HEXE: A TOOLKIT FOR HETEROGENEOUS MEMORY MANAGEMENT

Lena Oden, Jülich Supercomputing Center, l.oden@fz-juelich.de
Pavan Balaji, Argonne National Laboratory, balaji@anl.gov

***Abstract*—Heterogeneity in memory is becoming increasingly common in high-end computing. Several modern supercomputers, such as those based on the Intel Knights Landing or NVIDIA P100 GPU architectures, already showcase multiple memory domains that are directly accessible by user applications, including on-chip high-bandwidth memory and off-chip traditional DDR memory. The next generation of supercomputers is expected to take this architectural trend one step further by including NVRAM as an additional byte-addressable memory option. Despite these trends, allocating and managing such memory are still tedious tasks. In this paper, we present `hexe`, a highly flexible and portable memory allocation toolkit. Unlike other memory allocation tools such as `malloc`, `memkind`, and `cudaMallocManaged`, `hexe` presents a rich and portable memory allocation framework that allows applications to carefully and precisely manage their memory across the various memory subsystems available on the system. Together with a detailed description of the design and capabilities of `hexe`, we present several case studies where the flexible memory allocation in `hexe` allows applications to achieve superior performance compared with that of other memory allocation tools.**

I. INTRODUCTION

As we continue to build larger and more power-efficient supercomputing systems, heterogeneity in computational units and in memory is becoming an important characteristic. Although processor heterogeneity has received a lot of attention, the shift in architecture is not isolated to processors alone. The memory architecture is similarly increasing in complexity, with the aim of improving the overall bandwidth and reducing power consumption. As we move to exascale, we will observe a dramatic change in the memory architecture leading to multilevel memory hierarchies. Heterogeneous multilevel memory hierarchies with slow and fast memories suitable for regular general-purpose cores and accelerator cores, scratchpad addressable caches, and nonvolatile memories are already becoming increasingly common and can also be seen on the future supercomputing system roadmaps of several key vendors.

Systems with heterogeneous memory systems exist today. Recently Intel released the Knights Landing (KNL) platform with two types of memory: up to 16 GB of on-chip high-bandwidth memory and an additional off-package DDR4 memory (up to 384 GB). NVIDIA's P100 architecture allows graphics processing units to host directly accessible on-chip high-bandwidth memory together with traditional host DDR4 memory.

Despite these trends, allocating and managing such memory are still tedious tasks. Most traditional memory allocation techniques, such as `malloc`, `calloc`, and `mmap`, ignore memory heterogeneity and default to only one type of memory. Recently, `memkind` [2] was introduced as an alternative memory allocation framework that is aware of heterogeneity in memory on Intel KNL platforms; and NVIDIA's CUDA model provides a number of memory allocation calls that allow buffers to be allocated on GPU systems. Although such tools are indeed a step up from traditional memory allocation tools, however, they suffer from two primary shortcomings.

1) **Portability:** Existing tools are either unaware of heterogeneous memory (e.g., `malloc`) or are specific to a particular architecture (e.g., `memkind` for Intel Xeon Phi or `cudaMallocManaged` for NVIDIA GPUs). Attempting to use these on other configurations causes erroneous behavior, making the application responsible for detecting the appropriate hardware configuration before using these tools.
2) **Flexibility:** While tools such as `memkind` and `cudaMallocManaged` allow common memory allocation patterns, they are inflexible in more complex usage models that require data to be shared across multiple classes of memory. This inflexibility can hurt performance or limit the problem sizes that can be executed for some applications.

To address these shortcomings, we present `hexe`, a highly flexible and portable memory allocation toolkit. Like `memkind` and `cudaMallocManaged`, `hexe` is aware of the availability of heterogeneous memory in the system. Unlike `memkind` and `cudaMallocManaged`, however, `hexe` presents a feature-rich and portable memory-allocation framework. The higher degree of flexibility that is offered by `hexe` allows applications to precisely manage their memory across the various memory subsystems available on the system and to allocate their data objects on the memory that is most appropriate for their access patterns. This ability results in superior performance for `hexe` compared with other memory allocation tools.

Together with a detailed description of the design and the capabilities of `hexe`, we present several case studies showcasing `hexe`'s flexibility in memory allocation. We evaluate the performance of `hexe` with several microbenchmarks and application kernels and compare it with `malloc`, `memkind`, and `cudaMallocManaged`.

Terminology: To be consistent, in this paper, we will refer to on-chip high-bandwidth memory as "HBM" for both Intel KNL and NVIDIA GPU platforms. We will refer to traditional DDR memory such as the off-chip memory on KNL and the host memory on GPU platforms as "DRAM" in this paper.

II. BACKGROUND

In this section we present a short overview of the Intel Xeon Phi (focusing on Knights Landing) and NVIDIA GPU (focusing on P100) architectures. We focus primarily on the memory subsystems of these architectures rather than the processing elements because of their higher relevance to our research in this paper.

