MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing Processors: Developer's Guide

Ryo Asai

Colfax International

May 4, 2016

Abstract

In this publication we discuss the on-package high-bandwidth memory (HBM) based on the multi-channel dynamic random access memory (MCDRAM) technology in 2nd generation Intel® Xeon Phi™ processors codenamed Knights Landing (KNL):

- The three configuration modes of HBM: *Flat mode*, *Cache mode* and *Hybrid mode*,
- Utilization of the HBM as addressable memory using two methods: by setting affinity policy with the numactl tool and through the usage of special allocators in the memkind library,
- Guidelines for determining the optimal usage model for applications running on bootable Knights Landing.

Table of Contents

1 Introduction . . . **2**

2 HBM Modes . . . **3**
- 2.1 Cache Mode . . . 4
- 2.2 Flat Mode . . . 5
- 2.3 Hybrid Mode . . . 5

3 Using HBM as Addressable Memory . . . **6**
- 3.1 numactl . . . 6
- 3.2 Memkind Library . . . 7
- 3.3 Fortran . . . 8

4 Choosing Memory and Programming Model **9**
- 4.1 Programming with HBM... . . . 9
- 4.2 ...and Programming without HBM . . . 10

Appendix A Application Memory Footprint . **11**

Appendix B Bandwidth-critical data . . . **11**

Colfax International is a leading provider of high-performance computing solutions and expert-level educational programs for parallel computing. Ready-to-go Colfax systems include workstations, servers, clusters, storage and personal supercomputing solutions. Educational programs provided by Colfax enable software developers to achieve top performance on cutting-edge computing platforms, closing the loop between hardware innovation and progress in computational disciplines. The comprehensive set of services provided by Colfax delivers to its clients significant price/performance advantages, and increased IT agility, that accelerates their business outcomes and paves the path to discovery. Colfax International's extensive customer base includes Fortune 1000 companies, educational institutions, and government agencies. Founded in 1987, Colfax International is based in Sunnyvale, California and is privately held.

1. Introduction

Memory bandwidth in computing systems is one of the common bottlenecks for performance in computational application. Bandwidth-limited applications are characterized by algorithms that have few floating point operations per memory access (low arithmetic intensity). Algorithms in this category include BLAS Level 1 and Level 2 routines (such as vector dot-product and matrix-vector multiplications), fast Fourier transforms and stencil operations. With low arithmetic intensity in an application, the floating-point capabilities of a processor are generally unimportant, but memory bandwidth determines the application's performance limit (see, e.g., [1]).

To address this demand for memory bandwidth from such applications, the 2nd generation Intel® Xeon Phi™ processors based on the Knights Landing architecture (KNL) have on-package high-bandwidth memory (HBM) based on the multi-channel dynamic random access memory (MCDRAM). This memory is capable of delivering up to ≈5x performance (≥400 GB/s) compared to DDR4 memory on the same platform (≥90 GB/s). Taking maximum advantage of HBM is be key to achieving great performance for bandwidth-sensitive applications.

The available usage models for HBM are different between the two available Knights Landing processor form-factors: self-bootable processor version and the PCIe add-in coprocessor card version. In this publication we will discuss the HBM usage for self-bootable processor version. Its memory organization is shown in Figure 1.

