $\widehat{B}$ have the same color.

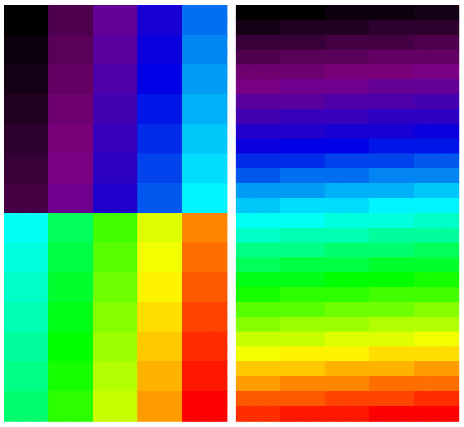

Figure 3. Example of a $\dot{C}$ before and after transposition. Elements of each $\widehat{C}$ have the same color.

```
/*
A dims: {NB, M, NB, K};
B dims: {NB, K, NB, N};
C dims: {NB, M, NB, N};

Tiles:
A dims:        {M, K};
A reshaped:    {NBLOCKM, NBLOCKA, MR, NBLOCKK, KB};
A transposed:  {NBLOCKK, NBLOCKM, NBLOCKA, KB, MR};

B dims:        {K, N};
B reshaped:    {NBLOCKK, KB, NBLOCKN, NR};
B transposed:  {NBLOCKK, NBLOCKN, KB, NR};

C dims:        {M, N};
C reshaped:    {NBLOCKM, NBLOCKA, MR, NBLOCKN, NR};
C transposed:  {NBLOCKM, NBLOCKN, NBLOCKA, MR, NR};

Tiles are prefetched from DDR to MCDRAM and
reshaped and transposed during transfer.
*/
void top_kernel( const double *A, const double *B,
                 double *C ) {

  prefetch(firstTileOf_A);
  prefetch(firstTileOf_B);
  prefecth(firstTileOf_C);
  for (int i = 0; i < block_number; i++) {
    for (int j = 0; j < block_number; j++) {
      wait(tile of C);
      prefetch(nextTileOf_C)
      for (int k = 0; k < block_number; k++) {
        wait(tileOf_A);
        prefetch(nextTileOf_A);
        wait(tileOf_B);
        prefetch(nextTileOf_B);
        outer_kernel(NBLOCKM, NBLOCKN, NBLOCKK,
          tileOf_A , tileOf_B , tileOf_C);
      }
      wait(flushPreviousTileOf_C);
      flush(tileOf_C);
    }
  }
  wait(flushLastTileOf_C);
}
```

Listing 2. Pseudocode for the DGEMM top kernel.

Inline Transformation Top Kernel: The DGEMM algorithm presented in Section II is adapted to use our framework to handle its tiling as well as the data movement and simultaneous transformations. Data movement is executed ahead of time when possible, using a scheme similar to double buffering. That is, for each matrix, the next block of the outer kernel is prefetched. Result tiles (those of matrix C) are copied back into the source data in DRAM, too. To perform this data movement, dedicated threads each perform active polling on a workqueue protected by a spinlock (one lock per queue per thread). Each thread is in charge of one of the 4 operations in this algorithm: prefetch and transform $\dot{A}$, $\dot{B}$, $\dot{C}$, and flush previous $\dot{C}$. Figures 1, 2, and 3 showcase examples of these transformations between the top and outer kernel, for $MR = 2$, $NR = 3$, $KB = 5$, $NBLOCKA = 7$, $NBLOCKM = 2$, $NBLOCKN = 5$, $NBLOCKK = 4$. Listing 2 provides the pseudocode for this kernel, while detailing the layouts and data movements.

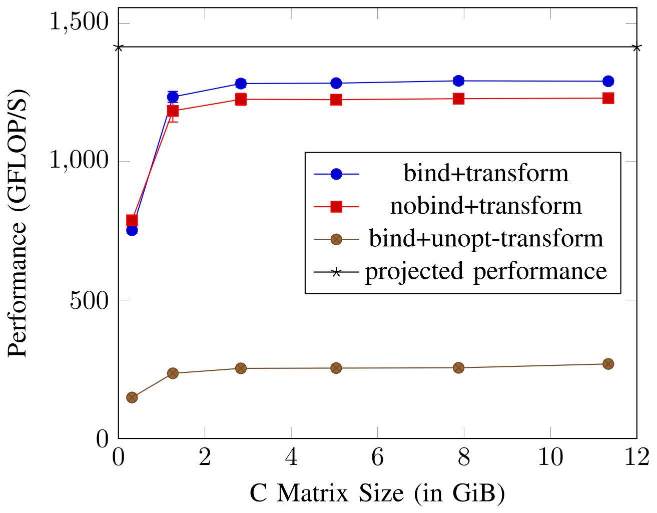

Figure 4. DGEMM performance depending on transform implementation.