A. Intel Xeon Phi: Knights Landing

The Intel Knights Landing [4] is the second generation of the Intel Xeon Phi many-core architecture. In contrast to the first generation, Knights Corner, KNL comes in a self-booting version (i.e., does not need a "host" Xeon processor to drive it) and is byte-compatible with mainline Intel Xeon processors.

Current KNL versions support up to 16 GB HBM, called multichannel DRAM (MCDRAM), together with up to 384 GB of traditional DRAM. Each KNL chip can support eight HBM devices, each connected through its own embedded DRAM controller (EDC). Every HBM node has a separate read and write bus connecting it to its EDC. The DRAM is connected by using two DDR4 memory controllers on the opposite sides of each chip.

a) KNL Memory Modes: KNL provides three operation *modes* in which the memory can be organized: *flat mode*, *cache mode*, and *hybrid mode*. Flat mode allows for the HBM to be directly accessible. Cache mode treats it as an additional level of cache. Hybrid mode allows a part of the HBM to be treated in flat mode and the rest to be treated in cache mode.

b) KNL Cluster Modes: KNL also provides three decomposition or isolation modes for its memory (both HBM and DRAM), referred to as *cluster modes*. The goal of these modes is to keep on-chip communication as low as possible. The modes are the *all-to-all*, *quadrant/hemisphere,* and *sub-NUMA clustering (SNC)*. The SNC mode is further split into *SNC-2* and *SNC-4*, depending on whether the decomposition of memory is into two halves or four quarters. The all-to-all and quadrant/hemisphere modes expose the available memory as one large chunk of high-bandwidth memory, while the SNC modes expose them as multiple NUMA domains.

B. NVIDIA P100 GPUs

In contrast to the Intel KNL, NVIDIA P100 GPUs [8], which are more commonly referred to as *Pascal* GPUs, are not self-booting devices. They need to be connected to a host system over the PCIe or NVLink interconnect. Two aspects of the P100 GPUs are important in this paper: data movement, and the memory subsystem.

a) Data Movement: Traditionally, one of the main bottlenecks of GPU computing was the limited PCIe bandwidth. All data to be transferred between the GPU and the host or between two GPUs on the same host had to be transferred over PCIe. To overcome this limitation, NVIDIA introduced NVLink, a new high-speed interconnect especially for GPU computing. The NVLink implementation in P100 supports configurations with aggregate maximum bidirectional bandwidth of 160 GB/sec. NVLink for the P100 focuses mainly on improving the data transfer between GPUs on the same host. However, some CPUs, such as the IBM's Power8 CPU, also support NVLink, allowing for improved data transfer between the host system and the GPU as well. We note that on a multi-GPU system, the GPUs usually have to share links. Thus, the total aggregated bandwidth can be up to 160 GB/sec, while the bandwidth between two single GPUs or the CPU and a single GPU may be lower.

b) Memory Subsystem: The P100 is the first GPU that comes with a new memory architecture, called High-Bandwidth Memory 2 (HBM2). HBM2 is stacked on the same chip as the GPU cores, thus allowing a much denser packing of GPU servers and a much higher bandwidth than on previous GPUs. A P100 GPU has four HBM2 stacks for a total of 16 GB of HBM2 memory.

The new 49-bit addressing of the P100 is large enough to cover the 48-bit address spaces of modern CPUs, as well as the GPU's own memory and the memory of other GPUs. The page fault capability enables on-demand page migration and coherency between host an GPU. This aspect is discussed in more detail in the next section.

III. Explicit Usage of Multiple Memory Classes

A. Heterogeneous Memory Usage on Intel KNL

When the KNL chip is configured in flat or hybrid mode, the HBM is exposed as one or more NUMA nodes. In the SNC modes, the HBM and the DRAM are further divided into more NUMA nodes, but the CPUs are always on the same NUMA nodes as the DRAM is.

If simple memory allocation tools such as `malloc` are used, they treat the KNL architecture as a simple NUMA architecture and naturally pick the memory that is "local" to the CPUs. This local memory is DRAM in all of the above cases, which results in these memory allocation tools ignoring the HBM unless the amount of memory allocated is large. We note that this policy is not accidental but by design, since it ensures that noncritical data (e.g., for the operating system) is not allocated in HBM and defaults to the DRAM.

The simplest approach to using HBM is through the *numactl* [6] tool, which allows one to bind all memory to the appropriate HBM node(s). This ensures that all variables, static and dynamic, are allocated in HBM. While this approach is simple and convenient, it is restrictive for applications that require more fine-grained management of where to allocate specific data objects.

For more fine-grained control, *libnuma* can be used. In this case, the application first has to detect the current configuration of the KNL, including which NUMA nodes correspond to the HBM. Since the enumeration of the HBM and DRAM nodes changes for the different modes, the application has to be specially optimized each configuration.

