H2M: Exploiting Heterogeneous Shared Memory Architectures

Jannis Klinkenberg, Anara Kozhokanova, Christian Terboven, Clément Foyer, Brice Goglin, Emmanuel Jeannot

H2M: Exploiting Heterogeneous Shared Memory Architectures

Jannis Klinkenberg[a,*], Anara Kozhokanova[a], Christian Terboven[a], Clément Foyer[b,c], Brice Goglin[b], Emmanuel Jeannot[b]

[a] *RWTH Aachen University, Chair for Computer Science 12, Seffenter Weg 23, Aachen, 52074, NRW, Germany*
[b] *Inria, Univ. Bordeaux, LaBRI, Talence, France*
[c] *Université de Reims Champagne-Ardenne, LICIIS, LRC DIGIT, Reims, 51097, France*

Abstract

Over the past decades, the performance gap between the memory subsystem and compute capabilities continued to spread. However, scientific applications and simulations show increasing demand for both memory speed and capacity. To tackle these demands, new technologies such as high-bandwidth memory (HBM) or non-volatile memory (NVM) emerged, which are usually combined with classical DRAM. The resulting architecture is a heterogeneous memory system in which no single memory is "best". HBM is smaller but offers higher bandwidth than DRAM, whereas NVM provides larger capacity than DRAM at a reasonable cost and less energy consumption. Despite that, in several cases, DRAM still offers the best latency out of all three technologies.

In order to use different kinds of memory, applications typically have to be modified to a great extent. Consequently, vendor-agnostic solutions are desirable. First, they should offer the functionality to identify kinds of memory, and second, to allocate data on it. In addition, because memory capacities may be limited, decisions about data placement regarding the different memory kinds have to be made. Finally, in making these decisions, changes over time in data that is accessed, and the actual access pattern, should be considered for initial data placement and be respected in data migration at run-time.

In this paper, we introduce a new methodology that aims to provide portable tools and methods for managing data placement in systems with heterogeneous memory. Our approach allows programmers to provide *traits* (hints) for allocations that describe how data is used and accessed. Combined with characteristics of the platforms' memory subsystem, these traits are exploited by heuristics to decide where to place data items. We also discuss methodologies for analyzing and identifying memory access characteristics of existing applications, and for recommending allocation traits.

In our evaluation, we conduct experiments with several kernels and one proxy application on Intel Knights Landing (HBM + DRAM) and Intel Ice Lake with Intel Optane DC Persistent Memory (DRAM + NVM) systems. We demonstrate that our methodology can bridge the performance gap between slow and fast memory by applying heuristics for initial data placement.

Keywords: Heterogeneous Memory, Non-Volatile Memory, High-Bandwidth Memory, Heuristics, HPC, Data Placement

1. Introduction

High-performance computing (HPC) is crucial to advance computational science and engineering. In all modern computing systems, the performance gap between compute and memory continues to spread, particularly in the face of multi-core and accelerated systems. In consequence, the memory subsystem is changing: the evolution of the cache hierarchy is followed by new technologies with new kinds of memory. In the context of HPC, this has been pioneered by combining traditional main memory with a small fraction of high-bandwidth memory (HBM). Moreover, non-volatile memory (NVM) has also emerged. New kinds of memory offer new and different trade-offs between size, speed, and energy consumption, and have to be exploited by the simulation codes.

Thus, the question of how to program such systems and use them efficiently from the application's perspective is becoming of paramount importance. In fact, several research questions need to be addressed. Addressing the system's or middleware's functionality, given a buffer, what is the best memory subsystem where this buffer should be allocated? What are the appropriate abstractions that help the application programmer exploit heterogeneous memory in a portable way? And for both, if the usage of or the access pattern to a chunk of data changes during the application execution and might deliver better performance in a different memory space, what kind of mechanisms could be developed to enable fast and efficient data migration?

Answers to these questions should be given in a portable, vendor- and technology-neutral manner. In this spirit, OpenMP introduced support for memory management by providing allocators to select a target memory kind. While we have shown that this low-level API fits good into OpenMP and provides the necessary functionality to replace technology-specific libraries, the need for a higher-level functionality to enable a productive use of heterogeneous memory in scientific or technical applications became obvious.

*Corresponding Author
Email addresses: j.klinkenberg@itc.rwth-aachen.de (Jannis Klinkenberg), kozhokanova@itc.rwth-aachen.de (Anara Kozhokanova), terboven@itc.rwth-aachen.de (Christian Terboven), clement.foyer@univ-reims.fr (Clément Foyer), brice.goglin@inria.fr (Brice Goglin), Emmanuel.Jeannot@inria.fr (Emmanuel Jeannot)

In this work, we introduce a new methodology that aims to provide portable tools and methods for managing data placement in systems with heterogeneous memory. This manuscript significantly extends prior work [1] that was only presented as a poster in an early phase of the project. Our methodology consists of three components. First, we have developed a library that, given a set of hints (called *traits* in the paper) on the way data items are used and accessed, can efficiently map them to the available memory subsystems. Second, we have designed a monitoring workflow for existing codes or executables that profiles and traces memory allocations and data accesses and, based on that, analyzes the behavior of each data item. Last, we provide a tool that converts these profiles/traces into traits describing how application data items are used and accessed. Via *allocation abstraction* such traits are then incorporated into the application and used by the library to dynamically guide the data allocation at execution time. We have tested our methodology on several use cases and are able to outperform a typical first-come-first-served allocation strategy that is used in most codes.

The paper is organized in the following manner: In Section 2 we give an overview of the existing memory subsystems and platforms and describe how *hwloc* is able to detect available memory kinds. We discuss how to characterize memory in Section 3. We briefly review memory management in OpenMP and show how our approach extends it in Section 4. In Section 5, we present our experimental evaluation. Related work is discussed in Section 6 before we conclude in Section 7.

2. Heterogeneous Memory

Although memory capacity has been steadily increasing over the history of computing, memory bandwidth has not kept pace with processor throughput [2], particularly in the face of growing amounts of parallelism. As a result, the memory subsystems of modern computing devices — socketable processors and accelerators alike — have a deep hierarchy employing a variety of memory technologies [3]. In today's devices, there are traditional main memory (DDR), high-bandwidth memory (HBM) in particular on accelerators but soon also on regular compute nodes, texture memory and scratchpad memory on accelerators, and multiple levels of cache.

In the remainder of this work, we will use the term *memory technology* to refer to a concrete technology and/or product, such as HBM. We use the term *memory kind* to refer to memory with a specific, distinct property, such as the memory with the largest capacity in the system.

We first introduce the current and upcoming heterogeneous memory platforms before describing how they are exposed by the software, and in particular, by the hwloc library.

2.1. Hardware Landscape

Heterogeneous memory first appeared widely in high performance computing in 2016 with the Intel *Knights Landing* Xeon Phi processor (KNL). It embedded 16 GB of MCDRAM with approximatively 400 GB/s of bandwidth (similar to HBM), while still supporting larger off-package DRAM at approximatively 100 GB/s [4]. This MCDRAM could be used as a hardware-managed cache in front of the DRAM, or as an explicit separate NUMA domain requiring the allocator to decide between low-capacity high-bandwidth and large-capacity low-bandwidth memory [5].

KNL is now discontinued and no similar platforms have been used in HPC until recently. However, several CPU vendors announced the return of integrated high-bandwidth memory in future (exascale) platforms. First, the new Intel Xeon *Sapphire Rapids* optionally embeds 64 GB of HBM2e [6]. Combined with the usual off-chip DRAM, its memory subsystem is very similar to KNL. Then, several exascale initiatives are planning to use ARM Neoverse V1 CPUs with both HBM and DRAM. SiPearl's *Rhea* processor will be used in the *European Processor Initiative*. In South Korea, ETRI (*Electronics and Telecommunications Research Institute*) will develop the *K-AB21* CPU. In India, the *Centre for Development of Advanced Computing* (C-DAC) is designing the *Aum* processor. All these developments employ HBM in one form or the other.

A different kind of heterogeneity was brought by the emergence of non-volatile memory (NVM). This new memory technology can be used as persistent storage but also as large-capacity main memory. Then, instead of being the large-capacity (but slow) memory in KNL's architecture, in the presence of NVM, DRAM is considered to be the low-capacity (but fast) memory. This technology is available in Intel Xeon platforms since the Cascade Lake generation as non-volatile memory DIMMs[1]. However, Intel is phasing out this product which was not widely used in HPC yet despite showing interesting benefits for performance and capacity on some applications [7].

System architects at different vendors are also designing hardware protocols to manage memory in a cache-coherent manner so that both CPUs and GPUs may compute using data stored either in host memory or GPU memory. This effectively exposes heterogeneous memory to applications. For instance, a POWER9 platform may interconnect the host and NVIDIA GPUs through the NVLink protocol and expose GPU HBM as additional NUMA nodes. However, that particular HBM has very different performance from the host DRAM, since it is accessed via the NVLink bus [8]. NVIDIA is continuing this road by combining an ARM CPU and a Hopper GPU in the so-called *Grace Hopper* single package, where the CPU LPDDR5 memory and the GPU HBM3 memory are shared in a cache-coherent way [9].

All these platforms expose significant memory performance variances: MCDRAM and HBM have much higher bandwidth than DRAM, but their latency is in some cases higher, especially when accessed through additional buses (e.g., NVLink). NVM has significantly higher latency and lower bandwidth, but offers larger capacity. Hence there is no best target for all needs: Capacity usually decreases when bandwidth increases, and they have no correlation with latency.

[1] Available commercially under the name *Optane DataCenter Persistent Memory Modules.*.

However, the main reason for the heterogeneity of future memory subsystems is that vendors are developing new interconnects with generic support for different kinds of memory. The POWER10 processor will use the *Open Memory Interface* (OMI) which allows connecting DRAM, HBM and NVM [10]. Gen-Z, another memory interconnect, is now part of the *Computer Express Link* (CXL) initiative [11]. CXL is being adopted by many vendors, and it is expected to bring standard support for NVM to many processor models [12]. This variety of technologies and the ability to use cables of different length or even switches will bring much more diverse heterogeneity, since the interconnect topology may impact the end-to-end bandwidth and latency.

