Estimating the Performance Impact of the MCDRAM on KNL Using Dual-Socket Ivy Bridge Nodes on Cray XC30

Zhengji Zhao
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, USA
e-mail: zzhao@lbl.gov

Martijn Marsman
Computational Materials Physics
University of Vienna
Vienna, Austria
e-mail: martijn.marsman@univie.ac.at

Abstract— **NERSC is preparing for its next petascale system, named Cori, a Cray XC system based on the Intel KNL MIC architecture. Each Cori node will have 72 cores (288 threads), 512 bit vector units, and a low capacity (16GB) and high bandwidth (~5x DDR4) on-package memory (MCDRAM or HBM). To help applications get ready for Cori, NERSC has developed optimization strategies that focus on the MPI+OpenMP program model, vectorization, and the HBM. While the optimization on MPI+OpenMP and vectorization can be carried out on today's multi-core architectures, optimization of the HBM is difficult to perform where the HBM is unavailable. In this paper, we will present our HBM performance analysis on the VASP code, a widely used materials science code, using Intel's development tools, Memkind and AutoHBW, and a dual-socket Ivy Bridge processor node on Edison, a Cray XC30, as a proxy to the HBM on KNL.**

Keywords-HBM; MCDRAM; KNL; VASP; memory bandwidth; Memkind; AutoHBW; performance

I. INTRODUCTION

NERSC's next petascale system will be a Cray XC 40 system based on the Intel Knights Landing (KNL), Many Integrated Core (MIC) Architecture [1], named Cori [2]. Cori is scheduled to arrive at NERSC this fall (August, 2016), and will have over 9300 single-socket KNL, MIC processor nodes, interconnected with Cray's Aries Dragonfly high-speed network. Each Cori node will have 72 cores (288 threads or logical cores with Hyperthreading) running at a <1.4 GHz clock-speed, 512 bit vector units, 96 GB DDR4 memory, and a 16 GB on-package high bandwidth memory, named Multi-Channel DRAM (MCDRAM, or HBM for simplicity), which will have about 5 times bandwidth of the DDR4 memory (>400GB/s). To help the current MPI predominant NERSC workload [3] make a successful transition to this new energy efficient architecture, NERSC has developed optimization strategies in collaboration with vendors, which focus on the MPI+OpenMP program model, vectorization, and the efficient use of the HBM. NERSC has also initiated a program, named NESAP [4], to help the applications get ready for Cori (Phase II). While optimization on MPI+OpenMP and vectorization can be performed on today's multi-core architectures, optimization on the HBM is difficult to perform where the HBM is unavailable. If entire NERSC workload can fit into the 16 GB HBM on the KNL nodes, optimization of the HBM would be less of a concern. However, at least 50% of the NERSC workload will not be able to fit into the HBM, as shown in our recent workload analysis [3]. Therefore, the efficient use of the low capacity HBM on the KNL nodes will be an important optimization for applications to get the most performance out of Cori. Specifically users and developers need memory management tools that can selectively allocate data structures to the HBM while allocating the rest of the data to the DDR memory.

Intel has developed multiple tools to manage the HBM on KNL, such as the Memkind [5] and AutoHBW [6] libraries. Furthermore, it possible to emulate the HBM performance on today's dual-socket Xeon nodes by making use of the fact that the bandwidths of accessing the near socket memory and the far socket memory via QPI are different. For example, on a dual-socket Ivy Bridge node, the Stream Triad bandwidths of the two memory accessing modes differ by 33%. The near socket memory can be used as the proxy of the HBM (MCDRAM) on KNL, and the far socket memory can be used as the proxy of the DDR memory on KNL. Note that this is not an accurate model of the bandwidth and latency characteristics of the HBM (MCDRAM) on KNL, but it is a reasonable way to determine which data structures rely critically on bandwidths, and get applications ready for the HBM before the KNL nodes arrive. In this paper, we will present our HBM performance analysis on the VASP [7-8] code, a widely used materials science application, using Intel's development tools, Memkind and AutoHBW, and the dual-socket Ivy Bridge processor nodes on Edison [9], a Cray XC30, as the proxies to the HBM on KNL. Our analyses were performed on the two representative VASP workloads.

The work described here is based on one of the optimizations attempted during the VASP Dungeon Session [1] in Oct. 2015, which is a part of the NESAP activities to get the applications ready for KNL. Some of the pre-dungeon preparation and the post-dungeon follow-up work are included as well. The rest of the paper is organized as

[1] Intel Dungeon Session is a multi-day intense application optimization session held at Intel Office at Hillsboro, OR with a team of application developers, Intel engineers, and NERSC staff to get the NESAP application codes (selected among the NERSC workload) ready for Cori (KNL).

follows: We will describe the environment and methods used for our HBM analysis in Section II, and will describe the VASP code and the test cases used in Section III. In Section IV, we will apply the HBM analysis methods to the VASP code and will analyze the HBM performance results. We will conclude the paper by summarizing our observations in Section V.

