Interactive Code Adaptation Tool for Modernizing Applications for Intel Knights Landing Processors

Ritu Arora
Texas Advanced Computing Center
The University of Texas at Austin
rauta@tacc.utexas.edu

Lars Koesterke
Texas Advanced Computing Center
The University of Texas at Austin
lars@tacc.utexas.edu

ABSTRACT

The process of code adaptation to take advantage of the latest innovations in a supercomputing platform begins with learning about the details of the platform's underlying hardware. It can be challenging for many users to spend time and effort in developing an understanding of the innovative features in a supercomputing platform - such as deep memory hierarchies - and to harness their maximum possible performance by manually modernizing their applications. To mitigate the aforementioned challenge, we are developing an Interactive Code Adaptation Tool (ICAT). In its current form, ICAT can assist the users in modifying, compiling, and optimally running their applications on the latest HPC platforms that are equipped with the Intel Knights Landing (KNL) processors. ICAT detects a given application's characteristics such as memory usage pattern, type of memory allocation, and execution time. Depending upon the application's characteristics, it advises the user on optimal ways to take advantage of the KNL processor and its memory-hierarchy.

CCS CONCEPTS

• Computing methodologies~Parallel computing methodologies~Parallel programming languages • Software and its engineering~Source code generation • Computer systems organization~Architectures ~Parallel architectures~ Multicore architectures

PEARC17, July 09-13, 2017, New Orleans, LA, USA
 ACM 978-1-4503-5272-7/17/07…$15.00
http://dx.doi.org/10.1145/3093338.3093352

KEYWORDS

Code modernization, code migration, interactive code adaptation, memory optimization, advanced vectorization, clustering advisor, memory advisor, Intel Knights Landing

1 INTRODUCTION

Intel's manycore Knights Landing (KNL) processors have high-bandwidth memory called Multi-Channel DRAM (MCDRAM) on the package in addition to the usual DDR4 SDRAM. On the self-bootable version of the KNL processors, the theoretical peak bandwidth of MCDRAM is approximately 4.5 times higher than that of the DDR4 memory [1]. On the flip side, the MCDRAM is smaller in size - 16 GB compared to up to 400 GB of DDR4. MCDRAM can improve the performance of applications with a low floating-point intensity (low ratio of FLOPs v. memory access). Such applications are typically labeled as bandwidth-critical or bandwidth-bound. However, using MCDRAM optimally is not a straightforward task for many applications because of its limited size and a variety of memory and cluster modes (explained below) in which the MCDRAM can be configured. Depending upon an application's characteristics, a particular configuration mode may work better than the other modes.

Memory Modes: The three memory modes for the MCDRAM are named cache, flat, and hybrid. In cache mode the MCDRAM serves as a third-level cache. In flat mode the MCDRAM is addressable memory like DDR4. In hybrid mode, part of the MCDRAM is configured as a cache and a part of it is configured in flat mode.

The main difference between the flat and the cache mode is that the MCDRAM in flat mode is user-addressable while the MCDRAM in cache mode is not. As obvious, the cache mode is more convenient to use since it operates without user interaction, and, for applications with a small memory footprint, the cache mode usually provides high performance. However, for applications with large memory footprints, the efficiency of the cache can drop dramatically

(due to the frequent cache misses) and in those cases it may be advantageous to manage the cache from within the code and to hand-select arrays to be stored in the MCDRAM. Using the MCDRAM configured in hybrid mode gives the advantages of both cache mode and the flat mode, but results in smaller sizes of MCDRAM available in each of these modes.

Cluster Modes: Pairs of cores on a KNL chip are organized into tiles. Each core has its own L1 cache and each tile provides a shared L2 cache. The tiles are connected to each other with a cache-coherent mesh interconnect. There is a Distributed Tag Directory (DTD) to maintain coherency across L2 cache on all tiles. This DTD is organized as a set of Tag Directories (TDs) on each tile and is used to identify the state and the location of cache lines. The mesh interconnect supports three modes of cluster operations to keep the on-die communication - for handling memory requests originating from cores, forwarded to the TD, and then serviced by the right memory channel - as local as possible. These modes are named all-to-all, quadrant or hemisphere, and sub-NUMA cluster mode (SNC-4/SNC-2) [2, 3].