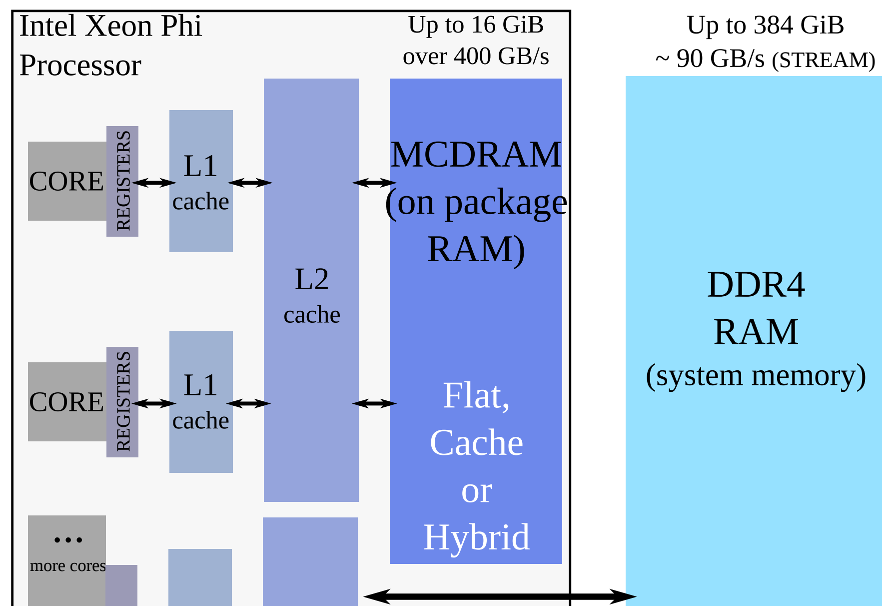
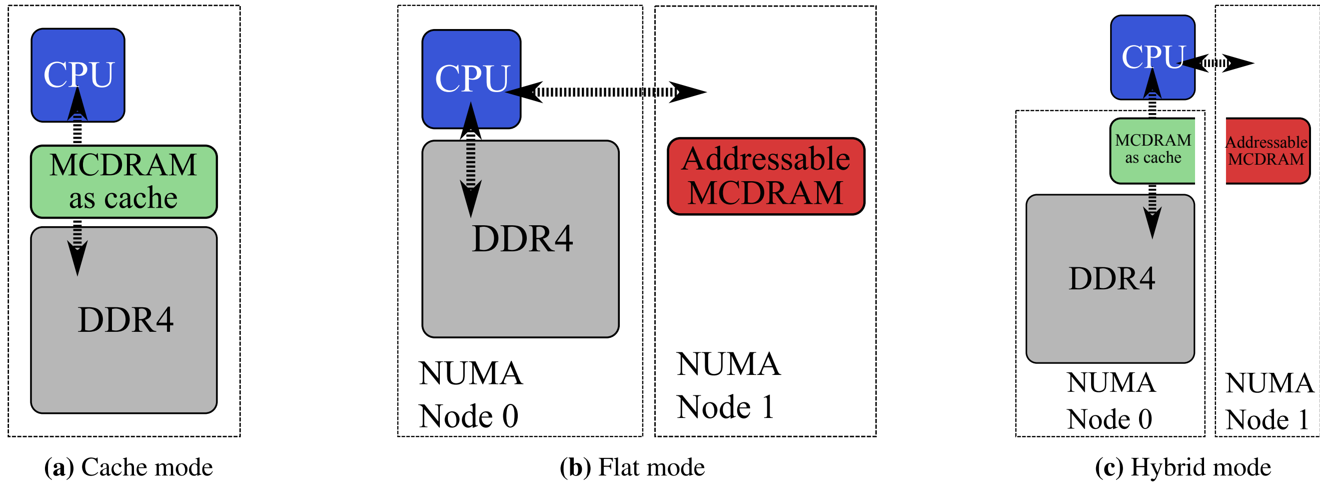


Figure 1: Physical memory organization in bootable 2nd generation Intel Xeon Phi processors (Knights Landing).

The on-package HBM (labeled "MCDRAM" in the figure) resides on the CPU chip, next to the processing cores. Unlike traditional memory modules, MCDRAM cannot be removed or replaced. Depending on the model, Knights Landing processors may have up to 16 GiB of HBM. The on-package memory is distinct from on-platform memory, which is installed as traditional DDR4 memory modules. Depending on the size of the memory modules, on-platform RAM may reach 384 GiB in size.

2. HBM Modes

HBM on a Knights Landing processor can be used either as a last-level cache, or as addressable memory. This configuration is determined at boot time, by choosing in BIOS settings between three MCDRAM modes: *Flat mode*, *Cache mode* or *Hybrid mode*. Figure 2 shows a schematic of the three modes.