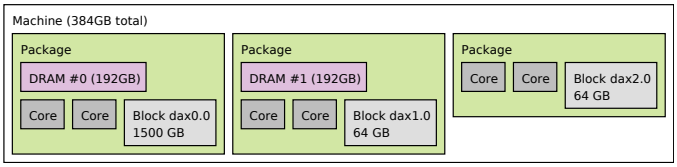
2.2. Memory Kinds in hwloc

hwloc [13] is the de facto standard tool for describing the topology of parallel platforms and managing the locality of tasks and data buffers. Support for heterogeneous memory architectures was added in its latest releases. This section will explain how the topology of heterogeneous memory is exposed by the hardware itself and by hwloc.

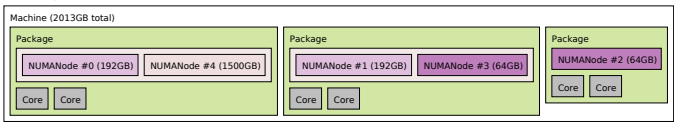
There are currently three main types of memory available on the market: DRAM, HBM and NVM. However, the hardware (E820 table) rather exposes *normal*, *special-purpose* and *non-volatile* memories. The goal of the special-purpose flag is to tell the operating system that HBM and other device-specific memory (GPU or network buffers, for instance) should not be used for default memory allocations (they are exposed as special DAX devices on Linux as shown in Figure 1a). A so-called DAX device represents a memory-like block device for which unnecessary copies in the page cache are avoided. It is then possible for the administrator to expose some special-purpose memory as normal, but the user will not be informed whether it is HBM or anything else (Figure 1b).

This issue of identifying which NUMA nodes are fast or slow, and how, is actually not handled by most research works on heterogeneous memory. These works assume it is known which NUMA node is DRAM or HBM, but this was only true on the Xeon Phi *Knights Landing* platform, where the number of different memory configurations was small and known in advance. Modern platforms can combine multiple sockets with multiple Sub-NUMA configurations comprising HBM and/or DRAM and/or NVM. Being able to precisely identify the kind of each of all these NUMA nodes is now critical to several aspects, including performance portability. This is the reason we extended hwloc to find out whether a NUMA node is HBM, DRAM or NVM[2]. To do so, hwloc combines the hardware knowledge (E820 special purpose flag, etc.) with performance attributes (bandwidths exposed in the ACPI HMAT table), as shown on Figure 1c.

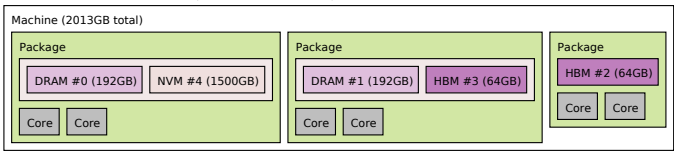
Upcoming platforms with CXL will blur the performance difference between these memory kinds because the CXL fabric may affect some memory performance. The knowledge about



(a) By default, HBMs and NVMs are reserved as DAX special files (`dax0.0` for NVM, `dax1.0` and `dax2.0` for HBMs). Only DRAM memory is exposed as NUMA nodes (#0 and #1).



(b) After `daxctl reconfigure -mode=system-ram`, HBMs and NVMs are exposed as additional NUMA nodes (#2 and #3, and #4).



(c) With hwloc 2.8 heuristics, each NUMA node is described as DRAM, HBM or NVM.

Figure 1: `lstopo` graphical outputs of a fictitious heterogeneous platform with 5 NUMA nodes depending on their configuration on Linux and hwloc heuristics for finding out memory kinds.

NUMA nodes being of HBM, DRAM or NVM will still be useful for debugging or optimization works carried out by expert, but HMAT performance attributes may be used by runtimes to decide where to allocate buffers. hwloc already provides an API to query these attributes, find the best memory target for a given metric, sort them, etc.[14].

We believe that combining static information about memory technologies (HBM, DRAM, NVM) and memory kinds with performance information (bandwidth, latency) is the key to answering the needs of both users and runtimes in terms of deciding where to allocate memory and verifying where it was actually allocated.

3. Characterization of Memory Kinds

Each of the different memory technologies discussed in the previous chapter has distinct performance characteristics (e.g., latency, bandwidth) and semantic characteristics (e.g., access permissions, volatility). The variation in these characteristics can be significant, and addressing them in software is becoming increasingly critical for HPC applications. In consequence, exposing heterogeneous memory to applications and users first requires to actually identify the available memory technologies and their characteristics. Only then, runtimes and applications may be able to actually select the appropriate memory kind matching their needs. The quality of the decision may impact application performance, as for instance memory with the highest bandwidth and lowest latency is limited in size.

3.1. Platforms

In this work, we particularly investigate two heterogeneous memory systems which are also used in the evaluation. The first

[2] Available since hwloc 2.8.

system is an Intel Xeon Phi 7210 codenamed *Knights Landing* (KNL) with 64 cores running at 1.30 GHz base frequency and comprising 192 GB of classical DRAM and 16 GB of MCDRAM. The KNL is configured in *Flat SNC4 mode* meaning that the chip is split into 4 Sub-NUMA clusters, also called quadrants, each having a subset of the available DRAM and MCDRAM, as described in Figure 2. Further, the MCDRAM is *not* used as a cache in front of DRAM but is exposed as separate NUMA domains.

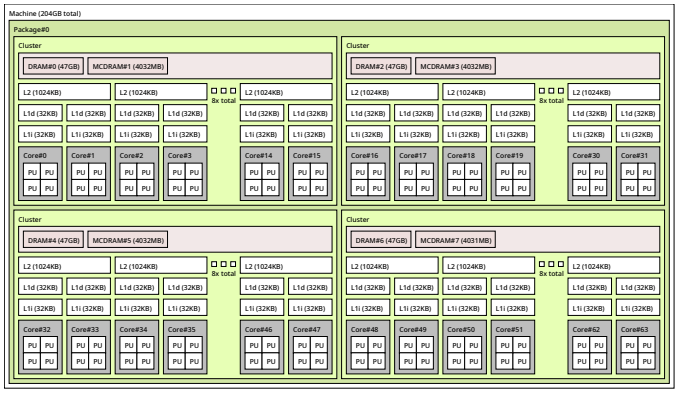


Figure 2: Topology of the KNL platform.

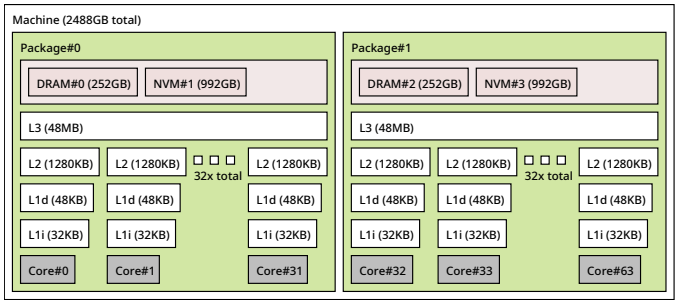


Figure 3: Topology of the Ice Lake platform.

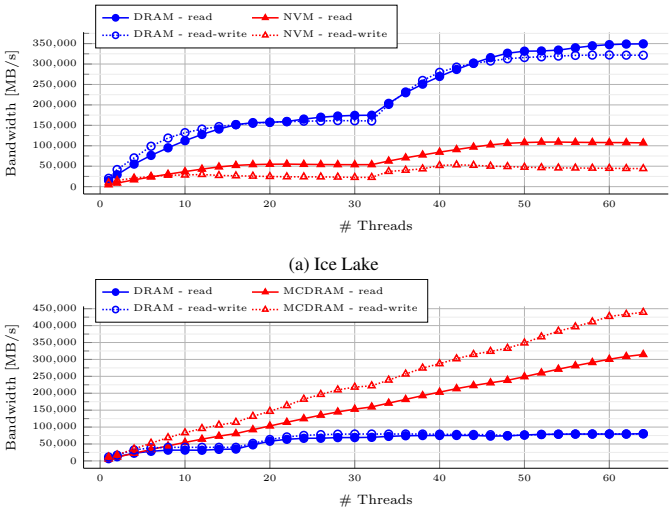
The second system is a dual-socket Intel Xeon Gold 6338 system codenamed *Ice Lake* with a total of 64 cores running at 2.00 GHz base frequency. This system is equipped with 512 GB of DRAM and 2 TB of *Intel Optane DC Persistent Memory* (NVM) which is exposed as a separate NUMA domain per socket, as shown in Figure 3. The per-socket NVM is configured in interleaved mode.

3.2. Bandwidth and Latency

With different kinds of memory present in heterogeneous systems, in most cases there is no "best" memory for all relevant applications, in fact, not even for a single application. Different kinds of memory offer different capacities, and different performance and energy characteristics. As we will show, even a performance comparison becomes complex, as the result which memory is *faster* depends on whether the performance is bound by latency or by bandwidth, by how many threads are being used, and if the capacity is sufficient at all. In selecting the right kind of memory for placing data, optimizing for bandwidth and latency might be conflicting goals.

In consequence, one of the key factors to efficiently manage data in heterogeneous memory is the knowledge about performance characteristics of the underlying memory subsystem. In a first step, we investigate the memory bandwidth scaling of the systems using the *Intel Memory Latency Checker* [15]. Because some memory technologies like NVM do not exhibit uniform performance for reads and writes, we distinguish between read-only and read-write scenarios, which can directly be configured with the Memory Latency Checker tool. Throughout our experiments, each thread was pinned to one physical core.