In the all-to-all cluster mode, the memory addresses are distributed uniformly across all the TDs, hence, it can suffer from a high-latency of cache miss and hit. This mode is mostly used for diagnostic purposes.

In the quadrant cluster mode, the tiles of the KNL package are divided into four parts called quadrants such that each quadrant is in proximity to a memory controller. The memory addresses that are controlled by the memory controller in each quadrant are mapped locally to the TDs in that quadrant. This arrangement reduces the latency of a cache miss as compared to the all-to-all mode because the memory controller and TD's are in the same locality, and thus, there is no need to go across the quadrants. The hemisphere mode is similar to the quadrant mode with the difference that the tiles on the chip are divided into two parts.

In the sub-NUMA mode, just like the quadrant or hemisphere mode, the tiles are divided into four or two parts. However, each part acts as a separate NUMA node such that, the core requesting access to memory, the TD, and the memory channel for servicing the memory access request, are all in the same part (quadrant or hemisphere). The multi-threaded NUMA-aware applications can experience improved performance in this mode by pinning the threads and memory to the specific quadrants or hemisphere on each NUMA node.

Given the different memory and the cluster modes for the MCDRAM configuration, the burden is on the software developers to find the modes that will work best for their applications. There are some default recommendations by Intel that may work well for cache-friendly applications. For all other applications, the process of finding the best memory and cluster mode begins with developing an understanding of the architecture of the KNL processors. It may also involve using tools like Vtune [4] for understanding the application characteristics. Manual reengineering of the code may be necessary for improving the parallelization and vectorization of the code. And in flat mode, code modifications to direct memory allocations to the MCDRAM are also required.

To facilitate the migration of applications to the KNL and future generations architectures, we are developing an Interactive Code Adaptation Tool (ICAT). ICAT can analyze a user-supplied serial or parallel application using built-in heuristics, and tools like Vtune and `perf` [5]. Based on its analysis, ICAT can advise the user on modifying, compiling, and optimally running an application on the KNL nodes. If the user desires, ICAT can automatically modify the application code to use MCDRAM for specific arrays.

Code analysis and modification regarding MCDRAM is explained in Section 2 of the paper. Additional details on the decision-trees used by ICAT to provide recommendations to the users are presented in Section 3. The techniques for memory optimization that will soon be part of ICAT are discussed in Section 4. Advanced vectorization support will also be added to ICAT in future and it is discussed in Section 5. The usage of ICAT is demonstrated with an example in Section 6 of the paper.

2 CODE ADAPTATION

As mentioned in Section 1, MCDRAM is a smaller high-bandwidth memory compared to DDR4. Applications that fit in MCDRAM will likely not benefit from code modifications and can be run in cache mode or in flat mode using MCDRAM (in the latter case select appropriate `numactl` options). Applications that require more than 16GB of memory will not fit entirely in the MCDRAM. Users may rely on the MCDRAM cache and may measure the cache performance by comparing data from tests with varying memory footprints. If the performance degradation is high, which is likely for footprints much larger than the 16GB of MCDRAM cache, the application developers may use tools like Vtune to identify the bandwidth-critical data structures. The bandwidth-critical data structures can be selectively allocated on MCDRAM with or without any application reengineering.

The autoHBW library [6] can be used for allocating all the data structures beyond a particular memory size on MCDRAM without involving any reengineering of the applications or recompilation. Another option that the users can explore for allocating memory on MCDRAM or DDR4 is to use appropriate `numactl` command options [7].

However, if a fine-grained control on memory allocation is needed such that out of the multiple data structures of the same size, only a few should be allocated on MCDRAM, then one can either use the memkind interface [8] (which is basically a heap manager for enabling allocation to specific types of memory), or one can use a simplified version of the memkind interface that is known as High-Bandwidth Memory ALLOCator (HBWMALLOC) interface [9]. Using these interfaces would entail application reengineering to insert appropriate API calls in C/C++ code or directives in Fortran code.