(a) Cache mode **(b)** Flat mode **(c)** Hybrid mode

Figure 2: Usage modes of HBM in bootable Kights Landing processors.

The *Flat mode* uses the entirety of HBM as addressable memory, whereas *Cache mode* uses the entirety of HBM as cache. With *Hybrid mode*, a portion of the HBM is used as addressable memory and the rest is used as cache. Addressable memory may be utilized by the user for explicit allocation of objects, while HBM as cache is not visible in the operating system (OS) and operates "behind the scenes" as a last-level cache between the L2 cache and the on-platform DDR4 memory.

Advantages and disadvantages of the three modes are:

- *Cache mode* – No work required to use, but may have lower performance than flat mode in case of frequent misses in HBM as cache.
- *Flat mode* – May offer better performance than cache mode, but requires modifications of the code and/or execution environment.
- *Hybrid mode* – Benefit of both *Flat mode* and *Cache mode*, but smaller sizes for each.

The best mode to use will depend on the application. Guidelines for choosing between the modes for a particular application are discussed in more detail in Section 4. In the remainder of this section, we will delve into the technical details of the three MCDRAM modes.

2.1. Cache Mode

When the Knights Landing processor is booted in *Cache mode* or *Hybrid mode*, all or a part of the HBM is used as cache. HBM cache is treated as a Last Level Cache (LLC), which is located between L2 cache and addressable memory in the memory hierarchy of Knights Landing processors. HBM caches the entire physical address space, and is itself cached by L2 cache.

The advantage of using the HBM as cache is that it is managed by the platform and is transparent to software. So no action is required for the developer to use the HBM in an application, which makes HBM as cache effective for developers unfamiliar with memory performance tuning, or for applications that are difficult to tune.

The drawback of HBM as cache is that it potentially increases the latency of access to the on-platform memory (DDR4). Figure 3 illustrates the case where a core requests data from a memory address not cached in the L1 or L2 cache, and the HBM cache does *not* have the requested address, either.

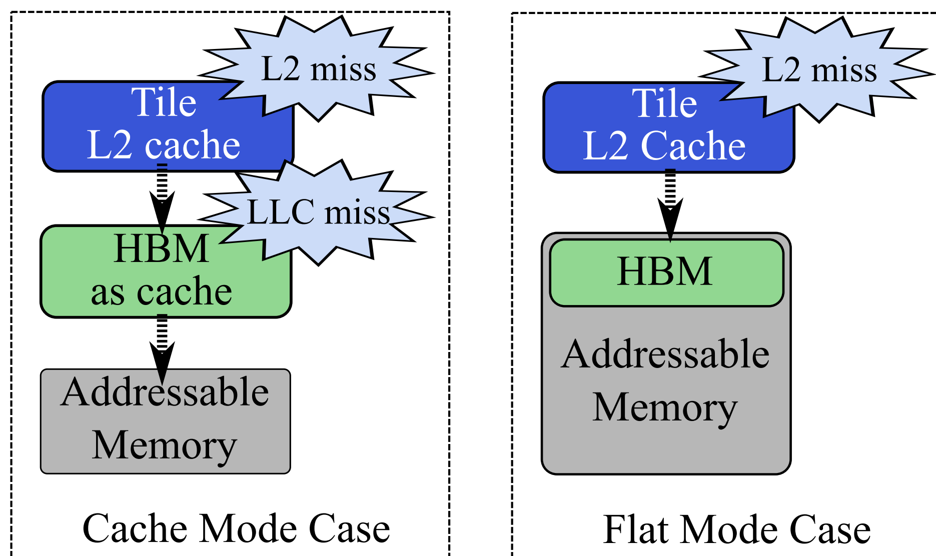


Figure 3: Cache miss latency for *Flat mode* and *Cache mode*.

In the *Cache mode*, the core will have to query the HBM as cache, and only after that will it forward the request to the on-platform memory controllers. In contrast, if HBM is used in the *Flat mode*, as a part of the addressable memory, the core with an L2 cache miss will go directly to the on-platform memory. HBM as cache adds an extra step, and hence latency, when both L2 and HBM cache have a miss.