II. ENVRIONMENTS AND METHODS FOR HBM ANALYSIS

A. *HBM (MCDRAM) on KNL*

The KNL architecture brings in new memory technology, a high bandwidth on package memory called Multi-Channel DRAM (MCDRAM or HBM for simplicity) in addition to the traditional DDR4 memory. MCDRAM is a high bandwidth (~5x more than DDR4) and low capacity (16GB) memory, packaged with the Knights Landing Silicon (Fig. 1). MCDRAM can be configured (boot time options) as a third level cache (cache mode) or as a distinct NUMA node (flat mode) or somewhere in between (hybrid mode). If the MCDRAM is configured as a third level cache, no application source code changes are required, however, the misses are expensive because applications need to access both the MCDRAM and the DDR memory. If the MCDRAM is configured in the flat mode, it is mapped to physical address space and exposed as a NUMA node (allocatable memory). The advantage of this mode is allowing application developers and users have the full control over the HBM use. The downside is that code modifications may be required to be able to utilize the available HBM efficiently. It is very challenging from a software perspective to understand the best mode suitable for an application. In this study, we will focus on the flat mode in which the MCDRAM is configured as a NUMA node (or multiple NUMA nodes).

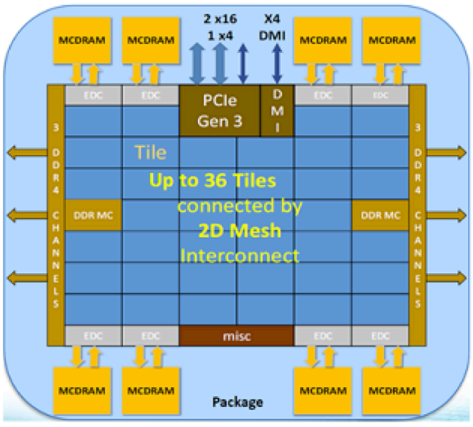

Figure 1. Intel KNL overview (source: Intel). Each KNL processor will have 72 cores (288 threads) at <1.4 GHz, 96GB DDR4 memory utilizing six 16GB DIMMs at 2400GHz, and a 16 GB MCDRAM with about 5x memory bandwidth of the DDR4 memory. Processor cores will be interconnected in a 2D mesh network with two cores per tile (36 tiles in total), with a 1-MB L2 cache shared between two cores in a tile, with two 512 bit vector processing units per core, and with multiple NUMA domain support per socket.

B. *Edison as HBM proxy*

Edison is NERSC's current petascale system, a Cray XC30, which has about 5600 dual-socket Ivy Bridge processor nodes, interconnected with Cray's Aries Dragonfly high speed network. Each Edison node has two sockets, and each socket has 12 cores (24 cores/48 threads per node) at 2.4GHz. Each node has 64 GB DDR3 1866 MHz memory utilizing eight 8 GB DIMMs. Stream TRIAD bandwidth per node is 103 GB/s. The two sockets on the Ivy Bridge node is connected with the QPI links (Fig. 2). The bandwidths of accessing the near socket memory and the far socket memory via QPI differ by 33%. This bandwidth difference can be used to emulate the MCDRAM and DDR on KNL. The near socket memory can be used as the proxy of the MCDRAM on KNL, and the far socket memory can be used as the proxy of the DDR on KNL. This is not an accurate model of the bandwidth and latency characteristics of the KNL on package memory, but is a reasonable way to determine which data structures rely critically on memory bandwidths.

C. *Intel development tools to manage HBM on KNL*

1) Memkind – code changes are required

The Memkind [5] library is a user extensible heap manager built on top of Jemalloc [10], a general purpose heap manger, which enables the control of memory characteristics and a partitioning of the heap between kinds of memory (including user defined kinds of memory). The Memkind library is capable of managing a few different memory types including HBM, hugepage memory, HBM with hugepages, etc., with the HBM being the default memory type. The Memkind library requires minimal code changes. For the Fortran codes, the Memkind library uses

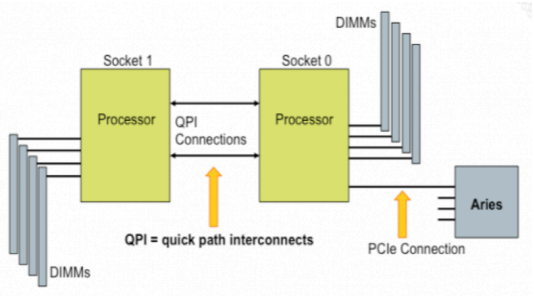

Figure 2. Edison compute node diagram (source: Cray). Each Edison compute node has two 12 cores Ivy Bridge processors (24cores/48 threads per node) at 2.4 GHz. Each node has 64GB DDR3 1866 MHz memory utilizing eight 8 GB DIMMs. Each core has its own L1 and L2 caches, with 64 KB (32 KB instruction cache, 32 KB data) and 256 KB, respectively; a 30-MB L3 cache shared between 12 cores on the processor (socket). Stream TRIAD bandwidth per node is 103 GB/s.

the Intel compiler directive, !DIR$ ATTRIBUTES FASTMEM to select data arrays in the code and to allocate them in the HBM. For the C/C++ codes, it uses the malloc or calloc wrapper APIs, such as hbw_malloc, hbw_calloc, etc., to replace the malloc or calloc calls in the codes, to place those memory objects on to the HBM. The applications need to be linked to the Memkind library (and Jemalloc library as well). To use the Memkind library on Xeon nodes (non-KNL nodes) where the HBM is unavailable, an environment variable, MEMKIND_HBW_NODE, is needed to designate a socket (or a NUMA node) as the HBM node (Note multiple NUMA nodes can be set as the HBM nodes on a node with more than two sockets or NUMA nodes). Fig. 3 shows the steps to use the Memkind library on a dual socket Xeon node.