Another possibility is to use `memkind` [2], which was developed by Intel to allow explicit allocation of HBM. The benefit of `memkind` is that it has been designed and optimized for fine-grained memory allocations and multithreading support. Unfortunately, neither of these is typically as relevant for HPC applications that tend to allocate large memory segments at initialization time using a single thread. Furthermore, `memkind` does not offer the user enough portability and flexibility to navigate the complex configuration space of the KNL and future Intel Xeon Phi architectures. Moreover, it is not portable to other heterogeneous memory architectures such as NVIDIA's GPUs.

B. Heterogeneous Memory Usage on NVIDIA P100

The preferred memory allocation model on NVIDIA P100 GPUs is based on the `cudaMallocManaged` call. This call was introduced with unified memory in CUDA 6, but P100 GPUs are the first generation with hardware support for this feature.

Unified memory was introduced mainly to simplify programming, as it allows both the CPU and GPU to use a single pointer, while the CUDA system software automatically migrates data between the GPU and the CPU. On GPUs prior to P100, all managed memory touched by the CPU had to be synchronized with the GPU before any kernel launch. This approach is no longer necessary as a result of the new page fault capability. Specifically, if a kernel running on the GPU accesses a page that is not resident in its memory, it triggers a page fault, allowing the page to be automatically migrated to the GPU memory on-demand. Pages that are swapped out of the GPU memory are managed on the host memory. Note that accessing host-memory from the GPU was possible before (if `cudaMallocHost` was used), but now the GPU can dynamically decide whether a page needs to be migrated to the GPU or stays on the host. In this mechanism global data coherency is guaranteed if unified memory is used. Thus, with P100, the CPUs and GPUs can access unified memory allocations without any programmer synchronization.

To optimize the performance of unified memory, the programmer can give *hints* to the CUDA runtime to optimize data location and use prefetching instructions to move data between the GPU and the CPU.

IV. `Hexe`: Design and Implementation

In this section, we discuss the overall design and implementation details of the `hexe` framework.

A. Topology Detection and Initialization

The initialization of the `hexe` runtime system and the topology detection happens in three steps.

1) The top-level identification of the overall architecture, such as the Intel Xeon Phi vs. NVIDIA GPUs, is provided by the core operating system, which `hexe` utilizes.
2) For the next level of topology detection we use the `hwloc` library to detect the available compute and memory resources. This gives us information about the location, size, and other properties of the available HBM.
3) The third level of topology detection is architecture specific. For Intel Xeon Phi architectures, we detect what memory and cluster modes the system is configured as. For NVIDIA GPUs, we detect aspects such as whether peer-to-peer access is possible within multiple GPUs (and if yes, among which GPUs).

We note that `hexe` performs a more comprehensive topology and architectural feature detection than what is presented above. For brevity, an exhaustive list of all detected features is not described here. But we wish to convey two important ideas. First, detecting various architectural features is complex; while applications can implement this ability themselves, doing so portably is challenging and error-prone. Second, initialization and detection are a one-time process that allows `hexe` to gather information about the system and later intelligently use it for managing the memory allocations on the system.

B. Memory Allocation Attributes

Memory allocation in `hexe` is divided into three parts: virtual address allocation, mapping the virtual address to a class of physical memory, and mapping the virtual address to an exact physical memory page to be used. The first part is always immediate and done as soon as the allocation call is issued. The second part can be done immediately or lazily (we will discuss both models in this paper). The third part is handled by the OS and is not `hexe`'s responsibility.

A clear definition of responsibilities is in order here. `Hexe` divides the available memory into classes (e.g., on-chip high-bandwidth memory, off-chip persistent memory). During the memory allocation routines, `hexe` only gives hints to the operating system as to which memory class to use; it does not enforce the actual mapping. That responsibility sits with the operating system, which can, in rare circumstances, ignore the hints and choose its own policy. Such a case is possible, for example, when multiple applications are competing for the available memory resources. Having said that, in most high-performance computing environments, application processes that share a node are often symmetrical, making this less of a concern.

Memory allocation in `hexe` is based on five attributes: memory class, data splitting, flexibility in memory binding, severity of mismatches, and memory allocation priority. All memory allocation routines in `hexe` return a virtual address. The attributes described above determine what physical memory class this virtual address corresponds to, how and when the mapping is done, and what happens when the requested mapping is not possible.

a) Memory class: This refers to the different kinds of heterogeneous memory available in the system (e.g., HBM or DRAM). Each memory class has an additional locality attribute called "index," which refers to the specific instance of the memory object when multiple instances exist. For example, if four HBM nodes are available, the index refers to specific node to be used. `Hexe` allows multiple memory classes and memory class indices to be provided within a single allocation call. This feature allows the memory allocation to be bound to multiple memory regions simultaneously, which is important for data splitting (discussed below). `Hexe` also supports some generic indices such as "ALL" and "LOCAL".

b) Data splitting: This refers to how the virtual address space is split between multiple different physical memory classes. Four data splittings are currently defined: `OS`, `SPILL_OVER`, `EQUAL`, and `user-defined chunk size`. The `user-defined chunk size` splits the memory in chunks of the size U (that needs to be a multiple of the page size) and distributes them across the classes. `EQUAL` splitting is a special case of the `user-defined chunk size` where the buffer is split into equal contiguous chunks across the different memory classes. The SPILL_OVER splitting allocates as much data as possible on the preferred class of physical memory and then spills over to the next class. The OS splitting leaves the actual splitting to the operating system, instead of being managed explicitly by the user.