2.2. Flat Mode

When the Knights Landing processor is booted in *Flat mode*, the entirety of the HBM is used as addressable memory. HBM as addressable memory shares the physical address space with DDR4, and is also cached by L2 cache. With respect to Non Uniform Memory Access (NUMA) architecture, the HBM portion of the addressable memory is exposed as a separate NUMA node without cores, with another NUMA node containing all the cores and DDR4.

A valuable command-line tool when using the HBM as flat mode is the numactl tool (part of the Linux distribution). To find out which NUMA node is associated with HBM, run the command `numactl` with the `--hardware` or `-H` option and look for the node with no cores.

```
user@knl% numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16  ... rest of the cores ...
node 0 size: 98207 MB
node 0 free: 94141 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15923 MB
```

Listing 1: Using `numactl -H` to find the size and NUMA node of HBM.

For the system in Listing 1 (a pre-production 2nd generation Intel Xeon Phi coprocessor with 16 GiB of MCDRAM), NUMA node 0 is the on-platform DDR4 memory with all the cores, and NUMA node 1 is the on-package MCDRAM with no cores associated with it (i.e., the HBM).

The advantage of HBM as addressable memory is that the developer has more detailed control over HBM when compared to the HBM as cache. With appropriate tuning, an application with memory usage over 16 GiB may perform better with HBM as addressable memory than with HBM as cache. The reason for this is that in *Flat mode*, unlike *Cache mode*, accesses to objects in on-platform memory do not have to query the on-package HBM first.

However, using HBM as addressable memory often requires more application development compared to HBM as cache, because the usage of HBM is completely left to the developer. Application memory is allocated to DDR4 by default, so the developer needs to specify allocation in HBM with either command-line tools or special allocators (discussed in Section 3). Performance tuning is often more difficult for HBM as addressable memory, especially for larger or more complex applications.

2.3. Hybrid Mode

When the Knights Landing processor is booted in *Hybrid mode*, a portion of the HBM is used as addressable memory and the rest used as cache. The ratio between the two is chosen at boot-time. The addressable memory portion acts just as it does in *Flat mode*, and the cache portion acts just as it does in *Cache mode*. This mode is useful for large multi-user clusters, where not all applications have been adapted to use HBM as addressable memory.

3. Using HBM as Addressable Memory

When using *Flat mode* (or *Hybrid mode*), the developer has to manually direct the application to use HBM as addressable memory. There are two methods for this: the numactl tool and the memkind library. The recommended method depends on the size of HBM as addressable memory and the memory footprint of the application. The tool `numactl` can be used to find the size of HBM as addressable memory (see Listing 1) and a procedure for checking the memory footprint is discussed in Appendix A. If the memory footprint of the application is *less* than the size of HBM as addressable memory, then it is recommended to use numactl. If it does not, then use the memkind library.

Note: For the remainder of Section 3, HBM will refer to HBM as addressable memory.

3.1. NUMACTL

If the memory requirement of an application is smaller than the size of the available HBM, then the recommended way to take advantage of HBM is to have all allocations happen in HBM by default. This can be achieved by using the command `numactl` with the `--membind` or `-m` option in order to bind an application to a particular NUMA node. When an application is bound to a NUMA node, all allocations within the application will happen on the specified NUMA node's memory by default. Because `numactl` is a command that is used at run-time, this method requires neither any code modification, nor recompilation.

As we have determined (see Listing 1) that NUMA node 1 is the HBM, we can to bind memory allocation in executable `run-app` to HBM like this:

```
user@knl% numactl --membind 1 ./run-app
```

It is important to note that the NUMA node associated with the HBM is different depending on a separate boot-time configuration option for cluster mode (see [?] for more information on cluster modes). Thus, it is good practice to always check the NUMA configuration of the Knights Landing processor with `numactl -H` before using `numactl -m` to bind the application to the NUMA node with HBM.