In Fig. 3 the memory on Socket 0 on a dual-socket node simulates the HBM on a KNL node (via MEMKIND_HBW_NODES=0), and the memory on Socket 1 simulates the DDR memory on KNL. The code, ./a.out, runs (bound to) on Socket 0, allocating the memory from the DDR memory (Socket 1). Only the arrays, which were prefixed with the FASTMEM directive (for example, the arrays a, b, and c in Fig. 3), will be allocated to the HBM (Socket 0). Note that the FASTMEM directive is currently an Intel compiler specific directive. This causes optimization of the HBM to lack of portability between different compilers. Fortunately, Cray compilers will soon support this directive as well [11]. In addition, the Memkind library can only allocate heap variables to the HBM. There is no good way to allocate stack variables to the HBM currently. We showed the steps to use the Memkind library on a standalone dual-socket Xeon node (e.g., under Edison Cluster Compatibility Mode (CCM) [12]). However, the same steps are applicable to the native environment on Cray XC systems where the Cray compiler wrappers and the Cray MPICH libraries are used (see [13]). The Memkind library is an open source tool. See [5] for more details about the Memkind library.

1) *Modify the codes:*
   a) Fortran codes: Add Intel compiler directive !DIR$ ATTRIBUTES FASTMEM
      real, allocatable :: a(:,:), b(:,:), c(:)
      !DIR$ ATTRIBUTES FASTMEM :: a, b, c
   b) C/C++ codes: Use hbw_malloc, hbw_calloc to replace the malloc, calloc calls in the codes
      #include <hbwmalloc.h>
      malloc(size) -> hbw_malloc(size)
2) *Link the codes to the memkind and jemalloc libraries using Intel compilers*
      mpiifort -O3 -qopenmp mycode.f90 -L<path-to-mekind-library> –lmemkind –ljemalloc
3) *Run the codes with the numactl and env MEMKIND_HBW_NODES*
      export MEMKIND_HBW_NODES=0
      numactl --membind=1 --cpunodebind=0 ./a.out

Figure 3. The steps to use the Memkind library on a (standalone) dual-socket Xeon node. To run on multiple nodes, or when a job lands on a mom node instead of a head compute node, a parallel job launcher, e.g., mpirun, srun, aprun or ccmrun, should be used in front of the numactl command in Step 3 as shown above. Then, the "numactl --membind=1 --cpunodebind=0" command could be replaced by the corresponding task/memory affinity options of the job launchers.

2) AutoHBW – no code changes are required

The AutoHBW [6] library is a tool that can be used to automatically allocate arrays within certain size ranges to the HBM without modifications to the application source codes. AutoHBW intercepts the standard heap allocations (e.g., malloc, calloc, etc.,) in the application codes and allocates them in the HBM if the arrays are within the specified size range. An environment variable, AUTO_HBW_SIZE (in M, K, or G) is used to specify the size range for the arrays to be allocated in the HBM. This tool is built on top of the Memkind library. Application codes need to be linked to the AutoHBW and Memkind libraries. If applications are linked dynamically, then the AutoHBW and Memkind libraries need to be preloaded (via LD_PRELOAD) or added to the library search path (LD_LIBRARY_PATH). AutoHBW uses environment variables to control its behavior and to interact with the application codes. See Fig. 4 for the available environment variables. To use AutoHBW on non-KNL nodes the environment variable, MEMKIND_HBW_NODES, must be used to specify which socket/NUMA node to set as the to use the AutoHBW library on a dual-socket Xeon node. Similar to the Memkind library, the AutoHBW library can also be used within the native environment on Cray XC systems with the Cray compiler wrappers and the Cray MPICH libraries (See [13]).

In Fig. 5 all arrays sized between1M to 5M will be

AUTO_HBW_SIZE=x[:y]
Indicates that any allocation larger than x and smaller than y should be allocated in HBM. x,y (in K, M, or G)
AUTO_HBW_LOG=level
Sets the verbosity of the logging level:
0 = no messages are printed for allocations; 1 = a log message is printed for each allocation (Default); 2 = a log message is printed for each allocation with a backtrace.
MEMKIND_HBW_NODES=<list of numa nodes>
Sets a comma separated list of NUMA nodes as HBW nodes, e.g., MEMKIND_HBW_NODES=0
For non-KNL node this env must be set
AUTO_HBW_MEM_TYPE=<memory_type>
Sets the type of memory that should be automatically allocated. Default: MEMKIND_HBW.
Examples:
AUTO_HBW_MEM_TYPE=MEMKIND_HBW (Default)
AUTO_HBW_MEM_TYPE=MEMKIND_HBW_HUGETLB
AUTO_HBW_MEM_TYPE=MEMKIND_HUGETLB

Figure 4. The environment variables used in AutoHBW.

allocated to the HBM. As with the Memkind library, this library also works with the heap arrays only. See [6] for more details about the AutoHBW tool.

III. Application Code and Test Cases

A. VASP code description and computational problem

The Vienna Ab-initio Simulation Package (VASP) [7-8] is a widely used materials science application for performing Ab-initio electronic structure calculations and quantum-mechanical molecular dynamics (MD) simulations using pseudopotentials or the projector-augmented wave method and a plane wave basis set. VASP computes an approximate solution to the many-body Schrödinger equation, either within the Density Functional Theory (DFT) to solve the Kohn-Sham equation or the Hartree-Fock (HF) approximation to solve the Roothaan equation. Hybrid functionals that mix the HF approach with DFT are implemented, and Green's functions methods (GW quasi-particles and ACFDT-RPA) as well as many-body perturbation theory (2nd-order Møller-Plesset) are available in VASP.