We note that the SPILL_OVER mode is based on what hexe detects as the available memory on the node. If multiple processes compete for the same memory, then hexe's view of the available memory might not be accurate. We are considering parallel allocation strategies that would allow hexe to share such information across multiple processes; but the current version of hexe does not provide such capability.

c) Flexibility in memory mapping: This refers to how soon the virtual address space should be mapped to a class of physical memory. Two modes are currently defined: IMMEDIATE and ON_COMMIT. In the IMMEDIATE mode, the virtual address is mapped and bound to a class of physical memory as soon as the memory allocation is done. In the ON_COMMIT mode, a separate commit call is needed before the virtual address is mapped to the appropriate class of physical memory. This mode gives hexe the ability to look at multiple memory allocation calls and make a global decision as to which allocations should be mapped to which class of physical memory.

d) Severity of mismatch: This refers to how the hexe runtime should deal with cases where it is unable to meet the user memory allocation requests. Two modes are defined: ERROR and FALLBACK. In the ERROR mode, if a user-provided requirement for one of the above attributes cannot be met, then the memory allocation fails immediately. In the FALLBACK mode, the memory allocation silently falls back to a different class of memory as long as any memory is available in the system.

e) Memory allocation priority: This refers to the priority to be given to this memory allocation call when multiple memory allocation requests are pending. Allocation priority is useful only when multiple allocations are outstanding, for example, in the ON_COMMIT mode described above. In the IMMEDIATE mode, however, the priority attribute is ignored.

Internally, on KNL platforms, hexe uses mmap to allocate memory and mbind to bind the memory to specific classes of physical memory. For allocations of more than 2 MB, we use madvise to enable huge pages for better performance (similar to what malloc does on newer Linux distributions). On GPU platforms, it uses cudaMallocManaged to allocate memory and cudaMemAdvise to give hints for the location of the memory.

A few words of caution are warranted.

1) While the hexe interface allows memory allocations to be generally usable on different architectures, users are expected to use them in a way that is in line with their programming interface. For example, splitting data between multiple HBM nodes works well with a program written in traditional C/C++ when used on KNL architectures. While one can do so even with a CUDA program, such an action is not quite in line with general CUDA programming.
2) The locality subattribute in hexe makes sense only when the process itself is bound to a subset of cores where a certain class of memory is closer than the rest. We recommend that users use hwloc or other tools to perform such process binding before requesting "local" memory binding.

C. Lazy Memory Allocation

As described in Section IV-B, hexe allows the mapping of the virtual address to the class of physical memory to be done either immediately or lazily. In this section, we discuss the lazy allocation model. The general idea of this model is that all memory allocation requests are tracked inside hexe until the application calls a hexe_commit call. During a commit call, hexe library analyzes all allocation requests and maps them to the appropriate class of physical memory so as to maximize allocations that match the user's requested attributes.

In this mode the function hexe_malloc is only a request for memory. It allocates a virtual address space corresponding to the buffer allocation but does not map this virtual address space to any class of physical memory at this time. Later, during the hexe_commit call, the buffers are mapped to the actual classes of physical memory. The runtime uses a *knapsack* algorithm to decide, based on the priority and the size, which objects are bound to what class of physical memory.

Within hexe, we optimized the knapsack algorithm to work well with large allocations. If the total amount of the requested memory significantly exceeds the available memory, only the small objects are allocated in their preferred class of memory, while everything else is allocated in a split manner. If two allocations for a particular class of physical memory have almost the same size and the same priority, our algorithm splits both allocations between their preferred class and another fallback class.

An important consideration here is whether the data has been touched before the hexe_commit call. This approach works best if the data are not touched before hexe_commit is called. The reason is simple. Before the memory is touched, no mapping exists between the virtual and physical pages. In this case, binding the memory to a specific class of physical memory causes almost no extra overhead. As soon as the data is touched, however, the operating system performs a virtual to physical mapping. At this point, rebinding pages to a new memory region is expensive since the pages now have to be moved. While hexe does not restrict usage one way or another, users should be careful about this issue from a performance standpoint.

Both immediate and lazy memory allocations can be used simultaneously inside the application. As one would expect, only the lazy memory allocations are remapped to the appropriate class of physical memory during a hexe_commit call.

The application also can use multiple "epochs" of memory allocations, each epoch ending with a hexe_commit call. In such cases, hexe allows the user to remap virtual address to physical memory class bindings either for only the allocation calls within that epoch or for all allocation calls including those in the previous epochs. This process can be done by using either the `HEXE_COMMIT__EPOCH` or the `HEXE_COMMIT__ALL` parameter to the `hexe_commit` call. Using the `HEXE_COMMIT__EPOCH` is typically cheaper, depending on the number of memory allocations requested within that epoch.