In C/C++ applications, the application reengineering to use the HBWMALLOC interface is usually straightforward. The calls to the `calloc`, `malloc`, `realloc`, and `free` functions are replaced with the calls to the `hbw_calloc`, `hbw_malloc`, `hbw_realloc`, and `hbw_free` functions that are defined in the HBWMALLOC interface. The signature of the functions in the HBWMALLOC interface is same as their analogs in the standard C library named `stdlib.h`. In Fortran applications, the reengineering effort usually involves adding a directive with the `FASTMEM` attribute for allocating the memory from MCDRAM after the allocatable data structure of interest has been declared.

The code snippets in Figure 1 show the method for explicitly allocating a data structure on MCDRAM. Line # 4 in Figure 1.(I) has a call to `malloc` that allocates memory for the array named `a` from the heap on DDR4. Line # 4 in Figure 1.(II) has the call to `hbw_malloc` for allocating memory for the array named `a` on MCDRAM. For portability reasons, one should write the code to make sure that the calls to the HBWMALLOC interface are used only if the MCDRAM is available in the underlying HPC system. The availability of MCDRAM can be queried with the `hbw_check_available` function [9]. If the MCDRAM is not present, then the usual calls for memory allocation should be used.

As can be noticed from Figure 1, the reengineering step itself is not difficult once the bandwidth-critical data structures are identified. However, *the combined effort involved in (1) understanding the KNL architecture and the various modes in which the MCDRAM can be configured, (2) identifying the bandwidth-critical data structures, (3) deciding which data structures to allocate on MCDRAM, and then, if needed, (4) adapting the code to use the HBWMALLOC interface, is not trivial.* ICAT automates all these steps for the user and hence, assists them in migrating their applications to the KNL architecture.

```
1.  #include <stdlib.h>
2.  int main(){
3.   int arraySize = 1024;
4.   double* a = (double*)
     malloc(sizeof(double)*arraySize);
5.   //other code
6.   free(a);
7.   return 0;
8.  }
```

(I) C code snippet - allocating memory on DDR4

```
1.  #include <hbwmalloc.h>
2.  int main(){
3.   int arraySize = 1024;
4.   double* a = (double*)
     hbw_malloc(sizeof(double)*arraySize);
5.   //other code
6.   hbw_free(a);
7.   return 0;
8.  }
```

(II) C code snippet - using HBWMALLOC interface

```
REAL, ALLOCATABLE :: arryA(:), arryB(:)
! arryA is allocated in DDR4
ALLOCATE (arryA(1:2048))
! arryB is allocated in DDR4
ALLOCATE (arryB(1:2048))
```

(III) Fortran code snippet - allocating memory on DDR4

```
REAL, ALLOCATABLE :: arryA(:), arryB(:)
!DEC$ ATTRIBUTES FASTMEM :: arryA
! arryA is allocated in HBM
ALLOCATE (arryA(1:2048))
! arryB is allocated in DDR4
ALLOCATE (arryB(1:2048))
```

(IV) Fortran code snippet - allocating memory on MCDRAM

Figure 1.(I) - (IV). Code snippets showing the usage of `hbw_*` calls and directives for using MCDRAM

As mentioned above, for assuring code maintainability and portability across multiple systems, it is important that the code modifications for taking advantage of the MCDRAM on the KNL nodes do not break the portability of the code. Therefore, instead of merely replacing the calls to malloc/calloc/realloc with the corresponding `hbw_*` calls, ICAT also inserts extra checks in the code so that if the MCDRAM is not available on the system on which the code is being compiled and run, the memory allocation happens using `malloc`/`calloc`/`realloc` calls instead of the `hbw_malloc`/`hbw_calloc`/`hbw_realloc` calls.

3 DECISION-MAKING IN ICAT

There are multiple decision trees that are part of ICAT for recommending the appropriate memory mode, cluster mode, and code adaptation. Some of the application characteristics used for decision-making are L2 cache hit bound (ratio of cycles spent handling L2 hits to all cycles), L2 cache hit rate, L2 cache miss bound (ratio of cycles spent handling L2 misses to all cycles), and L2 cache miss count. These metrics are determined by first running the application with `perf`, and then if needed, with Vtune as well. Metrics are also gathered from the process that is associated with the application. ICAT uses the collected metrics to prepare recommendation reports for the user and also assists in code adaptation. Figure 2 shows a high-level overview of how ICAT works.