The fundamental mathematical problem that VASP solves is a non-linear eigenvalue problem that has to be solved iteratively via self-consistent iteration cycles until a desired accuracy is achieved. This application makes use of efficient iterative matrix diagonalization techniques like the residual minimization method with direct inversion of the iterative subspace (RMM-DIIS) and the blocked Davidson algorithms. FFTs and Linear Algebra libraries (BLAS/LAPACK/ScaLAPACK) are heavily depended on.

Currently, the released production code (e.g., 5.3.5) is a pure MPI code. There is also an MPI+OpenMP hybrid code that the developers (Marsman) are working on to get ready for the next generation multi-core/many-core HPC systems. The majority of the VASP code is written in Fortran. In this study, we used the pure MPI code, version 5.3.5, and the MPI+OpenMP hybrid code as well (the development version as of 9/29/2015). The code was compiled with Intel compilers (15.1.133) and linked to the MKL (11.02 update1) and ELPA (2015.05.001) libraries. The FFT routines were also from MKL via the wrapped fftw3 interfaces.

B. Test cases

In this study, we used two different test cases. The first case was used with the VASP 5.3.5 to test the HBM performance impact to the hybrid functional calculations (HSE06) in the VASP code. The hybrid functional calculations are memory intensive, and are one of the representative VASP workloads at NERSC. See Fig. 6 for its atomic structure configuration (denoted as B.HR105-s hereafter).

The second test case was used with the development version of VASP (as of 9/29/2015, an MPI+OpenMP hybrid code) to optimize the HBM use in the typical code path in VASP (the RMM-DIIS iteration scheme). This is a PdO slab, containing 174 atoms in total (denoted as PdO@Pd-slab hereafter). This system was carefully chosen so that it could fit in the single socket memory on the Haswell node (Xeon processor E5-2697 v3 at 2.6 GHz). The developers (Marsman) used this to prepare for the Dungeon Session, where single node performance was focused on. Fig. 7 illustrates the structure of this test system.

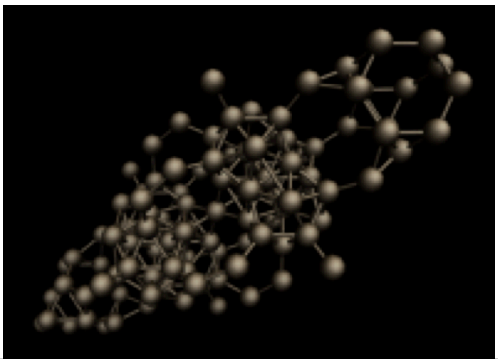

Figure 6. This benchmark test case is a system with 105 Boron atoms, 315 electrons in 216 bands, and 110592 planewaves per band. This system was used to test if the VASP hybrid functional calculations get benefit from using the HBM.

IV. HBM Analysis and Discussion

A. Estimating HBM effect on VASP performance using AutoHBW.

For the application end users, who do not modify the source code, the AutoHBW library could be a convenient tool for them to experiment with HBM performance to use the available HBM optimally on the KNL nodes. Using the AutoHBW library, we estimated the HBM performance impact on the VASP code. Fig. 8 shows the VASP run time when arrays with certain sizes were allocated to the

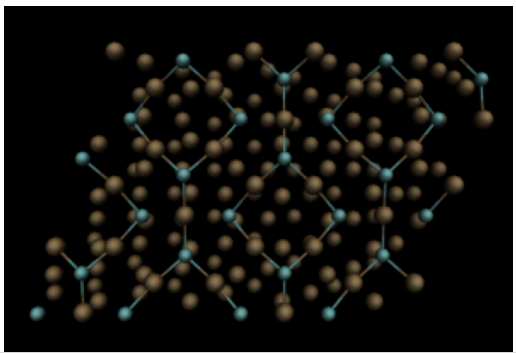

Figure 7. The benchmark test case used in profiling the MPI/OpenMP hybrid VASP code (the development version up-to 9/29/2015). The test system contains 150 Palladium (Pd) atoms and 24 Oxygen atoms (O) (denoted as PdO@Pd-slab hereafter), 1644 electrons in 1024 bands, and 33967 planewaves per band. The RMM-DIIS iteration scheme was tested, and it was executed over multiple bands simultaneously.

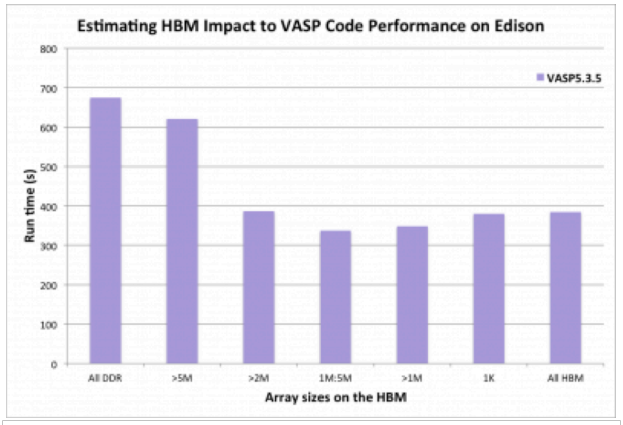

Figure 8. The simulated HBM performance impact to the VASP hybrid functional (HSE06) calculations on Edison. The production code VASP 5.3.5 (pure MPI code) was used. The tests were run with 4 Edison nodes, and had 48 tasks in total. The horizontal axis shows the size range of the arrays that were allocated to the HBM using the AutoHBW library tool, and the vertical axis shows the total run time of the VASP code. The leftmost bar (All DDR) shows the total run time when everything is allocated to the DDR memory (the far socket memory), and the rightmost bar (All HBM) shows the run time when everything is allocated to the HBM (the near socket memory). The test case used was the system containing 105 Boron atoms (B.HR105-s).