Figure 4 illustrates the bandwidth scaling behavior on Intel Ice Lake as well as KNL. On Ice Lake, we can observe the clear difference between NVM and DRAM performance, as well as read-only and read-write for NVM. On KNL, the on-chip MCDRAM is outperforming DRAM already at around 8 threads and continues scaling up to about 440 GB/s and 315 GB/s for *read-write* and *read-only* respectively. For DRAM, on the other hand, we can only observe small differences between read-only and read-write. Throughput increases at 16 threads as further quadrants participate in the execution before it stagnates at around 80 GB/s.



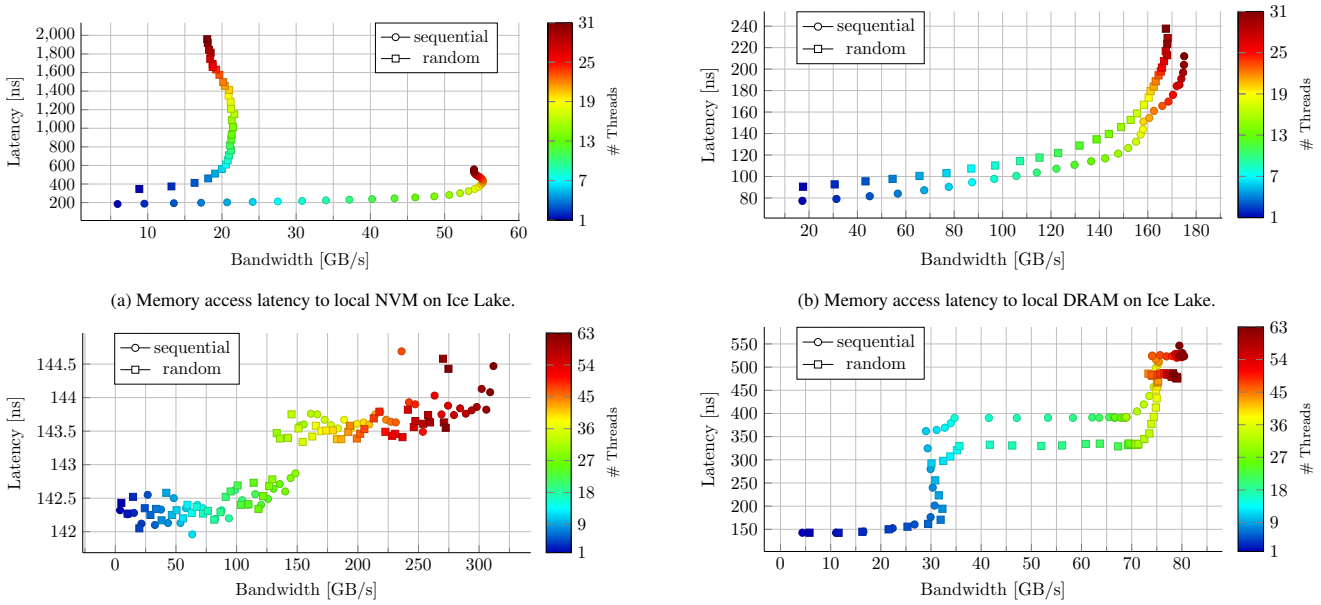
(a) Ice Lake

(b) KNL

Figure 4: Sustainable memory bandwidth on a dual-socket Intel Xeon Ice Lake Optane system and KNL.

In the same way, we measured idle memory latencies between CPU and memory NUMA domains for both systems using the Memory Latency Checker tool. The results are illustrated in Table 1. Previous work [16, 17] already revealed that MCDRAM on KNL shows roughly 20 % higher idle latency than DRAM, which is also reflected in our results. On Ice Lake, the latency to NVM is up to 3.3 × higher compared to DRAM.

Other factors such as memory access patterns and the number of employed cores impact latency too. To estimate the effect of thread scaling on latency with sequential and random memory accesses, we set up the following test. We measured the *loaded latency* on a local socket/quadrant of Ice Lake and KNL systems using read-only workloads with the Memory Latency Checker and present the results in Figure 5. On Ice Lake's



(a) Memory access latency to local NVM on Ice Lake.

(b) Memory access latency to local DRAM on Ice Lake.

(c) Memory access latency to MCRAM on KNL.

(d) Memory access latency to DRAM on KNL.

Figure 5: Loaded latency vs. attainable bandwidth per core on Ice Lake and KNL processors.

	DRAM		MCDRAM	
Intel KNL	local	remote	local	remote
	130 ns	139 ns	155 ns	162 ns
	DRAM		**NVM**	
Intel Ice Lake	local	remote	local	remote
	75 ns	121 ns	250 ns	312 ns

Table 1: Idle memory latencies to local and remote DRAM, MCDRAM and NVM on KNL and the Intel Ice Lake system.

NVM, the difference between sequential and random memory accesses is more pronounced than on other memory types. Figure 5a shows better scaling for both latency and throughput performance with sequential accesses on NVM compared to the random pattern. In the sequential data access case, the peak throughput is achieved with only 20 threads, whereas in the random accesses case, the peak throughput is less than half of the sequential pattern and achieved with 16 threads. Notably, latency is 3 × higher with random accesses compared to sequential. Any additional thread not only contributes to increased latency but also negatively impacts the overall performance in both access patterns. For NVM, this performance degradation is caused by the contention at the integrated memory controller and the on-DIMM buffer of Optane devices [18, 19].

The difference between the two access patterns on other memory domains is minimal. KNL's on-die high-bandwidth MCDRAM memory has smaller latency and higher throughput than DRAM (Figures 5c and 5d), although both memory technologies exhibit increased steps of latency at larger thread-count.

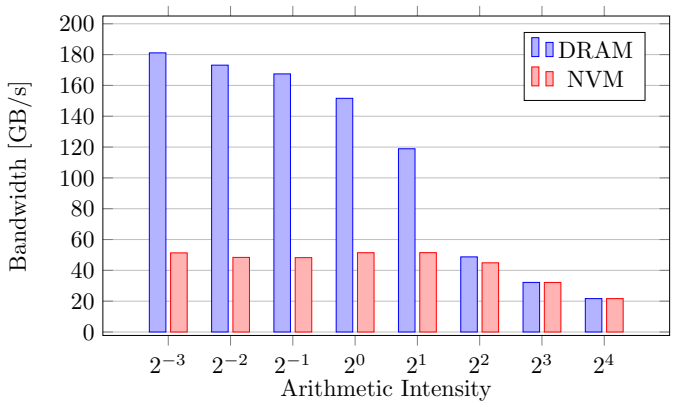
Those detailed insights about bandwidth and latency can be exploited and help to make good placement decisions at execution time.

3.3. Heterogeneous memory power consumption

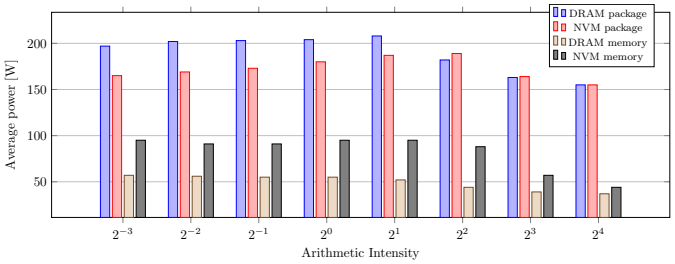
As an additional aspect to consider, a diverse memory landscape entails different energy demands for its components. Evaluation of power consumption as a performance optimization metric is becoming increasingly important. In this section, we provide a comparative overview of the package and memory power consumption with different memory allocations on Ice Lake and KNL systems.

For the assessment of power consumption under memory- and compute-bound constraints, we modified a synthetic STREAM [20] benchmark to include sequential read-only kernels with a range of flops-to-bytes ratios. The `numactl` tool was used to place data on corresponding memory domains and `perf` to measure the power draw of the package and memory subsystems. The benchmark ran on a single socket of Ice Lake CPU and was parallelized using all 32 cores with local DRAM and NVM data allocation. Power measurements of the memory domain include NVM and DRAM memory nodes, therefore we report the total power consumption of both memory nodes per memory allocation.

As already seen in the previous section, the DRAM bandwidth for the memory-bound case outperforms NVM by a factor of three (also in Figure 6a: kernel 2^{-3}). However, the package power consumption of the NVM data placement is 83 % of the DRAM's, while the power draw of the memory domain is higher on NVM by 40 % compared to the DRAM allocation. Both memory- and compute-bound workloads, when allocated



(a) Measured throughput.



(b) Package and memory power consumption.

Figure 6: Bandwidth and power consumption for DRAM and NVM memory on Ice Lake CPU.

on NVM, consume more memory power than DRAM allocations as shown in Figure 6b. The package power consumption, on the other hand, evens out on both memory allocations with compute-bound kernels on NVM memory.

Although not strictly linear, the power dissipation of the NVM memory subsystem is proportional to the attainable throughput, given that the appropriate data locality is maintained. Optane memory modules operate at the 256 bytes granularity which means data accesses below this threshold will result in *read-* or *write-amplification* that leads to an inefficient performance and power draw [7, 18, 19].

Whereas the Intel KNL architecture has been the subject of many studies, which our experiments confirm, there are significantly fewer results in Intel Optane memory. The power consumption of the KNL architecture has been investigated in [21, 22]. They found MCDRAM memory is more energy efficient than DRAM with sequential accesses and exhibits comparably similar efficiency to DRAM in the case of random memory accesses [21]. Another work [22] reports significant power-saving gains under bandwidth-bound workloads.

4. Memory Management and Methodology

Proper support in the parallel programming system is a necessary requirement to exploit the capabilities and parallel performance advantages of heterogeneous memory. Some of our earlier work has addressed the incorporation of allocation support for heterogeneous memory into OpenMP, whereas the focus in this paper lays on our novel methodology to efficiently manage the data placement based on allocation abstraction and traits (Section 4.2).

```
double *a = malloc(N*sizeof(double));
```

Listing 1: Example usage of C's `malloc` function.