V. Experimental Evaluation

In this section we showcase some experimental results for memory management using `hexe` compared with `malloc`, `memkind`, and `CUDA`. Because of the large configuration space of KNL, we cannot present here results for all configurations. Therefore, we limit our presentation to the cache and flat memory modes and to the quadrant and SNC-4 cluster modes. For the experiments with GPUs, we use two configurations: (1) host memory and memory on one GPU and (2) host memory and memory on two GPUs (with peer access enabled).

A. *Experimental Testbed Platforms*

We used two testbed platforms for our experiments. The first platform is the KNL cluster in the Joint Laboratory for System Evaluation at Argonne National Laboratory. This system is based on the Intel Knights Landing 7120. The KNL 7120 comes with 64 cores that clock at 1.3 GHz base frequency. Each node has 16 GB of HBM and 192 GB of DRAM. The memory modes and cluster modes can be changed at boot time.

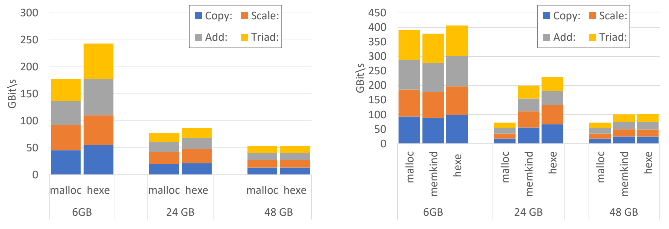
The second platform that we used is a single node of the JURON pilot system at the Jülich Supercomputing Center. The node consists of a POWER8 processor (with 2 $\times$ 10 cores clocked at 4.023 GHz) and four NVIDIA Tesla P100 GPUs. Each GPU has 16 GB high-bandwidth memory and every node has 256 GB off-chip DDR4 memory. The GPUs and the host system are all connected to each other with NVLINK.

B. *Stream Benchmark*

Our first set of experiments is based on the `STREAM benchmark` [7]. The benchmark measures the memory bandwidth for continuous, long-vector memory accesses and is generally considered to be the ideal benchmark to showcase the benefits of high-bandwidth memory and optimized allocation strategies. It requires three arrays (a, b, and c) of the same size and executes four compute kernels on these arrays.

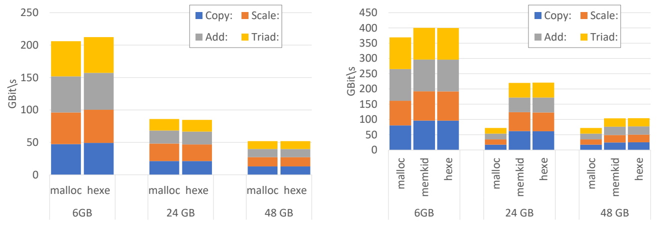
- COPY: $a(i) = b(i)$
- SCALE: $a(i) = q * b(i)$
- ADD: $a(i) = b(i) + c(i)$
- TRIAD: $a(i) = b(i) + q * c(i)$

For our experiments, we ran the benchmark with three different array sizes each of which has a different optimal placement for the arrays on heterogeneous memory systems. Our implementation of the benchmark that used `hexe` enabled the lazy memory allocation feature, so we did not have to change the code for the different array sizes. The `hexe` runtime system automatically adapts the physical memory usage depending on the allocation sizes. Furthermore, this



(a) Cache mode (b) Flat mode

Fig. 1: STREAM benchmark on KNL in SNC-4 mode



(a) Cache mode (b) Flat mode

Fig. 2: STREAM performance on KNL in quadrant mode

allowed us to use the same binary for all four tested modes using `hexe`, which was not possible with `memkind`.

Different sizes of the arrays require a different memory distribution, as shown in Table I. The top row in the table shows the array name and in parentheses the priority associated with them when `hexe` is used. All four compute kernels (`COPY`, `SCALE`, `ADD`, and `TRIAD`) write to array a. Since the write bandwidth on most memory subsystems is worse than that of the read bandwidth, we give a higher priority to a than to b and c. [1]

TABLE I: Memory distribution priorities used in `hexe`

Size	a(2)	b(1)	c(1)
6 GB	HBM (interleave)	HBM (interleave)	HBM (interleave)
24 GB	HBM	interleave	interleave
48 GB	(interleave)	interleave	interleave

We note that the `hexe`, `memkind`, and `malloc` versions of the STREAM benchmark are similar except for how the memory is allocated. Thus, the performance comparison is isolated to only the flexibility offered by each framework in how memory can be allocated. All benchmarks were run with 64 threads.

Since `memkind` is not portable to the cache mode, we could not evaluate the `memkind` version of the benchmark in that mode. Therefore we cannot present those results here. All graphs in this section show the average bandwidth of the four kernels including the cost of each kernel.