emulated HBM (the near socket memory) on Edison compute nodes. A production VASP code (version 5.3.5, pure MPI code) was used, and the memory intensive hybrid functional computation (HSE06) was tested with the test case, B.HR105-s. With only a 33% difference in bandwidths, the total run time of the VASP code was reduced by about 40% when all arrays sized from 1M to 5M were allocated to the HBM. Given the fact that the HBM on a KNL node has a bandwidth five times the size of the DDR memory, one could expect a much larger performance boost from utilizing the HBM on KNL. This experiment was very encouraging and motivated the VASP code team to look into HBM optimization during the Dungeon Session (Oct. 2015). To efficiently use the limited amount of the HBM on the KNL nodes, it is critical to allocate only the arrays that generate the most memory traffic to the HBM.

B. Indentifying HBM candidate routines in the code

To participate a Dungeon Session, developers are required to do some preparation work [14], including code profiling to select the hotspots to work on during the Dungeon Session. In the context of the HBM optimization, the developers need to determine if their applications (the hotspots) are memory bandwidth or CPU bound, as the HBM would benefit the applications only when they are memory bandwidth bound. Instead of the VTune memory access analysis, which is a proper tool to identify the arrays that generate the most memory traffic in the code, Ref. [14] recommended to run the half packed node and half clock-speed tests to determine if a code is memory or CPU bound (roughly). This was to avoid the complexity from using the VTune memory access analysis, which was still developing when the VASP Dungeon Session was held (Oct. 2015) and required some learning effort. This was also due to other limitations, such as licenses (some developers carried out the pre-dungeon preparation work on non-NERSC systems).

The profiling of the hybrid code showed a reasonable thread scaling up to 8 threads (Fig. 9), where the performance of the hybrid code matched the pure MPI code. The best performance was achieved with 4 threads per MPI task (15% speedup in comparison to the pure MPI code). Beyond 8 threads and up, the poor thread performance of the FFTs (threaded 3d cfftw from MKL), BLAS1 calls (in eddrmm), and an increase in the MPI_alltoall costs with a decreasing number of MPI ranks caused the code to stop scaling. Note that the top routines, eddiag, eddrmm, fft*, lincom and orht1, all depend heavily on the math libraries.

The run time comparison between running on the fully packed and half packed nodes (Fig. 10) showed that most of the subroutines such as FFTs (fftwav_mpi and fftext_mpi), the routines that map to the ZGEMM (lincom and orth1), and the DGEMM (rpromu and racc0mu) were likely bandwidth bound, especially the real-space projection routines rpromu and racc0mu (the bars in the blue box). They had the most run time reduction (30-35%) when running on the half-packed nodes doubled the memory bandwidth available per task. The FFTs work on contiguous data structures. The ZGEMM calls work on contiguous data structures and are blocked to reach peak performance. Unfortunately, rpromu and racc0mu access their relevant input and output data through an index table (gather/scatter), and that could be a contributing factor for the higher bandwidth demand on these two routines.

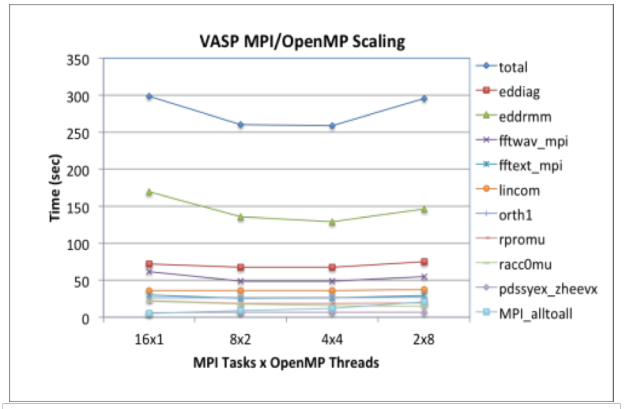

Figure 9. The thread scaling of the MPI/OpenMP VASP code with the test case PdO@Pd-slab. These are the fixed core (16 cores) tests running on one of the sockets on an Intel dual-socket Haswell node (Xeon processor E5-2697 v3 at 2.6 GHz) at the University of Vienna. The horizontal axis shows the number of MPI tasks and the OpenMP threads per tasks, and the vertical axis shows the total run time of the code (blue) and the run time breakdown for the top subroutines in the code. All run times were from the VASP internal profiler (compile VASP with the –DPROFILER preprocessor option to enable the internal profiler).