4.1. Memory Management in OpenMP

OpenMP has grown from an abstraction of thread-level parallelism to incorporate tasking, SIMD (vector) parallelism, and offloading code regions to remote targets. Many of these features introduce additional (and often implicit) memory allocations, including: private copies of variables allocated for each thread/task/SIMD lane; temporary variables used to implement reductions; and a combination of local and remote variables used to facilitate offloading. With the release of OpenMP 5.0 in 2018, we contributed to the addition of the OpenMP memory management [23] API as a platform- and vendor-neutral interface to allocate and free data in heterogeneous memory.

In iterative programming languages such as C/C++ or Fortran, which are dominant in high-performance computing, memory is managed by the use of allocation API routines, allocator objects, or allocation constructs, respectively. The only argument to the allocation is the amount of memory that is requested to be allocated.

In OpenMP, a *memory space* encapsulates a storage resource that is available in the system. Contemporary HPC server systems are still mainly equipped with DDR-based main memory, which could be the only memory space in a system, but as discussed in the previous section, memories with enhanced performance (e.g., high-bandwidth memory) or functionality (e.g., non-volatile memory) appear as additional memory spaces ever more often. Table 2 lists the pre-defined memory spaces [24]. For each of these spaces, there is also a pre-defined allocator available.

Memory traits define the characteristics of memory spaces. They allow to query, identify and describe the different memory spaces of a system. They are provided by the programmer and expected to be utilized by the runtime system. Memory traits can either be prescriptive, meaning an exact match is required, or descriptive, meaning the runtime is requested to select the optimal type of memory based on the requested properties. An *allocator* is an object that performs allocations of contiguous memory chunks from a given memory space. *Allocator traits* are analogous to memory traits and can be employed to specify different aspects of an allocator's behavior. A set of API routines to create and destroy memory spaces and allocators, taking a set of traits as an argument, is part of OpenMP.

Taking the above concepts into account, the OpenMP memory management API provides the additional functionality to request allocation in a certain memory kind and the specification of certain properties. Users request an allocation of a certain size from a given allocator via the `omp_alloc` routine. The memory must be deallocated with `omp_free`.

With only a minor change over a standard memory allocation as shown in the code snippet in Listing 1, the code snippet

Memory space name	Storage selection intent
omp_default_mem_space	The system default storage.
omp_large_cap_mem_space	Storage with large capacity.
omp_const_mem_space	Storage optimized for variables with constant values.
omp_high_bw_mem_space	Storage with high bandwidth.
omp_low_lat_mem_space	Storage with low latency.

Table 2: Memory spaces in OpenMP.

```
double *a = (double *)
    omp_alloc(N*sizeof(double), omp_high_bw_mem_alloc);
```

Listing 2: Example usage of OpenMP's `omp_alloc` function.

in Listing 2 allocates memory from the memory kind with the highest bandwidth available by using the pre-defined `omp_high_bw_mem_alloc` allocator. However, this comes with the cost of that the resulting program targets a certain memory configuration and requires the programmer to deal with the specific configuration of the memory subsystem of possible target computers, although fallbacks may be specified.

In summary, the selection of a memory space — that is selected kind of memory — and the corresponding allocation of data can be handled by OpenMP in a portable, vendor- and technology-neutral manner. However, application developers usually are neither HPC experts, nor do they target a single, concrete system architecture. We are convinced they need higher-level heuristics, such as describing the layout of the data in memory, that runtimes will translate in low-level metrics, such as selecting the right kind of memory. As applications are not being tuned specifically for a single platform, they need high level abstractions in the form of APIs to cope with the variety of available memory resources. This has to be provided in the form of an API and a corresponding runtime system.

4.2. Memory Management in H2M

Contrary to the OpenMP memory management which is working on an *allocator level*, our approach is based on abstracting single memory allocations to allow programmers to express traits regarding how data is used throughout the application run. Our runtime system combines that information with knowledge about topology and memory characteristics and then takes care of data allocation or movement by applying desired heuristics/strategies that deliver suitable placement decisions for data items as illustrated in Figure 7. Internally, available memory kinds are identified using hwloc as described in Section 2.2. Our runtime provides not only basic default strategies but also an interface to customize strategies and decision making.

Placement decisions. As shown in the overview, our runtime system will generate placement decisions for data items.

A placement decision consists of: (1) A memory space that can be `atv_mem_space_hbw`, `atv_mem_space_low_lat`

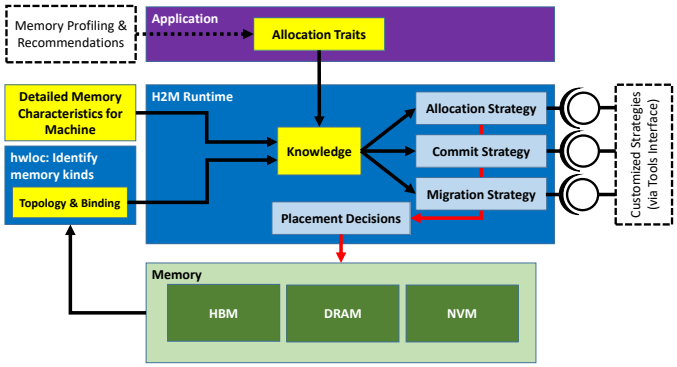


Figure 7: Ovierview of H2M runtime components and interplay with application and environment.

```
int err; double* data_it; size_t N; // size of data item
// Specify allocation traits (requirements and hints)
alloc_trait_t traits[5] = {
    atk_access_mode, atv_access_mode_readwrite,
    atk_access_freq, 2000, /* e.g, accesses per sec */
    atk_access_origin, atv_access_origin_single_thread,
    atk_access_pattern, atv_access_pattern_strided,
    atk_access_stride, 4 /* in bytes */};
// Allocation option 1: direct allocation
data_it = (double*) alloc_w_traits(N, &err, 5, traits);
// Allocation option 2: declaration and commit
declaration_t ld[N_ALLOCS]; /* list of declarations */
ld[0] = declare_w_traits(&data_it, N, &err, 5, traits);
err = commit_allocs(ld, N_ALLOCS); /* commit step */
// Execution phase changed -> update traits and migrate
err = update_traits(data_it, N, 5, traits);
err = apply_migration();
```

Listing 3: Sample trait set and allocation abstraction (shortened names).

or `atv_mem_space_large_cap` that represent memory with the highest bandwidth, lowest latency and largest capacity respectively. (2) Whether the requested memory space is required (`atk_req_mem_space`) or only preferred (`atk_pref_mem_space`). If it is *required* and the allocation request can not be fulfilled it fails and returns `NULL` as well as an error code. In the *preferred* case, it is still possible to fall back to the large capacity memory space. (3) A specification of desired memory alignment and partitioning. If not otherwise specified, the runtime will align to the value of `getpagesize()` and rely on the first-touch principle.

Allocation traits. The allocation trait specified for an allocation can contain prescriptive as well as descriptive key-value pairs. Prescriptive traits directly correspond to elements of the placement decision, e.g. by specifying a required memory space for a particular data item. Descriptive traits can be exploited by *allocation strategies* for decision making. An example for a trait set and allocation abstraction is illustrated in Listing 3.

Allocation options. Basically, there are two options how to allocate data with our runtime system. With the first option, memory is allocated directly, similar to `malloc` but respecting the additional traits that were specified. However, sometimes an application likes to allocate several data items with different access characteristics and needs. As memory capacities are limited, a first-come-first-served approach with direct allocations

might not be the best option as data items with a large performance impact might be handled late and the desired memory space might already be occupied. To tackle this challenge the second option only *declares* each allocation with the corresponding traits and returns a declaration handle. Consequently, a commit step is necessary to finally perform the allocations. A *commit strategy* then decides on the allocation order by respecting the hints provided in traits.

Migrating data. If an application consists of multiple execution phases that access different data items or exhibit vastly different access characteristics for data items, it might be beneficial to migrate data between memory kinds at execution time. Therefore, the API also contains routines to update traits for already allocated data items and to trigger data migration controlled by *migration strategies*. However, this aspect goes beyond the scope of this paper and will be investigated in future work.

4.3. Memory Profiling and Trait Recommendation

As previously mentioned, our approach allows the programmer to specify requirements for allocations and hints how this data will be used and accessed during execution. Nevertheless, the question arises whether a programmer always knows those requirements or has a deep understanding of the underlying access pattern for data items. To assist users we created a workflow to collect and analyze information about memory allocations and memory accesses of existing applications. Those insights can then be leveraged to *recommend* traits for individual data items.

Memory Profiling. There are two fundamental approaches to collect information about data accesses. The first approach tracks memory allocations and instruments individual memory accesses during the compilation step. An example for such an approach is the *AddressSanitizer* [25] in LLVM which is instrumenting memory accesses to detect various memory access errors. Although this technique provides a detailed view, it collects a lot of data and induces high overhead. Further, due to the instrumentation procedure the compiler might leave out some optimizations when creating the executable resulting in a different behavior compared to the uninstrumented executable.

The second approach also tracks memory allocations but uses a sampling technique that periodically interrupts execution and collects information about memory accesses. There are several advantages compared to instrumentation: (1) It can be applied to any existing unmodified executable. Thus, the behavior of the program is unchanged. (2) Although sampling does not yield full details, it can still provide sufficient insights about memory accesses at a reasonable cost without introducing high overhead[3]. (3) Most vendors support *Hardware Performance Counters* (HWC) that count events in hardware such as *Loads* and *Stores*. On recent architectures, these capabilities were extended to additionally provide information which thread accessed a memory location and whether the access has been served from cache or main memory reflecting a realistic execution behavior on the corresponding system. On Intel and AMD platforms, this feature is called *Precise Event Based Sampling* (PEBS) and *Instruction Based Sampling* (IBS) respectively. Due to design choices in hardware this information is unfortunately not available for store events.

In this work we use *NumaMMA* [26], a memory profiler based on PEBS/IBS. NumaMMA allows to execute programs and simultaneously gather information about memory accesses (loads and stores) for individual data items. Collected data is then grouped by callsite representing the code location where the allocation is performed. There might be several separate allocations per callsite. NumaMMA offers scripts to visually report access patterns but also allows dumping the collected data which we employ for recommending traits in the following step. Table 3 lists the relevant information that can be derived from dumps.