1) KNL STREAM performance (SNC-4 mode): In this set of experiments, we compare the performance of `hexe` with

[1]The actual difference in performance, however, depends on the actual memory architecture. On KNL, we noticed that the write bandwidth was up to two times worse than that of the read bandwidth, for both HBM and DRAM. On GPUs, the difference was less.

that of malloc and memkind on the KNL platform booted in SNC-4 mode. The results are shown in Figure 1.

6 GB Dataset: In the first case, each array requires 2 GB of data, which leads to a total memory requirement of 6 GB for all the data to fit into HBM.

The benchmark using malloc was started with numactl to guarantee that all allocations are placed in HBM. With memkind, the memory allocation capability is not rich enough to place the data in the correct quadrant of KNL HBM. There are only two options: (1) using memkind_hbw where all data is placed on the first quadrant irrespective of where it is being accessed from or (2) using memkind_hbw_interleave where all data is interleaved between all the quadrants. When multiple threads from different cores are running the STREAM benchmark, neither of the two modes provided by memkind is ideal since neither mode honors the locality of the cores and their proximity to the memory. We picked memkind_hbw_interleave for our experiments since our benchmark required more memory than what a single HBM node could provide. In our implementation of the STREAM benchmark with hexe, we used memory interleaving across all the HBM nodes with an equal chunk size, thus ensuring that each core accesses memory that is closest to itself.

We note that, in flat mode, the memkind version of the benchmark performs worse than the malloc version. This result is surprising since both malloc and memkind bind the buffer to all of the HBM and then let the operating system pick the buffer locality using its internal policy (such as first-touch). On further analysis we found that this is because memkind uses regular-size pages (4 KB) while malloc and hexe use large pages (2 MB) in this case, thus resulting in more TLB misses in memkind causing some performance loss.

In cache-mode hexe has a better performance, since it distributes the memory equally across the DRAM nodes, while malloc places the data with a first-touch strategy. Since, in cache-mode, MCDRAM is used as a *direct mapped cache* the distribution done by hexe leads to a better utilization of this cache.

24 GB Dataset: For a total dataset size of 24 GB, each array is 8 GB. At this size, two arrays completely fit into HBM while at least one has to be allocated in DRAM.

As noted earlier, array a has a higher priority compared with b and c in the hexe version of the benchmark. In the flat mode, hexe's lazy allocation places a completely in HBM, while b and c (which have the same priority) are interleaved between HBM and DRAM.

Since memkind does not allow for any prioritization, we manually adopt the same policy as hexe as described in Table I. While it is not as convenient to the user, this does allow memkind to achieve some of the performance gains of hexe. Nevertheless, despite these performance gains, memkind still does not achieve the same performance as hexe because of its lack of locality in the HBM. We also tried other memory allocation options with memkind, but this allocation model showed the best performance.

On the other hand, malloc shows worse performance than both hexe and memkind. The reason is that the problem size exceeds the size of the HBM, so numactl cannot be used, and everything is allocated in DRAM with malloc.

48 GB Dataset: The largest dataset size that we tried was 48 GB, where each array is 16 GB. In this case, none of the arrays completely fits into the HBM, so hexe and memkind interleave data from all arrays between HBM and DRAM. While hexe and memkind still perform better than malloc, the difference is smaller since the DRAM bandwidth becomes the limiting factor.

2) KNL STREAM performance (quadrant mode): Figure 2 shows the results for KNL in the quadrant mode: Figure 2a in the quadrant-cache mode and Figure 2b in the quadrant-flat mode.

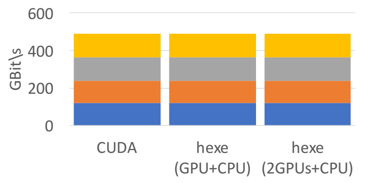
The experiment with a 6 GB dataset is similar to the equivalent experiment in SNC-4 mode. The primary difference is that the memkind and hexe versions of the benchmark both allocate the arrays in HBM using appropriate allocation API calls; hence, their performance is similar. The malloc version is slightly worse than the memkind and hexe versions. To understand this issue, we analyzed the performance further and noticed that all three frameworks internally use mmap but, in contrast to hexe and memkind, malloc does not return a page-aligned address, thus causing some performance loss. We modified the malloc version of the benchmark to manually page align the address returned by malloc, after which we achieved the same performance as memkind and hexe.

The experiments with the 24 GB and 48 GB datasets show trends similar to those in SNC-4 mode with the notable exception that now memkind performs similarly to hexe. The reason is again that both toolkits use similar allocation techniques and the additional flexibility offered by hexe is not beneficial in these case.

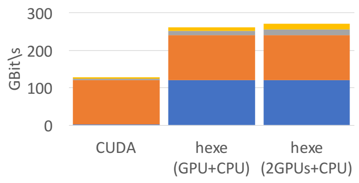
3) GPU-STREAM results: Figure 3 shows the results for the stream benchmark with NVIDIA GPUs. Here, the compute kernels are executed on a single GPU, although the high-bandwidth memory of that GPU, the DDR memory of the host, and the high-bandwidth memory of the other GPUs on the same node are all accessible to that GPU through peer access For better readability, we split up the results for the different dataset sizes. For the CUDA version of memory allocation, we allocated the memory with cudaMallocManaged. For the hexe version, we used the SPILL_OVER mode for memory allocation.