Performance Exploration: For our performance exploration we compare several versions of our kernel, depending on two changes:

- Transform performance: whether the implementation uses a generic and unoptimized algorithm to handle each transformation or a custom code, optimized for the specific block sizes involved here,
- Thread binding: whether the threads performing the data transformation share cores with the regular worker threads or use dedicated resources.

Figure 4 presents the results of these experiments, for varying sizes of the result matrix. Experiments are repeated 10 times, and standard error is displayed. We included in the figure the performance of one tile of the outer kernel, as a reference for maximum reachable performance for this strategy. Note that this performance is in line with the reported data in the original study.

These results highlight the impact of the method of transform and the placement of data management threads on overall performance of this algorithm. This is precisely the kind of experiments that are made easier by providing higher-level abstractions about memory management in a complex algorithm. We hope to explore further these design points for this algorithm and others in the future.

V. Related Work

Deep memory architectures have become widely available only in the past couple of years, and studies focusing on them are rare. Furthermore, since vendors recommend using them as another level of hardware-managed cache, few works make the case for explicit management of these memory types. Among existing ones, two major trends can be identified: studies arguing either for *data placement* or for *data migration*.

Data placement [3] addresses the issue of distributing data among all available memory types only once, usually at allocation time. Several efforts in this direction aim at simplifying the APIs available for placement, similar to work on general NUMA architectures: memkind [5], the Simplified Interface for Complex Memory [6], and Hexe [7]. These libraries provide applications with intent-based allocation policies, letting users specify *bandwidth-bound* data or *latency-sensitive* data, for example. While placement is critical for the efficient use of deep memory architectures, these mechanisms lack the means to move data around to accommodate workloads that cannot fit in the right layer, resulting in missing optimization opportunities. Nevertheless, our framework also provides placement features as the basis for its data movement facilities.

Data migration addresses the issue of moving data dynamically across memory types during the execution of the application. Our preliminary work [8] on this approach showcased that performance of a simple stencil benchmark could be improved by migration, using a scheme similar to out-of-core algorithms, when the compute density of the application kernel is high enough to provide compute/migration overlapping. Further work [9] studied performance models for such strategies, including heuristics to migrate data as part of the execution of a workflow with task dependencies. The prefetching strategy used in this paper matches some of these heuristics. Another study [10] discussed a runtime method to schedule tasks with data dependencies on a deep memory platform. Unfortunately, the scheduling algorithm is limited to scheduling a task only after all its input data has been moved to faster memory.

VI. Conclusion

We presented in this paper how additional memory management abstractions can be used to simplify and extend complex optimization strategies for deep memory and heterogeneous platforms. While we used Intel's Knights Landing architecture as a basis for this study, we expect that exascale topologies will exhibit the same kind of complexity and will require the same kind of optimization strategies. In particular, heterogeneous platforms are ideal for such strategies since a separate compute element can be used for transform operations.

Further tuning of the various abstractions can also be performed, in particular autotuning of the transform operators. As we move closer to exascale, we will also continue to improve the abstractions offered by our framework for future architectures. These improvements will include support for heterogeneous platforms (CPU-GPU with unified memory), as well as better abstractions to handle the distribution of data layouts over multiple NUMA nodes or the use of helper cores to perform data movement, which might be necessary for the Fujitsu A64FX Post-K computer architecture.

The AML library, documentation, and links to the benchmarks are available online at https://argo-aml.readthedocs.io/en/latest/.

Acknowledgments

Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under Contract DE-AC02-06CH11357. This research was supported by the Exascale

Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] A. Sodani, "Knights Landing (KNL): 2nd generation Intel® Xeon Phi processor," in *27th IEEE Hot Chips Symposium (HCS)*, 2015.

[2] R. Lim, Y. Lee, R. Kim, and J. Choi, "An implementation of matrix–matrix multiplication on the Intel KNL processor with AVX-512," *Cluster Computing*, June 2018. [Online]. Available: https://doi.org/10.1007/s10586-018-2810-y

[3] D. Unat, J. Shalf, T. Hoefler, T. Schulthess, A. Dubey, and others (Eds.), "Programming abstractions for data locality," Tech. Rep., 04 2014.

[4] B. Videau, K. Pouget, L. Genovese, T. Deutsch, D. Komatitsch, F. Desprez, and J.-F. Méhaut, "BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications," *International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 28–44, Jan. 2018. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01620778

[5] Intel Corporation, "Memkind: A user extensible heap manager," https://memkind.github.io/, 2018.

[6] "Simplified interface to complex memory," https://github.com/lanl/SICM, 2017.