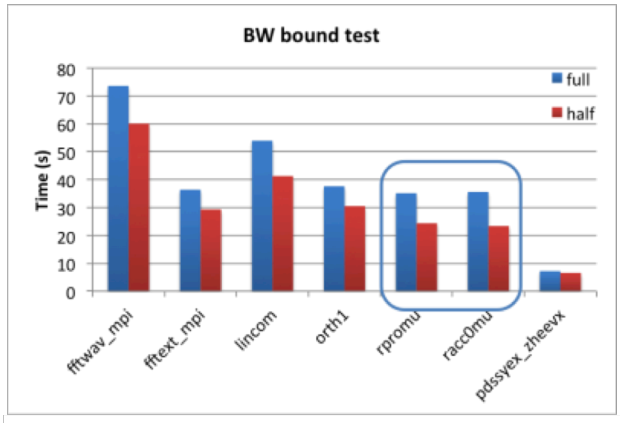

Figure 10. The VASP run time breakdown over the top subroutines when running on the fully packed (blue) and half-packed (red) nodes. The tests were done on an Intel Xeon E5-2697 v3 node and used the test case PdO@Pd-slab. When running on half packed nodes, the memory bandwidth available for each task is twice as much as it is when running on fully packed nodes. This experiment was designed to test if an application code is bandwidth bound.

The run time comparison between running on two different clock-speeds (Fig. 11) showed that the most of the subroutines were likely CPU bound as well. However, the two real-space projection routines, rpromu and racc0mu, hardly benefited from the increased clock-speed (the bars in the blue box), indicating that they are firmly memory bandwidth bound (See Fig. 10 as well). This suggests that the two routines would likely get the most performance benefit from allocation to the HBM.

C. *Allocating arrays in HBM using the Memkind library and the FASTMEM Intel compiler directive*

Based on the analysis in the previous subsection, we decided to allocate the largest arrays in the real-space projection routines RACC0MU and RPROMU in the HBM. Since the code is written in Fortran 90, we must add the Intel compiler directive FASTMEM in front of the arrays

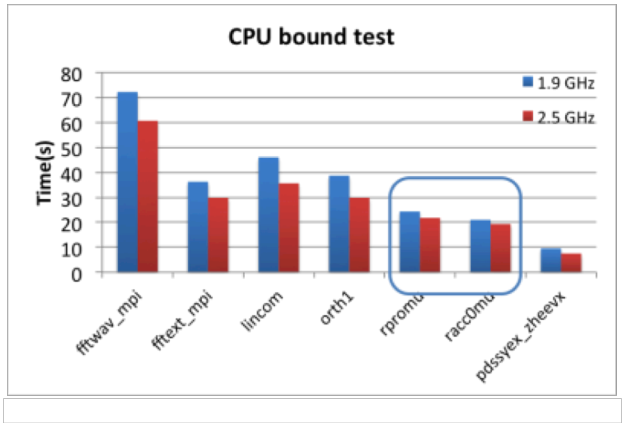

Figure 11. The VASP run time breakdown over the top subroutines when running at the clock speeds of 2.5 GHz, and 1.9 GHz on an Intel Xeon E5-2697 v3 node. The test case used was PdO@Pd-slab, and this experiment was designed to test if the code is CPU bound.

that will be allocated out of the HBM. Adding the FASTMEM directive to the code was straightforward if the arrays to be allocated in the HBM are allocatable (heap variables). For example, in the code example below (Fig. 12, upper panel), the only change needed to allocate the arrays WORK, TMP, and CPROJ to the HBM was to add a line, **!DIR$ ATTRIBUTES FASTMEM :: WORK,TMP,CPROJ**, in the code. See the lower panel in Fig. 12 for the changed code. Note the code snippets hereafter were based on the real code in VASP but were slightly modified for simplicity.

However, allocating the arrays associated with the array pointers was not straightforward. Intel compiler does not support the FASTMEM directive for the array pointers. For example, in the following code example (Fig. 13, upper panel), to allocate the arrays pointed by the three array pointers, NLI, RPROJ, and CRREXP to the HBM, we had to introduce three allocatable work arrays, fm_NLI, fm_RPROJ and fm_CRREXP, allocate them to the HBM by adding the !DIR$ ATTRIBUTES FASTMEM directive in front of them, and then associate the array pointers to these work arrays, respectively. See the lower panel in Fig. 13 for the changed code.

After the FASTMEM directive was added to the code, it was compiled and run on Edison. Fig. 14 (upper panel) shows the comparison between the total run time when everything was allocated in the DDR memory (blue:All DDR), when only a few selected arrays were allocated to the HBM (red:Mixed), and when all arrays were allocated to the

```
SUBROUTINE RACC0MU(NONLR_S, WDES1,
CPROJ_LOC, CRACC, LD, NSIM, LDO)
...
REAL(qn),ALLOCATABLE:: WORK(:),TMP(:,:)
GDEF,ALLOCATABLE    :: CPROJ(:,:)
...
ALLOCATE(WORK(ndata*NSIM*NONLR_S%IRMAX),TMP
(NLM,ndata*2*NSIM),CPROJ(WDES1%NPRO_TOT,NSIM))
...
END SUBROUTINE RACC0MU
```

```
SUBROUTINE RACC0MU(NONLR_S, WDES1,
CPROJ_LOC, CRACC, LD, NSIM, LDO)
...
REAL(qn),ALLOCATABLE:: WORK(:),TMP(:,:)
GDEF,ALLOCATABLE    :: CPROJ(:,:)

!To allocate WORK,TMP, CPROJ to HBM
!DIR$ ATTRIBUTES FASTMEM :: WORK,TMP,CPROJ
...
ALLOCATE(WORK(ndata*NSIM*NONLR_S%IRMAX),TMP
(NLM,ndata*2*NSIM),CPROJ(WDES1%NPRO_TOT,NSIM))
...
END SUBROUTINE RACC0MU
```