Per memory allocation	allocation id start address size allocation timestamp deallocation timestamp callsite address callsite (file and line number)
Per memory access	thread that performed access timestamp allocation id offset accessed access type (load or store) level (cache or main memory) weight (latency) for access

Table 3: Profiling data collected by NumaMMA and provided in dumps.

Trait Recommendation. We can now analyze the NumaMMA dumps to determine a suitable set of traits for individual memory allocations (callsites). This step of the procedure has a high flexibility and can be customized as needed. As an example, a user could identify whether an allocation is accessed by only one or multiple threads and whether it is mostly read or written. Providing a corresponding heuristic or strategy (see Figure 7) that works with these metrics then takes care of guiding data placement at execution time.

In this work, we try to estimate the performance impact of individual allocations by determining their *importance* and *sensitivity* and pass those metrics as traits. We calculate the overall aggregated access latency L_{mem} and the aggregated access latency $L_{mem}(M)$ for each memory allocation M served from main memory as illustrated in Equations 1 and 2. $\sum_{a=1}^{N} Latency(a)$ denotes the aggregated latency of all accesses a in the complete application whereas $\sum_{i=1}^{N} Latency(a, M)$ denotes the aggregated latency of all accesses to memory allocation M. $Level(a)$ and $Level(a, M)$ represent the level from which the corresponding

[3] Adjusting the sampling frequency results in a more fine- or coarse-grained view but also affects the overhead.

memory access has been served from which can either be *cache* or *memory*.

We then determine the corresponding relative share of access latencies to main memory $LS_{mem}(M)$ that represents the *importance* of an allocation as illustrated in Equation 3. A similar approach has also been taken by Wen et al. [27]. During the allocation phase of an application, our strategy will change the order of allocations favoring those with higher $LS_{mem}(M)$. For that, we *declare* single allocations followed by a *commit* step as described in Section 4.2.

$$L_{mem} = \sum_{a=1}^{N} Latency(a), \quad \text{if } Level(a) = memory \tag{1}$$

$$L_{mem}(M) = \sum_{a=1}^{N} Latency(a, M), \quad \text{if } Level(a, M) = memory \tag{2}$$

$$LS_{mem}(M) = \frac{L_{mem}(M)}{L_{mem}} \tag{3}$$

To estimate the best fitting memory space for an allocation, we calculate the ratio of memory accesses $R_{cache}(M)$ that have been served from cache as illustrated in Equation 4 where $N_{cache}(M)$ denotes the number of cache accesses for memory allocation M and $N(M)$ denotes the number of all memory accesses for M.

$$R_{cache}(M) = \frac{N_{cache}(M)}{N(M)} \tag{4}$$

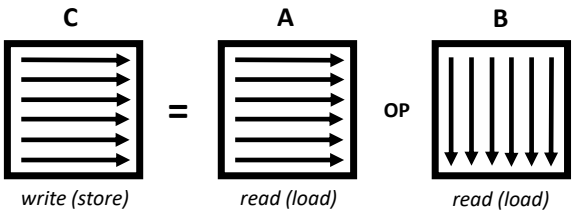
Allocations with a cache ratio $R_{cache}(M) < T_{low}$ are considered to be highly latency sensitive and should be allocated in the memory with the lowest latency. Allocations with a cache ratio $R_{cache}(M) > T_{high}$ can potentially profit most if placed in memory with the highest bandwidth. Values in-between indicate that an allocation can be sensitive to both latency and bandwidth. In this case we also try to place data in highest bandwidth memory and fall back to highest capacity if the request cannot be fulfilled. For our experiments, we empirically set $T_{low} = 0.25$ and $T_{high} = 0.75$.

5. Evaluation

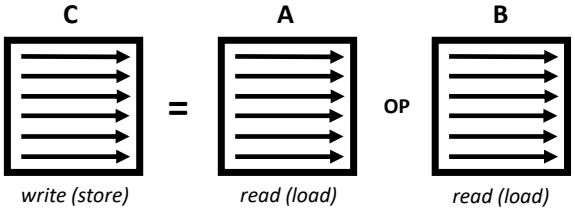
5.1. Kernels and Applications

To evaluate our approach and provide a proof-of-concept, we investigate the following kernels and one proxy application.

Naïve DGEMM Kernel. The first kernel that we analyze is a naïve DGEMM kernel performing $C = A \times B$ where C is used in a writing fashion and A and B are read-only. Memory for each matrix is allocated as a single chunk. Additionally, we compare two different versions of that kernel as illustrated in Figure 8. The first one, denoted as `stridedB`, represents a regular DGEMM and has a linear access pattern (row-major order in C++) on matrix A and C whereas B exhibits a strided access pattern (column-major order in C++). For the second version, denoted as `linearB`, we transpose accesses to B such that it will also be traversed linearly in row-major order. This will provide an alternative view due to the different access pattern combinations and potentially different optimizations applied by the compiler.



(a) `stridedB`: B exhibits a strided access pattern.



(b) `linearB`: B exhibits a linear access pattern.

Figure 8: DGEMM and matrix dot product versions where operation *OP* represents "×" or "·" respectively.

Matrix Dot Product Kernel. In a similar way, we analyze a matrix dot product kernel performing $C = A \cdot B$. Again, there are two different versions where accesses to be B are either strided or linear. Contrary to DGEMM, where we assume that accesses to B in the inner loop represent the bottleneck, here we have element-wise accesses to A, B and C.

XSBench. XSBench [28] has been developed at Argonne National Laboratory (ANL) and Lawrence Livermore National Laboratory (LLNL) and serves as a proxy application for full neutron transport applications like OpenMC [29]. It essentially implements the key computational kernel of the Monte Carlo neutron transport algorithm. XSBench provides several implementations such as an OpenCL and CUDA variant. For this work, we used the multi-threaded variant based on OpenMP. There are three dominant data items that are used and accessed throughout the application: `index_grid`, `nuclide_grid` and `unionized_energy_array`. As `index_grid` is very large and will cover a major share of the memory footprint, we slightly modified the application and split that data item into $N = 20$ equal parts that are allocated separately. As we limit the memory capacity in some experiments and data placement is done on a per allocation basis, this allows to at least allocate a portion of the grid in faster memory if desired.

5.2. Experimental Setup

For our experiments, we use the platforms described in Section 3.1. Our library as well as benchmarks and applications are compiled with the Intel C/C++ Compiler 2019.5.281 and `-O3` optimization. Additionally, we use the latest upstream NumaMMA version including dependencies and hwloc 2.8. To ensure reproducible results of parallel applications runs with OpenMP, we bind threads to physical cores using `OMP_PLACES`

and OMP_PROC_BIND. To focus on the differences between memory technologies and eliminate variations due to NUMA effects, we only use a single socket (Ice Lake) or quadrant (KNL) for our experiments. For all measurements average values of 5 repetitions are reported.

As a trade-off between sufficient execution time and occupied memory we choose the following problem sizes: For DGEMM we use matrix sizes of 4500×4500 on Ice Lake and 3000×3000 on KNL. For the matrix dot product, we use matrix sizes of 30000×30000 on Ice Lake and 12500×12500 on KNL. Input parameters for XSBench are set to "-t <T> -s large -l 30" where <T> represents the number of threads.

5.3. Experiments

Experiment 1: Manual Placement Analysis. First, we investigate how the data placement of single data items in kernels and applications affects overall performance. This will act as reference for the next experiment. Our baseline represents a measurement where all data is allocated in the slower memory of the underlying architecture which is NVM and DRAM for Ice Lake and KNL respectively. With the fundamental assumption that fast memory is limited in capacity, we then perform executions where only a single data item is allocated in faster memory to identify whether the corresponding data item is sensitive to latency or bandwidth or both and how much it affects overall performance. Further, we report a reference version where no capacity limitations are set and all data is allocated in fast memory which, in most cases, is representing a best case scenario or upper bound. All runs have been conducted with different number of threads to reveal the scaling behavior of the different kernels and memory technologies. Results are shown in Figure 9.

For DGEMM on Ice Lake, allocating *A* or *C* in fast memory alone does not result in a significant performance gain. However, we can clearly confirm the hypothesis that *B* is the limiting factor, because it is more frequently accessed in the inner loop, and can benefit from being allocated in faster memory. Especially on Ice Lake, this version gets close to the reference where all data is in DRAM. The linearB version reaches speedups around 3.2 X indicating that it profits from the higher memory bandwidth of the DRAM and better vectorization potential due to the linear access pattern to *A*, *B*, and *C*. The stridedB version only yields speedups up to 1.2 X. Strided accesses to *B* remain the bottleneck but also perform better on NVM indicating that *B* is to some extent also sensitive to latency. Surprisingly, we do not see significant performance differences for both versions on KNL where speedup is ranging from 0.96 X to 1.1 X. One explanation is that the latency difference between DRAM and MCDRAM is much closer compared to the other system, especially in low-to-medium bandwidth consumption scenarios which is the case here. Strided accesses in general seem to hurt performance a lot. Nevertheless, the best result for DGEMM linearB on KNL with 8 threads also only reaches a speedup of around 1.1 X.

For the matrix dot product, we can observe a slightly different behavior. Investigating the linearB version on Ice Lake, allocating *C* in DRAM gives speedups up to 5.9 X. Moving *A* or *B* to DRAM alone does not yield a significant improvement as accesses to the other two matrices will slow down execution. However, when allocating all data in DRAM we can observe a further improvement resulting in up to 10 X speedup. Further, we see that versions with *C* or all data in DRAM scale well with increasing the number of threads. For the stridedB version, we also see improvements up to 4.7 X. Although placing *B* in faster memory improves performance, the dominant data item that has the highest impact on performance seems to remain *C*. On KNL, results look similar for linearB scaling well if *C* or all data is in the faster MCDRAM achieving speedups of up to 1.9 X. Contrary, there is hardly any improvement for stridedB with more than 2 threads, confirming again that workloads with strided accesses do not profit from MCDRAM.