[7] L. Oden and P. Balaji, "Hexe: A toolkit for heterogeneous memory management," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2017.

[8] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra, and P. Beckman, "Exploring data migration for future deep-memory many-core systems," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2016.

[9] A. Benoit, S. Perarnau, L. Pottier, and Y. Robert, "A performance model to execute workflows on high-bandwidth-memory architectures," in *Proceedings of the 47th International Conference on Parallel Processing, (ICPP)*, 2018.

[10] K. Chandrasekar, X. Ni, and L. V. Kalé, "A memory heterogeneity-aware runtime system for bandwidth-sensitive HPC applications," in *IEEE Int. Parallel and Distributed Processing Symposium Workshops, Orlando, FL, USA*, 2017, pp. 1293–1300. [Online]. Available: https://doi.org/10.1109/IPDPSW.2017.168

[11] A. Haidar, S. Tomov, K. Arturov, M. Guney, S. Story, and J. Dongarra, "LU, QR, and Cholesky factorizations: Programming model, performance analysis and optimization techniques for the Intel Knights Landing Xeon Phi," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2016.

[12] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating small matrix multiplications by runtime code generation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[13] *Intel Math Kernel Library. Reference Manual*, 2018. [Online]. Available: https://software.intel.com/en-us/articles/mkl-reference-manual

[14] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing vector-friendly compact BLAS and LAPACK kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[15] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, 1998.

[16] I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Beyond 16GB: Out-of-core stencil computations," in *Proceedings of the Workshop on Memory Centric Programming for HPC*, 2017.

[17] L. Alvarez, M. Casas, J. Labarta, E. Ayguade, M. Valero, and M. Moreto, "Runtime-guided management of stacked DRAM memories in task parallel programs," in *Proceedings of the 2018 International Conference on Supercomputing (ICS)*, Beijing, China, 2018. [Online]. Available: http://ics2018.ict.ac.cn/essay/ICS18-Paper130.pdf

[18] L. Alvarez, M. Moreto, M. Casas, E. Castillo, X. Martorell, J. Labarta, E. Ayguade, and M. Valero, "Runtime-guided management of scratchpad memories in multicore architectures," in *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT)*, 2015.

[19] C. Rosales, J. Cazes, K. Milfeld, A. Gómez-Iglesias, L. Koesterke, L. Huang, and J. Vienne, "A comparative study of application performance and scalability on the Intel Knights Landing processor," in *ISC Workshops*, 2016.

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

[21] C. Pohl, "Stream processing on high-bandwidth memory," in *Grundlagen von Datenbanken*, 2018.

[22] N. Butcher, S. L. Olivier, J. Berry, S. D. Hammond, and P. M. Kogge, "Optimizing for KNL usage modes when data doesn't fit in MCDRAM," in *International Conference on Parallel Processing*, 2018. [Online]. Available: http://par.nsf.gov/biblio/10064736

[23] (2018) OpenBLAS: An optimized BLAS library. [Online]. Available: https://www.openblas.net/

[24] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, "The design and performance of batched BLAS on modern high-performance computing systems," *Procedia Computer Science*, vol. 108, pp. 495–504, 2017, International Conference on Computational Science (ICCS). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050917307056

[25] R. Asai, "Clustering modes in Knights Landing processors: Developer's guide," Colfax International, Tech. Rep., 05 2016.

[26] A. Vladimirov and R. Asai, "MCDRAM as high-bandwith memory (HBM) in Knights Landing processors: Developer's guide," Colfax International, Tech. Rep., 05 2016.

[27] A. J. Pena and P. Balaji, "Toward the efficient use of multiple explicitly managed memory subsystems," in *IEEE Int. Conf. on Cluster Computing (CLUSTER)*, 2014, pp. 123–131.

[28] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta, "Automating the application data placement in hybrid memory systems," in *IEEE International Conference on Cluster Computing, (CLUSTER)*, 2017, pp. 126–136. [Online]. Available: https://doi.org/10.1109/CLUSTER.2017.50

[29] G. Voskuilen, A. F. Rodrigues, and S. D. Hammond, "Analyzing allocation behavior for multi-level memory," in *Proceedings of the Second International Symposium on Memory Systems, (MEMSYS)*, 2016, pp. 204–207. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989116

[30] NVIDIA, "CUDA: Unified memory programming," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd, 2018.

[31] R. Landaverde, T. Zhang, A. K. Coskun, and M. Herbordt, "An investigation of unified memory access performance in CUDA," in *IEEE High Performance Extreme Computing Conference (HPEC)*, 2014.

[32] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-aware task scheduling on multi-accelerator based platforms," in *IEEE Int. Conf. on Parallel and Distributed Systems*, Dec 2010, pp. 291–298.