Figure 12. The upper panel shows the original code before adding the FASTMEM directive; the lower panel shows the code after adding the FASTMEM directive.

```
INTEGER, POINTER :: NLI (:,:) ! index for gridpoints
REAL(qn),POINTER, CONTIGUOUS :: RPROJ (:) ! projectors
on real space grid
COMPLEX(q),POINTER, CONTIGUOUS::CRREXP(:,:,:) !
phase factor exp (i k (R(ion)-r(grid)))

...

ALLOCATE(NONLR_S%NLI(IRMAX,NIONS),NONLR_S%R
PROJ(NONLR_S%IRALLOC))
ALLOCATE(NONLR_S%CRREXP(IRMAX,NIONS,1))
```

```
INTEGER, POINTER :: NLI (:,:) ! index for gridpoints
REAL(qn),POINTER, CONTIGUOUS :: RPROJ (:) ! projectors
on real space grid
COMPLEX(q),POINTER, CONTIGUOUS::CRREXP(:,:,:) !
phase factor exp (i k (R(ion)-r(grid)))

! To exploit fastmem, introduce extra allocatable work arrays
INTEGER, ALLOCATABLE :: fm_NLI (:,:) ! index for
gridpoints
REAL(qn), ALLOCATABLE :: fm_RPROJ (:) ! projectors on
real space grid
COMPLEX(q), ALLOCATABLE :: fm_CRREXP(:,:,:) !
phase factor exp (i k (R(ion)-r(grid)))

!add FASTMEM directive
!DIR$ ATTRIBUTES FASTMEM ::
fm_RPROJ,fm_CRREXP,fm_NLI

...

! allocate work arrays to HBM
ALLOCATE(NONLR_S%fm_NLI(IRMAX,NIONS),NONL
R_S%fm_RPROJ(NONLR_S%IRALLOC))
ALLOCATE(NONLR_S%fm_CRREXP(IRMAX,NIONS,1))

!associate arrays pointers to the work arrays
NONLR_S%NLI =>NONLR_S%fm_NLI
NONLR_S%RPROJ=>NONLR_S%fm_RPROJ
NONLR_S%CRREXP=>NONLR_S%fm_CRREXP
```

Figure 13. The upper panel shows the original code before adding the FASTMEM directive; the lower panel shows the code after adding the FASTMEM directive. In this code snippet, the arrays associated to the array pointers, NLI, RPROJ, and CRREXP, will be allocated to the HBM.

HBM (green:All HBM). About 9% of speedup in the total run time was achieved when a few selected arrays were allocated to the HBM in comparison to when everything was allocated in the DDR memory, while about 14% of speedup was achieved when allocating everything out of the HBM. Further looks into the run time breakdown of the two real space projection routines (middle and lower panels) revealed that allocating only a few selected arrays in the HBM (red:Mixed) achieved the same level of speedup as allocating everything to the HBM (23-26%). In fact as one can see from the lower panel (RPROMU) results, allocating only a few selected arrays to the HBM (Mixed runs, red bars) outperforms allocating everything in the HBM (green bars). (Note this is a reproducible result, not due to the run

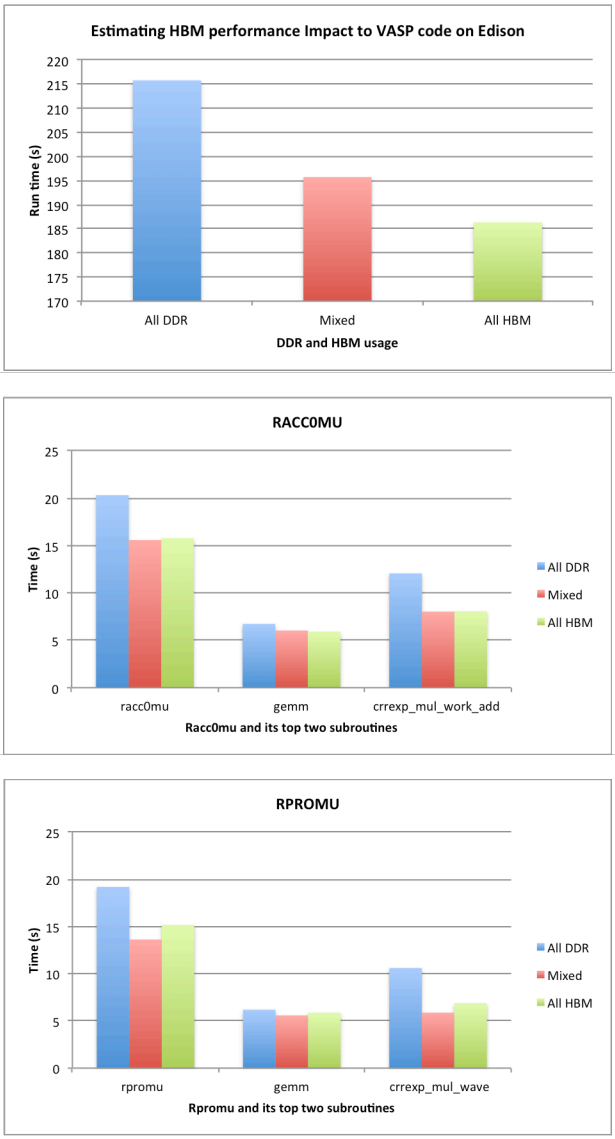

Figure 14. VASP performance comparison when everything was allocated in the DDR memory (blue: All DDR), when only a few selected arrays were allocated in the HBM (red: Mixed), and when everything was allocated to the HBM (green: All HBM). The upper panel shows the comparison of the total run time. The middle and lower panels show the run time of the real space projection routines, RACC0MU, and RPROMU, and the top two subroutines, respectively. The tests were run on a single socket on an Edison node with two MPI tasks and six threads per task. Al time shown here were from the VASP internal profiler.