6 GB Dataset: For the 6 GB dataset, the entire data fits on a single GPU, so using cudaMallocManaged or hexe does not make any difference in performance. We note that hexe internally uses cudaMallocManaged as well but gives additional hints to the operating system. These memory hints may help for the first touch; but after that, all data resides in GPU memory, so they make little difference in this test.

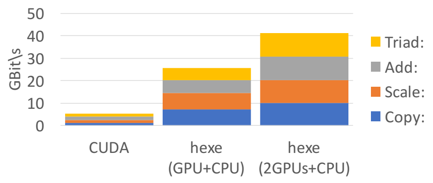
16 GB Dataset: The results for the 16 GB dataset are more interesting. Here, hexe shows better performance than cudaMallocManaged. The reason is that hexe forces



(a) 6 GB memory required



(b) 16 GB memory required



(c) 26 GB memory required

Fig. 3: Results for the STREAM benchmark on GPUs

arrays a and b to reside on the local GPU memory and array c to reside on either the CPU memory (in the GPU+CPU mode) or on the other GPU's memory (in the 2 GPUs+CPU mode). `cudaMallocManaged` dynamically binds memory to the available memory based on page faults. Thus, part of each of the three arrays is on the local GPU, while the rest is on CPU memory (which is used as swap space).

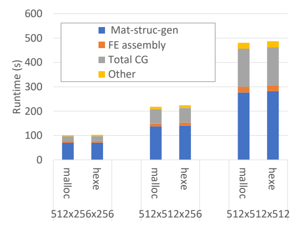
An interesting observation here is the difference in performance for the individual benchmarks. While the `Scale` kernel achieves high bandwidth, the remaining three kernels achieve very low bandwidths. The reason is the order of execution of the kernels. The `Scale` and `Copy` kernels both operate on the same two arrays, a and b, while the `Add` and `Triad` kernels require all three arrays. The `Scale` kernel executes immediately after the `Copy` kernel. Thus, the `Copy` kernel notices all the page faults required to fetch the data into its physical memory (thus facing performance penalty), but the `Scale` kernel already has both arrays in memory when it executes and thus does not face similar performance penalties.

26 GB Dataset: For the 26 GB dataset, only one array fits in the local GPU memory. In this case, `hexe` places a in the local GPU memory, while b and c are placed in host memory (1 GPU/CPU) or b on remote GPU memory and c in host memory (2 GPUs/CPU). For CUDA, this memory requirement causes a lot of page faults for all three kernels. This limits the total bandwidth to 5 GBit/sec for all three kernels. With `hexe`, these page faults are avoided by forcing the placement of the arrays statically based on their priorities, leading to improved performance.

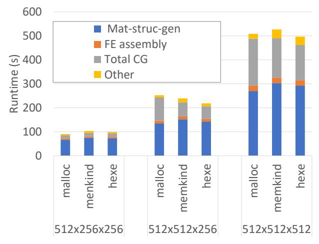
C. MiniFE

We also evaluated the different memory allocation models using the MiniFE miniapp. MiniFE is a bandwidth-limited application in the Mantevo suite [3]. It is a finite-element application that implements a couple of kernels that are representative of implicit finite-element applications. We used the OpenMP-optimized version 2.0.1 of this code [1]. Since the code is written in C++, we developed our own `hexe` allocator class that can be used with C++ STL containers. For comparison, we also implemented an allocator using `memkind`.

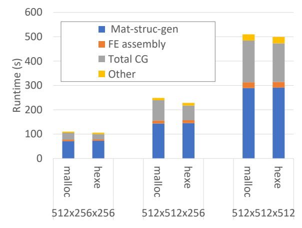
The `memkind` allocator uses the *preferred* option to allocate as much memory as possible on the HBM. For `hexe`, all allocations use a priority of *one*, so the placement was decided mainly based on the size. We believe that a better profiling of the access patterns can further improve performance, but we have not done so here. As with the stream benchmark we ran the MiniFE application with different problem sizes. For the



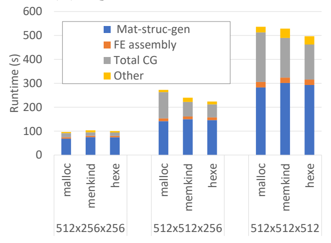
(a) Quadrant cache mode



(b) Quadrant flat mode



(c) SNC cache mode



(d) SNC flat mode

Fig. 4: Results for the miniFE miniapp on KNL

smallest size of $512 \times 256 \times 256$ all data fits into the HBM. For the largest problem size more than 48 GB of memory is required, and only small structures can be allocated completely in the HBM.

The results for the different memory modes of KNL are shown in Figure 4. The graphs show the total runtime of the application split into different phases. `Hexe` shows the lowest runtime for most cases, but the total difference is smaller compared with the STREAM benchmark.