To use HBM NUMA nodes in the SNC-4/SNC-2 modes (the above cluster modes), use a comma separated list for `numactl --membind`

```
user@knl% numactl -m 1,3,5,7 ./run-app
```

Note that this methodology works for *any* executable with memory footprint smaller than 16 GiB, regardless of the programming language used (provided that the code does not change the NUMA policy and does not specifically allocate to the on-platform memory).

3.2. Memkind Library

Memkind library is a user-extensible heap manager built on top of jemalloc, a C library for general-purpose memory allocation functions. The library is generalizable to any NUMA architecture, but for Knights Landing processors it is used primarily for manual allocation to HBM using special allocators for C/C++. The library also has limited support for Fortran, which is discussed in Section 3.3.

An open source version of the memkind library can be downloaded from [2]. The memkind library has two interfaces: *hbwmalloc* and *memkind*. Both interfaces use the same back-end. In fact, *hbwmalloc*, which stands for high-bandwidth memory allocator, calls the *memkind* interface functions internally. Thus *memkind* interface has all the functionality of *hbwmalloc* interface, as well as some experimental features not available to *hbwmalloc* interface. However, at the time of writing this paper, the *hbwmalloc* interface is stable but *memkind* interface is only partially stable. Thus in this publication we will focus on *hbwmalloc* interface. For users interested in the memkind library, you can find the Linux manual page for *memkind* interface with:

```
user@knl% man memkind
// ... Memkind manual ... //
```

Listing 2 shows basic memory allocation/deallocation using *hbwmalloc* interface, and Listing 3 shows the procedure for compilation.

```
1 #include <hbwmalloc.h>   // hbwmalloc interface
2 // ... //
3
4 const int n = 1<<10;
5 double* A = (double*) hbw_malloc(sizeof(double)*n); // Allocation to HBM
6 // ... //
7
8 hbw_free(A); // Deallocate with hbw_free
```

Listing 2: Basic allocation/deallocation with *hbwmalloc*

```
user@knl% icpc foo.cc -lmemkind -o run-app-intel
user@knl% g++ foo.cc -lmemkind -o run-app-gcc
```

Listing 3: Linking for *memkind* and *hbwmalloc*.

`hbw_malloc()` works just like the standard `malloc()`: it allocates a block in the HBM and returns the pointer to the start of the block. Two variants of heap allocators in `jemalloc` library, `calloc()` and `realloc()`, have HBM counterparts, `hbw_calloc()` and `hbw_realloc()`.

It is important to note that the memory allocation functions will only *attempt* to allocate in the HBM. If there is insufficient space in available HBM, it will fall back to DDR4 memory *without warning*. This behavior can be modified with `hbw_set_policy()`. For the description of the available policies, visit the Linux manual page for *hbwmalloc* with:

```
user@knl% man hbwmalloc
// ... hbwmalloc manual ... //
```

Listing 4: *hbwmalloc* manual.

It is also possible to determine whether HBM is available by using the `hbw_check_available()` function. Note that this does *not* return how much is left: only whether or not there is any available.

hbwmalloc also has more advanced allocators that provide the control required for certain optimization. For example, sometimes you may want to specify memory alignment for your allocations (see [3]). To allocate an aligned memory block with *hbwmalloc*, use `hbw_posix_memalign()`. Figure 5 demonstrates aligned allocation in HBM.

```
1 double* A;
2 int ret = hbw_posix_memalign((void*) A, 64, sizeof(double)*n);
3 // ..... //
4 hbw_free(A);
```

Listing 5: Aligned allocation for HBM.

It is also possible to specify the memory page size using `hbw_posix_memalign_psize()`. For more information on the available page size options, check the Linux manual page for *hbwmalloc* (see 4).