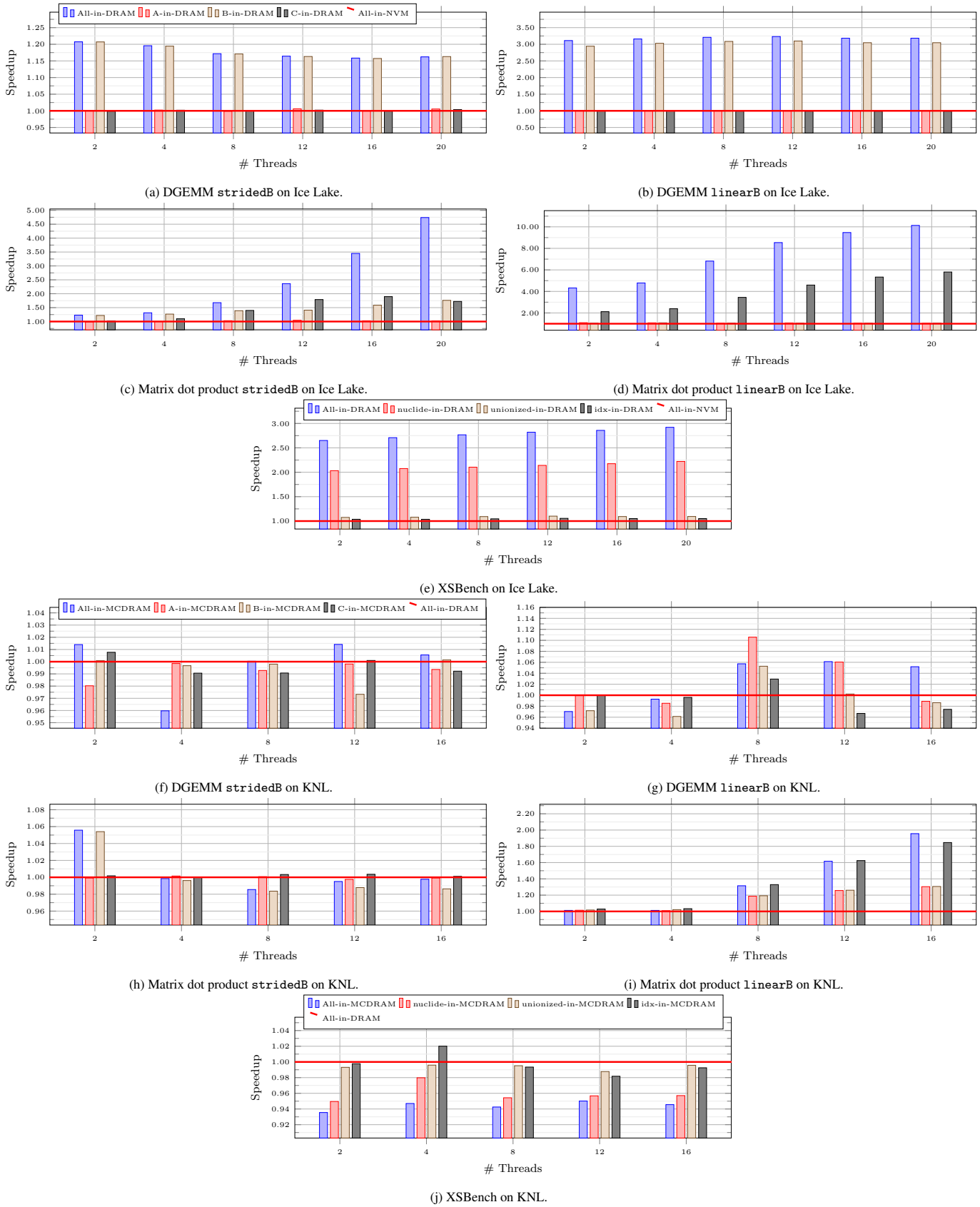
Inspecting results for XSBench on Ice Lake reveals that placing nuclide_grid in the faster DRAM has the highest impact on performance with a speedup of up to 2.8 X as it exhibits a very irregular random access pattern. Allocating unionized_energy_array only improves performance up to 1.1 X whereas index_grid has barely any effect and can stay in slow memory. On KNL, we can actually observe small performance declines using MCDRAM as the critical path of the application is dominated by irregular accesses to nuclide_grid which should stay in DRAM.

Experiment 2: Profiling-Guided Data Placement Heuristics. Insights gained from Experiment 1 help us to understand the impact of placing individual data items in faster memory and enable a comparison against the H2M approach. However, such a brute-force approach is usually not desirable because it would lead to a combinatorial explosion for larger applications with lots of different data items.

Another challenge arises as memory capacities are limited. Keeping all data in fast memory is often not possible for real codes. In most applications, data is allocated on a first-come-first-served basis which could complicate or even prevent placing data items with the highest performance impact into fast memory if that has already been occupied by previous allocations. As an example, DGEMM strideB normally allocates and initializes *A*, *B* and *C* in that order. If the usual preferred faster memory is limited to the size of a single matrix, this might result in a situation where *A* resides in fast memory and *B*, which has a higher impact, needs to fall back to slow memory. We will show how our *commit* approach mitigates this problem.

In this experiment, we apply the methodology explained in Section 4.3 to the codes described in Section 5.1. First, we profile application runs using NumaMMA. Based on the resulting data, we determine the *importance* and *desired memory space* for each data item. Via allocation abstraction and declaration we provide these traits to our runtime system. In a commit step, our *commit strategy*, denoted as *H2M prio*, will then decide about the actual allocation order and where data items are allocated leveraging the specified traits. Finally, we compare the strategy against a first-come-first-served approach, denoted as *first-served*, and the manual placements from Experiment 1.

As a case study, we discuss the process for XSBench. We apply NumaMMA to XSBench on Ice Lake using 16 threads and



(a) DGEMM `stridedB` on Ice Lake.

(b) DGEMM `linearB` on Ice Lake.

(c) Matrix dot product `stridedB` on Ice Lake.

(d) Matrix dot product `linearB` on Ice Lake.

(e) XSBench on Ice Lake.

(f) DGEMM `stridedB` on KNL.

(g) DGEMM `linearB` on KNL.

(h) Matrix dot product `stridedB` on KNL.

(i) Matrix dot product `linearB` on KNL.

(j) XSBench on KNL.

Figure 9: Speedup of manual data placement analysis for matrix kernels and XSBench on Ice Lake and KNL.

use the dumps to calculate metrics described in Equations 1 to 4 as well as generate trait recommendations. Table 4 displays the results for the three dominant data items (callsites) in XSBench. We can see that `nuclide_grid` has the highest importance factor and has a poor cache hit ratio due to an irregular access pattern. Therefore, this allocation should have the highest priority and should be placed in the memory space with the lowest latency. Both, `index_grid` and `unionized_energy_array` have a good cache hit rate of more that 92%. They may benefit from highest bandwidth if available, but this is not as critical as for `nuclide_grid`. Hence, they should be allocated later. These traits are then used to create a code version with allocation abstractions.

Finally, we conduct tests using our commit strategy *H2M prio* and compare it against *first-served* and the manual item placements. We use 16 threads for both, Ice Lake and KNL. The overall memory footprint of the XSBench execution is approximately 6 GB. To demonstrate the effectiveness of our data placement, we enforce capacity limitations for the higher bandwidth memory to 500 MB and 1 GB. Results are illustrated in Figures 10e and 10j.

On Ice Lake, we can see that *first-served* fails to place `nuclide_grid` in faster DRAM due to the existing order of allocations. Even if the limit is set to 1 GB most of the space is occupied by `index_grid`. *H2M prio* however respects the specified traits and dynamically prioritizes `nuclide_grid` that has a higher $LS_{mem}(M)$. It is clearly outperforming the other versions almost reaching the best case performance while only using 500 MB of the fast memory. On KNL, there is almost no speedup possible as the critical path is dominated by the irregular access pattern. In that case, our strategy can also not improve performance but will also not induce any performance declines.

We applied the same procedure to DGEMM and matrix dot product kernels. However, for those cases capacity limitations have been set that only one or two of the matrices fit into fast memory. The remaining results are also visualized in Figure 10.

For DGEMM kernels on Ice Lake, it can be observed that *first-served* using a capacity limit of 1 matrix fails to allocate *B* in fast memory and thus reaches similar results than *A-in-DRAM*. *H2M prio* directly prioritizes *B* and yields close to best case performance. On KNL, the picture is similar to XSBench: strategies do not result in improvement but also not in a performance decline. Same is also true for matrix dot product `stridedB`.

Investigating the matrix dot product, *H2M prio* clearly outperforms *first-served* on Ice Lake for both capacity limitations. For `stridedB` and a limit of 2 matrices, it essentially places *B* and *C* in faster memory almost reaching the *All-in-DRAM* reference. For `linearB`, *first-served* will not place the important *C* in fast memory. On KNL with `linearB`, *H2M prio* again outperforms other versions by directly focusing on data items with most impact.

6. Related Work

Contemporary systems support the simultaneous execution of multiple processes; naturally this requires arbitration of the system's resources, which is done at an operating system or driver level that carries some elevated privilege. It protects, allocates, and shares system memory among multiple executing processes by means of virtual memory. *Virtual memory* entails the separation of physical memory addresses from logical ones exposed to software, with hardware support present in almost all contemporary processors. Virtual memory is almost always implemented by dividing memory into pages — contiguous ranges of memory that embody the physical/logical mapping. Whereas the default has been 4 k pages, applications may benefit from larger page sizes and modern systems support 2 M and 1 G pages.