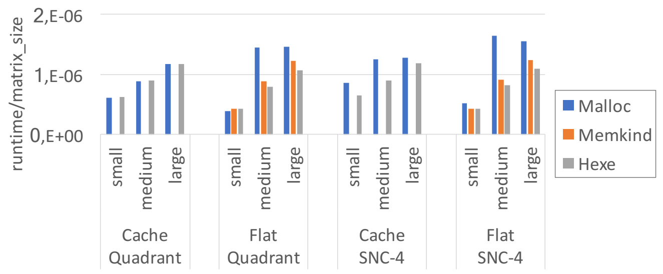


Fig. 5: Runtime of the compute kernels (CG-Solve) for the miniFE benchmark

To further understand these results, we split the total runtime of the application into different phases. Most of the time is spent in the matrix generation phase. Here, the runtimes for `malloc` and `hexe` do not differ. This part of the application is not bandwidth limited. Therefore, it does not make any difference whether the field is allocated in HBM or DRAM. However, the compute-kernels of the MiniFE benchmarks are bandwidth limited. Figure 5 shows the runtime of only the miniFE compute kernels, normalized to the benchmark size. Here, we can clearly see the benefits of `hexe` and high bandwidth memory compared with other allocation types.

VI. Related Work

In [14] and [12], different page-allocation and migration strategies for heterogeneous memory architectures are discussed. The authors propose operating system support for automatic page migration. In [13] this approach is used and compared with transparent usage of HBM as cache in terms of energy and performance. These results are based on hardware simulation and show that no model is the best for all use cases. All these approaches use the operating system to manage the data location, whereas `hexe` distributes the data in user space. This approach also gives `hexe` the advantage of allowing user control of specific memory allocation and placement policies.

In [11], different migration strategies for heterogeneous memory systems are evaluated. The authors use a NUMA system for evaluation. They focus on data migration, a feature that is currently not in `hexe`. However, the results shown in this paper are promising, and such capabilities can certainly help improve `hexe` in the future.

In [5] the authors use the LLVM compiler suite to analyze applications and decide when it is beneficial to allocate data in high bandwidth memory. Internally, they use `memkind` to allocate HBM memory. An approach like this can work well in conjunction with `hexe` because it can be used to analyze applications and set priorities automatically.

Another approach to find the best possible memory layout is tracing. Tools like `hexe` can be used only when the user knows which objects are best to place in HBM. For this purpose, object-based tracing can be used. In [10] and [9] the author introduces an extension to Valgrind and Callgrind that can help find the best static memory layout for applications. It allows a per-object tracking of accesses to memory.

VII. Concluding Remarks

We presented `hexe`, a highly flexible and portable memory allocation toolkit for heterogeneous memory. Compared with other memory allocation tools such as `malloc` and `memkind`, `hexe` improves both the flexibility and portability of memory allocation and management, thus allowing for more sophisticated usage models. Experimental results demonstrate the sophisticated memory allocation model and the corresponding performance benefits it can achieve.

Acknowledgment

This material is, in part, based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357 and, in part, based upon work supported by the Human Brain Project PCP Pilot Systems at the Jülich Supercomputing Centre, which received co-funding from the European Union (Grant Agreement no. 604102).

References

[1] Mantevo Suite Release Version 2.0. https://mantevo.org/download/previous-releases/.

[2] Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. memkind: An extensible heap memory manager for heterogeneous memory platforms and mixed memory policies. Technical Report SAND2015-1862C, Sandia National Laboratories, Albuquerque, NM, USA, 2015.

[3] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[4] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[5] Dounia Khaldi and Barbara Chapman. Towards automatic HBM allocation using LLVM: A case study with Knights Landing. In *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, pages 12–20. IEEE Press, 2016.

[6] Andi Kleen. A NUMA API for Linux. SUSE Labs white paper, 2005.

[7] John D. McCalpin. STREAM benchmark. 1995.

[8] NVIDIA. NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[9] Antonio Peña and Pavan Balaji. A data-oriented profiler to assist in data partitioning and distribution for heterogeneous memory in HPC. *Parallel Computing*, 51:46–55, 2016.

[10] Antonio J. Peña and Pavan Balaji. A framework for tracking memory accesses in scientific applications. In *International Conference on Parallel Processing Workshops*, pages 235–244. IEEE, 2014.

[11] Swann Perarnau, Judicael A. Zounmevo, Balazs Gerofi, Kamil Iskra, and Pete Beckman. Exploring data migration for future deep-memory many-core systems. In *IEEE International Conference on Cluster Computing*, pages 289–297. IEEE, 2016.

[12] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95. ACM, 2011.

[13] ChunYi Su, David Roberts, Edgar A León, Kirk W Cameron, Bronis R de Supinski, Gabriel H Loh, and Dimitrios S Nikolopoulos. HpMC: An energy-aware management system of multi-level memory architectures. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 167–178. ACM, 2015.

[14] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 101–112. IEEE, 2009.