3.3. Fortran

Memkind library does have support for Fortran, but it is limited in scope. With Fortran, only allocatable arrays may be explicitly placed in HBM. Attribute `FASTMEM` directs the runtime system to place an allocatable array to in the HBM.

```
1  REAL, ALLOCATABLE :: A(:), B(:)
2
3  ! FASTMEM attribute
4  !DEC$ ATTRIBUTES FASTMEM :: A
5
6  ! A is allocated in HBM
7  ALLOCATE (A(1:1024))
8
9  ! B is allocated in DDR4
10 ALLOCATE (B(1:1024))
```

Listing 6: Manual memory allocation in HBM with Fortran

At the same time, as mentioned above, Fortran applications may be run completely from HBM using the numactl tool, as long as these applications fit completely within the installed HBM (up to 16 GiB).

4. Choosing Memory and Programming Model

4.1. Programming with HBM...

So far in the publication, we have discussed three MCDRAM modes for different HBM usage (see Section 2) as well as two methods for employing the HBM as addressable memory (see Section 3). This leaves open the question of choosing the optimal method of HBM utilization. As a starting point for this decision, we recommend using the flow chart shown in Figure 4 to decide on the usage model of HBM.

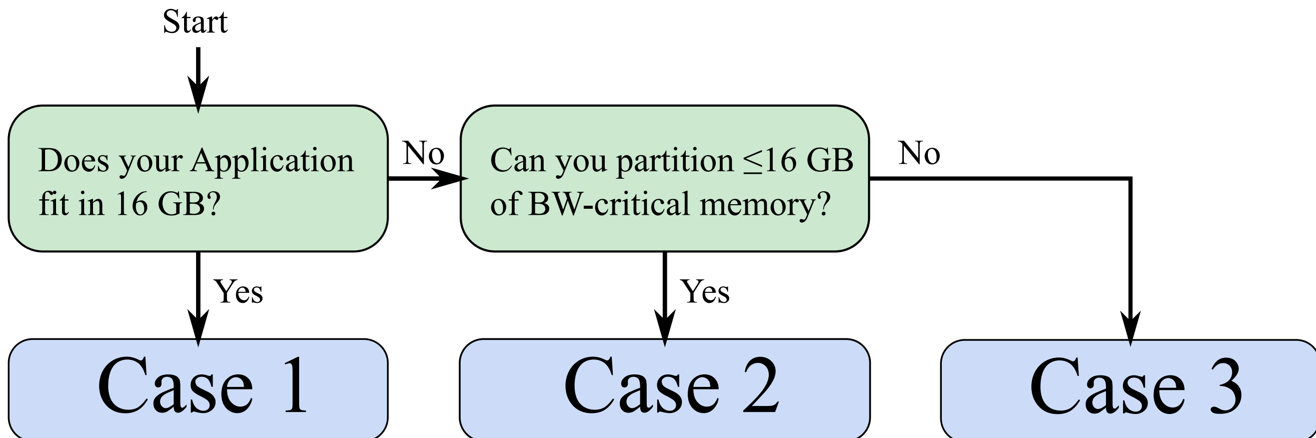


Figure 4: Flowchart for HBM usage.

- **Case 1: The entire application fits in HBM.**

  This is the best case scenario. If your application fits in the HBM, then set the configuration mode to *Flat mode* and follow the `numactl` instructions in Section 3.1. This usage mode does not require any code modification, and works with any application (written in any language) provided that it does not have special allocators that specifically allocate elsewhere. Note that, although this procedure requires no source code changes, applications could still benefit from general memory optimization. For more on memory traffic optimization, refer to the various online references on optimization such as [4].

  If `numactl` cannot be used, then using *Cache mode* could be an alternative. Because the problem fits in the HBM cache, there will only be a few HBM cache misses. HBM cache misses are the primary factor in the performance difference between addressable memory HBM and cache HBM, so using the *Cache mode* could get close to or even match the performance with *Flat mode*. However, there is still some inherent overhead associated with using HBM as cache, so if `numactl` is an option, we recommend to use that method.

- **Case 2: Bandwidth critical part of the application can be partitioned and fits in HBM.**

  If portions of the data set that are *bandwidth-critical* can fit in HBM, then we recommend using the *Flat mode* and the memkind library (see Section 3.2). This usage mode will require some code modification, as well as knowing which components of the data set are *bandwidth-critical*. For strategies for determining this, refer to Appendix B.