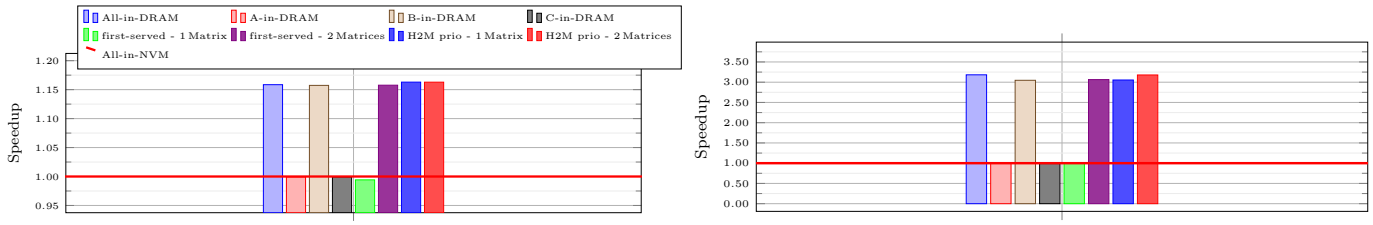
For accelerators like GPUs, a device driver typically handles these same issues, although the specialized nature of these components allows some simplification compared to general-purpose operating systems.

Processors may be able to access some memories (e.g. those which are physically closer) faster than others; this arrangement is referred to as Non-Uniform Memory Access (*NUMA*) [30]. Many NUMA systems strive to transparently deliver functionality and performance to software, but nearly all can be more effectively used with some level of application awareness.

Many *programming languages* require the user to perform memory management with facilities provided by the language: C++'s `new`/`delete` operators and Fortran 90's `ALLOCATE`/`DEALLOCATE` statements are familiar examples of this. Notably, C's `malloc` function is not part of the language itself, but provided by the Standard Library. These built-in memory facilities were designed decades ago and are not prepared to express the variety of configurations possible today.

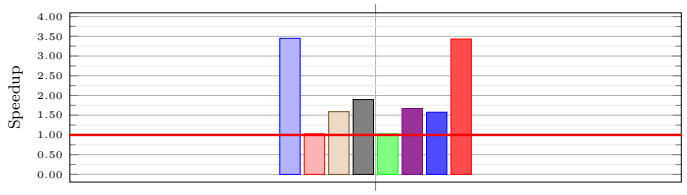
There have been many efforts to produce *software libraries* to expose functionality and provide specialized behavior for memory systems. `libnuma` [31] provides a mechanism for querying memory locations (NUMA nodes) and for specifying user-level preferences about page placement. However it only manipulates NUMA nodes without knowing or taking into account their characteristics (performance, capacity, etc.). This is only the low-level interface to memory binding system calls on Linux. The *memkind* library [32] builds upon NUMA node functionality to enable allocations of high-bandwidth memory within C code. It was first designed for KNL where the characteristics of memory nodes were known in advance and could be hardwired in software. It was later extended to discovering NVM nodes. However, now it relies on our work in hwloc for identifying generic HBM nodes in a portable manner. SICM [33] rather performs an architecture profiling to find out which memory nodes are DRAM, HBM or NVM. However this step is difficult because it must detect bandwidth and latency differences, as well as read/write asymmetry. We believe that hwloc's reading performance attributes from the HMAT ACPI table is a more reliable way to identify and characterize the different kinds of nodes.

As explained in Section 2.2, this issue of identifying which

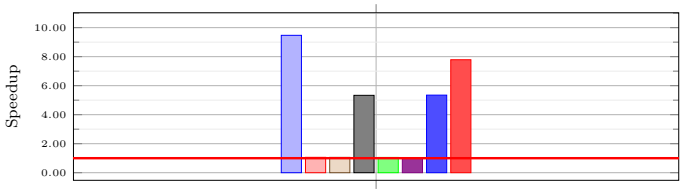


(a) DGEMM `stridedB` on Ice Lake.

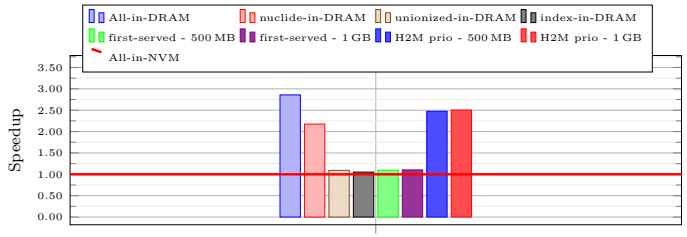
(b) DGEMM `linearB` on Ice Lake.



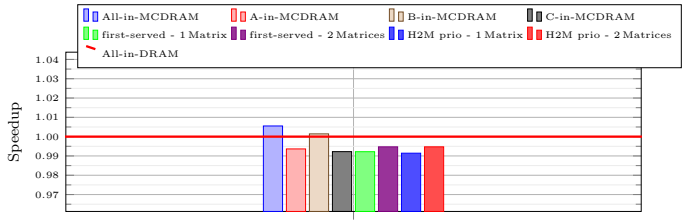
(c) Matrix dot product `stridedB` on Ice Lake.



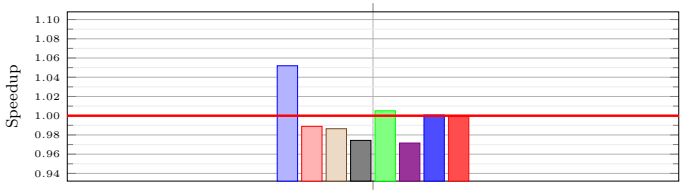
(d) Matrix dot product `linearB` on Ice Lake.



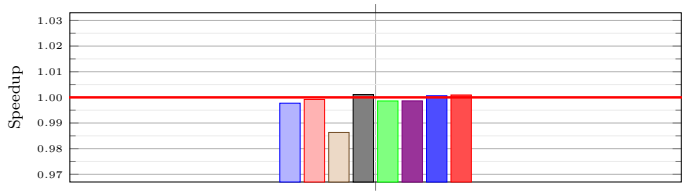
(e) XSBench on Ice Lake.



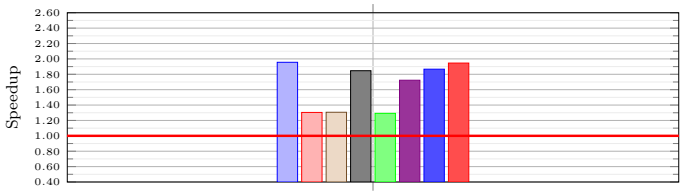
(f) DGEMM `stridedB` on KNL.



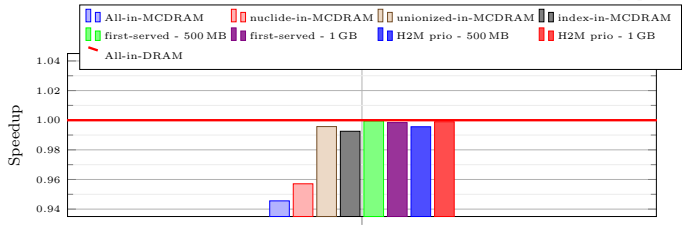
(g) DGEMM `linearB` on KNL.



(h) Matrix dot product `stridedB` on KNL.



(i) Matrix dot product `linearB` on KNL.



(j) XSBench on KNL.

Figure 10: Comparison of H2M commit strategies on Ice Lake and KNL. Results are based on executions with 16 threads.

Callsite	# Allocations	Size [MB]	$R_{cache}(M)$	$LS_{mem}(M)$
index_grid	20	5697.84	0.9215	0.0049
nuclide_grid	1	192.60	0.0762	0.5720
unionized_energy_array	1	32.10	0.9225	0.0055

Table 4: Trait metrics calculated based on NumaMMA profiling data for XSBench using 16 threads.

NUMA node is fast or slow is silently ignored by most research works on heterogeneous memory. They assume the platform details are known and rather focus on optimizing memory placement by intercepting or replacing allocation calls. Servat et al. [34] use Extrae to track memory allocations and PEBS to sample load and store instructions. These traces are analyzed to find out which memory objects deserve allocation in the high bandwidth memory on KNL. Jordà et al. [35] also rely on Extrae and PEBS for tracking memory accesses. Based on a post-mortem optimization and allocation interception they prescribe initial data placement for data objects on a Xeon platform with Optane NVM. Narayan et al. [36] perform a post-mortem analysis of memory accesses based on cache misses and reorder-buffer stall times to classify objects for generic heterogeneous memory platforms. There are other works that make use of PEBS [37, 38, 27] or other profiling methods [39] to come up with a good data placement, either post-mortem or online.

Our approach leverages some of these ideas into traits that may be either guessed by the runtime or provided by the user. We believe the application developers often know which memory buffers may be bandwidth- or latency-critical, hence they may be able to help the runtime taking placement decision. Otherwise, the existing profiling strategies may be used as a fallback to recommend traits.

7. Conclusion & Future Work

In modern multi-/many-core systems with heterogeneous memory, effective utilization of all kinds of memory at different levels in the memory hierarchy has become even more crucial for achieving best performance and power efficiency. After reviewing the current hardware landscape, we have presented how hwloc detects different kinds of memory and how it makes the topology available to middleware and applications.

We characterized regular DRAM, high-bandwidth (HBM) and large-capacity (NVM) memory in Intel KNL and Ice Lake systems. Then we have shown, that with different kinds of memory present in heterogeneous systems, in most cases there is no ”best” memory for all relevant applications. In selecting the right kind of memory for placing data, optimizing for bandwidth and latency might be conflicting goals.

With *H2M*. we introduced a new portable, vendor- and technology-neutral methodology that provides portable methods and tools for managing data placement in systems with heterogeneous memory. The programmer is enabled to express requirements or hints (called traits in the paper) on the way in which data is used and accessed. Our library can efficiently map allocations extended with these hints to the available memory subsystems and can even perform migrations at run-time. In addition, our monitoring workflow supports the programmer in choosing traits for individual data items by automatically generating memory access profiles and analyzing the behavior of each data item to propose requirements or hints.