time variation). This should be related to the increased memory bandwidth from using both sockets on the node for the Mixed runs while the All DDR and All HBM runs used the memory from a single socket only on the dual-socket Edison node. This test case (PdO@Pd-slab) had a modest memory bandwidth usage. One could expect a further performance boost for the systems with higher memory bandwidth need. Given the fact that the HBM on a KNL

node has a bandwidth five times the size of the DDR memory, one could expect a larger performance boost from utilizing the HBM on KNL for the systems that have high memory bandwidth usage. To achieve a level of speedup of allocating everything in HBM (green bars) the VTune memory access analysis could be used to identify the arrays that generate most memory traffic, and allocate them to the HBM. After the Dungeon Session, the developers (Marsman) have added the FASTMEM directive to the rest of the codes. As far as the HBM is concerned, VASP code is ready to exploit KNL.

Note that the Memkind (also AutoHBW) library is not capable of allocating the stack arrays to the HBM, such as the automatic arrays in Fortran and the OpenMP private arrays. Therefore one has to change them to the allocatable arrays prior to place them to the HBM. However, changing arrays that are placed in the stack to the heap so to use the HBM may slowdown the memory accessing speed significantly in some cases. These are the limitations of the Memkind and AutoHBW methods (and Intel compilers) currently. Hopefully, Intel could address these issues in the near future.

V. Conclusions and Future Work

We simulated and analyzed the performance impact of the HBM (MCDRAM), which will be available on the future KNL architecture, on the VASP code, a commonly used materials science code, using the Memkind and AutoHBW tools and using the dual-socket Ivy Bridge nodes as the HBM proxy on KNL. With only 33% difference in bandwidths between the emulated HBM (near socket memory) and the DDR (far socket memory), we have observed up to 40% performance boost when using the HBM. Our analyses show that the HBM on KNL may have a significant performance benefit to applications. Identifying an application is memory bound or CPU bound is the first step to the HBM optimizations. The run time comparison between running an application on the fully and half packed nodes could be used to tell roughly if an application is memory bound or not. Selectively allocating arrays on the HBM is a key optimization tactic to use the small amount of the available HBM efficiently on KNL nodes. The VTune memory-access analysis is useful to identify the candidate arrays for HBM. Once the candidate arrays are identified, using Intel compiler directive, FASTMEM, or using Memkind APIs, one can selectively allocate those arrays to the HBM. For the application end users who rarely change the source codes, the AutoHBW tool could be used to achieve the optimal use of the HBM conveniently. Early adoption of the Memkind and AutoHBW tools is key to get applications ready for KNL as far as the HBM is concerned.

Using Edison Ivy Bridge nodes as the HBM proxy to estimate the HBM performance is not an exact analogy to the HBM on KNL, however, the approach used in this study will be applicable for KNL without modifications. Using Memkind to do the HBM optimizations still have some limitations and optimization portability concerns. Applications developers and end users rely on Intel and other compiler vendors to mitigate and/or resolve these issues in the near future.

Acknowledgement

The work presented in this paper was based on the VASP Dungeon Session held in Oct. 2015. Many Intel engineers and NERSC staff provided valuable insights for this work. Authors would like to thank Jeongnim Kim, Martyn Corden, Christopher Cantalupo, Sumedh Naik, Gregory Junker, Ruchira Sasanka and other engineers at Intel, and also Jack Deslippe at NERSC. This work was supported by the ASCAR Office in the DOE, Office of Science, under contract number DE-AC02-05CH11231. It used the resources of National Energy Scientific Computing Center (NERSC) and the computing resource at University of Vienna.

References

[1] https://www.nersc.gov/assets/Uploads/Preparing-Software-for-KNL-ISC15-IXPUG-Keynote.pdf

[2] http://www.nersc.gov/users/computational-systems/cori/

[3] http://portal.nersc.gov/project/mpccc/baustin/NERSC_2014_Workload_Analysis_30Oct2015.pdf

[4] http://www.nersc.gov/nesap

[5] **http://memkind.github.io/memkind**

[6] **http://memkind.github.io/memkind/examples/autohbw_README**

[7] **http://www.vasp.at/**

[8] G. Kresse and J. Furthm_ller. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. Comput. Mat. Sci., 6:15, 1996

[9] http://www.nersc.gov/users/computational-systems/edison/

[10] https://github.com/jemalloc/

[11] Luis DeRose, *"The Cray Programming Environment: Current Status and Future Directions"*, 2016 Cray User Group Meeting, 5-8-12, 2016, England UK.

[12] http://www.nersc.gov/users/computational-systems/edison/cluster-compatibility-mode/

[13] http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/using-high-performance-libraries-and-tools

[14] http://www.nersc.gov/users/computational-systems/cori/application-porting-and-performance/dungeon-session-worksheet/