If memkind library is inaccessible for some reason, (e.g. your application is in a language that does not support it) then you could still take advantage of this model with external library calls. For example, the Intel® Math Kernel Library (MKL) can be called from multiple languages, such as Python and R. Of course, you could also create your own C/C++ modules that use the memkind library.

- **Case 3: Application is too large and difficult to partition.**

  Unfortunately, this is the worst case scenario. At this point, our recommendation is to default to the *Cache mode*. This will allow your application to at least take some advantage of the HBM.

  It is recommended to also try the *Flat mode* with `numactl` to allocate *to the DDR4*. This is because the extra cache miss latency of HBM as cache (see Section 2.1) may outweigh the benefit of HBM in some applications.

  In either case, we recommend you to try the techniques listed in Appendix B to try to find and partition *bandwidth-critical* portions of the data. This will allow you to use the method in **case 2**.

4.2. ...and Programming without HBM

At this point, recall that HBM is useful only for applications sensitive to memory bandwidth. If your application has high arithmetic intensity (i.e., high data re-use in caches) or very little memory traffic (i.e., fits in the L2 cache), you may as well forget about the on-package memory and keep using the regular on-platform memory on Knights Landing like you would on a general-purpose CPU.

Examples of applications that are not sensitive to memory bandwidth are BLAS Level 3 routines (such as general matrix-matrix multiplication optimized for cache utilization) and some Monte Carlo calculations with small memory footprint.

References

[1] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4):65–76, April 2009.
http://dx.doi.org/doi:10.1145/1498765.1498785.

[2] Open source memkind library on github.
https://github.com/memkind/memkind.

[3] Andrey Vladimirov. Fine-Tuning Vectorization and Memory Traffic on Intel Xeon Phi Coprocessors: LU Decomposition of Small Matrices.
http://research.colfaxinternational.com/post/2015/01/27/LU.aspx.

[4] Colfax Hands On Workshop (HOW) series.
http://colfaxresearch.com/how-series.

Appendix A. Application Memory Footprint

In order to determine the usage model of HBM, you may need to know the memory footprint of an application . There are several methods for checking the memory usage of an application, one of which is the `ps` command. To get the memory usage of an executable `run-app`, run the application and on a separate terminal type:

```
user@knl% ps -C run-app u
USER    PID  %CPU  %MEM VSZ       RSS    TTY    STAT START TIME     COMMAND
user    6577 26330  0.0 17943576 78360 pts/2 Rl+   15:04 267:41 ./run-app
```

The memory being used is the value under *RSS* (in KB). Note that `ps` reports the usage at the time it is called: if the application has a lot of allocations and deallocations, you may want to combine `ps` with `watch` and monitor the usage.

Appendix B. Bandwidth-critical data

In order to use memkind library effectively, it is often beneficial to know what portion of the application data is *bandwidth-critical*. *Bandwidth-critical* memory is a region of the application data that will bring the most benefit to the overall application if it was allocated in HBM. Here are some characteristics to look for when finding *bandwidth-critical* data.

- The total size of the data is $\geq$30 MB

  If it is any smaller, the data will likely be cached by L2 cache. Although you may get some benefit from putting smaller data sets in the HBM, prioritize larger data because they may benefit more.

- The data is read multiple times.

  If some data is read once and never again, this may not be the best candidate for HBM. For allocation in HBM, prioritize data structures that are read or written multiple times.

- The data access pattern is contiguous.

  HBM performs best when the data access pattern is contiguous, and performs worst when the access pattern is random. Although HBM may outperform DDR4 even with some random access, the performance difference is small. You would want to prioritize data structures that are accessed contiguously for allocation in HBM.

- The data is used in a bandwidth-bound section.

  Even if the data itself has all the characteristics of *bandwidth-critical* memory, it may not gain much from HBM if the bottleneck for the workload is not memory (e.g. compute-bound, I/O-bound). Prioritize data that is used in locations where there are few arithmetic operations per data point, or few I/O operations per data point.