Our methodology has been evaluated on Intel KNL and Ice Lake systems with four kernels and one proxy application. We demonstrated the effectiveness of our approach that outperforms a first-come-first-served allocation scheme in nearly all cases and reaches close to best case performance while only using a part of the available fast memory. Finally, for cases on KNL that do not profit from MCDRAM, we showed that our approach also does not result in any performance declines.

Future Work. Based on the success of our approach, we are looking at leveraging our traits in the OpenMP standard to extend its memory allocation API and to make our work simpler to use by application programmers. Although we believe that it is beneficial to enable the programmer to express requirements and hints via allocation abstraction, we also investigate approaches based on allocation interception to avoid code modifications.

Most of related work is focusing on initial data placement, where data is allocated in a certain memory and resides there for the complete application lifetime. As larger applications usually consist of multiple execution phases that exhibit vastly different access characteristics, we plan to provide solutions that combine initial data placement as well as data movement at run-time. Consequently, our H2M API also provides memory migration abilities that will require more complex heuristics to tradeoff data movement overhead with achievable performance improvements.

Finally, we plan to investigate upcoming architectures comprising DRAM and HBM as described in Section 2.1, and a larger set of applications.

Acknowledgments

This work was supported in part by the French National Research Agency (ANR) in the frame of the ANR-DFG H2M project (ANR-20-CE92-0022-01).

Parts of this work have been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 446185093.

We would like to thank François Trahay for extending the NumaMMA software to better suit the requirements of our work.

References

[1] C. Foyer, B. Goglin, E. Jeannot, J. Klinkenberg, A. Kozhokanova, C. Terboven, H2M: Towards Heuristics for Heterogeneous Memory, in: 2022 IEEE International Conference on Cluster Computing (CLUSTER), 2022, pp. 498–499. doi:10.1109/CLUSTER51413.2022.00060.

[2] S. Borkar, A. A. Chien, The Future of Microprocessors, Commun. ACM 54 (5) (2011) 67–77. doi:10.1145/1941487.1941507.

[3] R. Smith, Hot Chips 2016: Memory Vendors Discuss Ideas for Future Memory Tech—DDR5, Cheap HBM, & More (Aug. 2016).
URL http://www.anandtech.com/show/10589/hot-chips-2016-memory-vendors-discuss-ideas-for-future-memory-tech-ddr5-cheap-hbm-more

[4] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y. Liu, Knights Landing: Second-Generation Intel Xeon Phi Product, IEEE Micro 36 (2) (2016) 34–46. doi:10.1109/MM.2016.25.

[5] T. Barnes, B. Cook, J. Deslippe, D. Doerfler, B. Friesen, Y. He, T. Kurth, T. Koskela, M. Lobet, T. Malas, L. Oliker, A. Ovsyannikov, A. Sarje, J. Vay, H. Vincenti, S. Williams, P. Carrier, N. Wichmann, M. Wagner, P. Kent, C. Kerr, J. Dennis, Evaluating and Optimizing the NERSC Workload on Knights Landing, in: 2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 2016, pp. 43–53. doi:10.1109/PMBS.2016.010.

[6] A. Biswas, Sapphire Rapids, in: 2021 IEEE Hot Chips 33 Symposium (HCS), IEEE Computer Society, 2021, pp. 1–22.

[7] I. B. Peng, M. B. Gokhale, E. W. Green, System Evaluation of the Intel Optane Byte-Addressable NVM, in: Proceedings of the International Symposium on Memory Systems, MEMSYS '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 304–315. doi:10.1145/3357526.3357568.

[8] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, K. J. Barker, Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect, IEEE Transactions on Parallel and Distributed Systems 31 (1) (2020) 94–110. doi:10.1109/TPDS.2019.2928289.

[9] A. C. Elster, T. A. Haugdahl, Nvidia Hopper GPU and Grace CPU Highlights, Computing in Science & Engineering 24 (2) (2022) 95–100. doi:10.1109/MCSE.2022.3163817.

[10] T. M. Coughlin, J. Handy, Higher Performance and Capacity with OMI Near Memory, in: 2021 IEEE Symposium on High-Performance Interconnects (HOTI), 2021, pp. 68–71. doi:10.1109/HOTI52880.2021.00023.

[11] M. Feldman, A. Norton, E. Joseph, CXL and Gen-Z Consortiums Combine Forces, Hyperion Research (Jul. 2020).

[12] D. Das Sharma, S. Tavallaei, Compute Express Link 2.0 White Paper (Nov. 2020).

[13] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, R. Namyst, hwloc: A generic framework for managing hardware affinities in HPC applications, Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010 (2010) 180–186doi:10.1109/PDP.2010.67.

[14] B. Goglin, A. Rubio Proaño, Using Performance Attributes for Managing Heterogeneous Memory in HPC Applications, in: The 23rd IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2022), held in conjunction with IPDPS 2022, IEEE, Lyon, France, 2022, pp. 890–899. doi:10.1109/IPDPSW55747.2022.00145.
URL http://hal.inria.fr/hal-03599360

[15] Intel, Memory Latency Checker (MLC), accessed: 2023-02-14 (2013).
URL http://www.intel.com/software/mlc

[16] S. Salehian, Y. Yan, Evaluation of Knight Landing High Bandwidth Memory for HPC Workloads, in: Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms, IA3'17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–4. doi:10.1145/3149704.3149766.

[17] I. B. Peng, R. Gioiosa, G. Kestor, J. S. Vetter, P. Cicotti, E. Laure, S. Markidis, Characterizing the performance benefit of hybrid memory system for HPC applications, Parallel Computing 76 (2018) 57–69. doi:10.1016/j.parco.2018.04.007.

[18] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, S. Swanson, An Empirical Guide to the Behavior and Use of Scalable Persistent Memory, in: Proceedings of the 18th USENIX Conference on File and Storage Technologies, FAST'20, USENIX Association, USA, 2020, pp. 169–182.

[19] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, J. Zhao, Characterizing and Modeling Non-Volatile Memory Systems, in: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 496–508. doi:10.1109/MICRO50266.2020.00049.

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

[21] T. Allen, C. S. Daley, D. Doerfler, B. Austin, N. J. Wright, Performance and energy usage of workloads on KNL and haswell architectures, in: High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 8th International Workshop, PMBS 2017, Denver, CO, USA, November 13, 2017, Proceedings, Springer, 2017, pp. 236–249.

[22] A. Haidar, H. Jagode, P. Vaccaro, A. YarKhan, S. Tomov, J. Dongarra, Investigating power capping toward energy-efficient scientific applications, Concurrency and Computation: Practice and Experience 31 (6) (2019) 1–14, e4485 cpe.4485. doi:10.1002/cpe.4485.

[23] J. Sewall, S. J. Pennycook, A. Duran, C. Terboven, X. Tian, R. Narayanaswamy, Developments in memory management in OpenMP, IJHPCN 13 (1) (2019) 70–85.

[24] OpenMP Architecture Review Board, OpenMP Application Programming Interface, Version 5.2 (Nov. 2021).
URL https://https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[25] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov, AddressSanitizer: A fast address sanity checker, in: 2012 USENIX Annual Technical Conference (USENIX ATC 12), USENIX Association, Boston, MA, 2012, pp. 309–318.
URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[26] F. Trahay, M. Selva, L. Morel, K. Marquet, NumaMMA: NUMA MeMory Analyzer, in: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 1–10. doi:10.1145/3225058.3225094.

[27] S. Wen, L. Cherkasova, F. X. Lin, X. Liu, ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems, in: Proceedings of the 2018 International Conference on Supercomputing, ICS '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 263–273. doi:10.1145/3205289.3205320.

[28] J. R. Tramm, A. R. Siegel, T. Islam, M. Schulz, XSBench – the development and verification of a performance abstraction for Monte Carlo reactor analysis, in: PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future, Kyoto, 2014, pp. 1–12.
URL https://www.mcs.anl.gov/papers/P5064-0114.pdf

[29] P. K. Romano, B. Forget, The OpenMC Monte Carlo particle transport code, Annals of Nuclear Energy 51 (2013) 274–281. doi:10.1016/j.anucene.2012.06.040.

[30] A. Cox, R. Fowler, The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with Platinum, in: Proceedings of the Twelfth ACM Symposium on Operating Systems Principles, SOSP '89, ACM, New York, NY, USA, 1989, pp. 32–44. doi:10.1145/74850.74855.

[31] A. Kleen, A NUMA API for Linux, Novel Inc (2005).

[32] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, S. D. Hammond, memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies, Tech. rep., Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States) (2015).

[33] M. K. Lang, Simplified Interface to Complex Memory (SICM) FY19 Project Review (Oct. 2019). doi:10.2172/1569724.

[34] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H.-C. Hoppe, J. Labarta, Automating the Application Data Placement in Hybrid Memory Systems, in: Proceedings of the IEEE International Conference on Cluster Computing, Hawaii, USA, 2017, pp. 126–136. doi:10.1109/CLUSTER.2017.50.

[35] M. Jordà, S. Rai, E. Ayguadé, J. Labarta, A. J. Peña, ecoHMEM: Improving Object Placement Methodology for Hybrid Memory Systems in HPC, in: 2022 IEEE International Conference on Cluster Computing (CLUSTER), 2022, pp. 278–288. doi:10.1109/CLUSTER51413.

2022.00040.

[36] A. Narayan, T. Zhang, S. Aga, S. Narayanasamy, A. K. Coskun, MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems, in: Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium, IEEE, Vancouver, BC, Canada, 2018, pp. 326–335. doi:10.1109/IPDPS.2018.00042.

[37] K. Wu, Y. Huang, D. Li, Unimem: Runtime data managementon non-volatile memory-based heterogeneous main memory, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 1–14. doi:10.1145/3126908.3126923.

[38] K. Wu, J. Ren, D. Li, Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, IEEE Press, 2019, pp. 1–13. doi:10.1109/SC.2018.00034.

[39] M. Laghari, N. Ahmad, D. Unat, Phase-Based Data Placement Scheme for Heterogeneous Memory Systems, in: 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2018, pp. 189–196. doi:10.1109/CAHPC.2018.8645903.