If ICAT determines that the memory footprint of an application is smaller than 16 GB, then it recommends to use either one of the flat or cache modes. While recommending flat mode, ICAT also provides appropriate `numactl` options. If the application is appropriately blocked (or tiled) to take advantage of the L2 cache and its memory footprint is smaller than 16 GB, then ICAT recommends ignoring the MCDRAM and using the DDR4. This recommendation is based on the fact that the DDR4 latency is lower than the MCDRAM latency. However, in some situations, ICAT also recommends testing the application by running it on MCDRAM configured in cache mode.

For memory footprints larger than 16 GB, ICAT recommends one of the following options depending on the results of the internal analysis: :

- Use MCDRAM configured in cache mode
- Use MCDRAM configured in flat mode and use numactl to prefer MCDRAM: `numactl –preferred=1 a.out`
- Use MCDRAM configured in flat mode and allocate selected bandwidth-critical data structures on the MCDRAM

ICAT also helps users in making their applications NUMA-aware. It gives recommendations to (1) establish MPI domains, (2) pinning threads in OpenMP code, or (3) both the previous two recommendations in hybrid code. The selection of the cluster mode is affected by this step as well. For example, using the sub-NUMA quadrant mode is only applicable if at least four MPI tasks (per KNL node) are used. However, if it is not required to make the application NUMA-aware, ICAT recommends to use the quadrant mode by default.

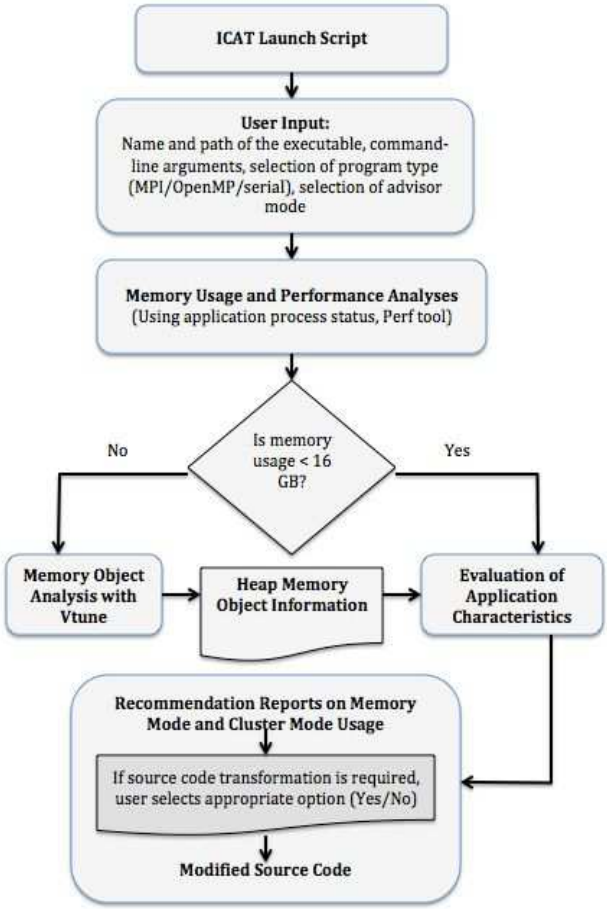

Figure 2: High-Level overview of ICAT

At the time of code adaptation, ICAT inserts additional lines of code to determine the size of data structures that are dynamically allocated. It also inserts conditional statements that can determine if the size of the data to be allocated dynamically is greater than 16 GB or not. If the size is greater than 16 GB, then the standard library calls like `malloc`, `calloc`, or `realloc` are used for data allocation. However, if the size of the data to be allocated is less than 16 GB, then the inserted code triggers the usage of the HBWMALLOC interface. Additionally ICAT inserts code to dynamically check the availability of MCDRAM and to make calls to the HBWMALLOC interface if applicable.

4 MEMORY OPTIMIZATION

On all the computing platforms that exist today, there is a significant gap between the performance of the processing elements and the main memory. To mitigate the effect of the high-latency and low-bandwidth of the main memory, deep memory hierarchies are provisioned on the HPC platforms. Optimizing memory access in an application continues to be important for achieving good performance. Several researchers have demonstrated that application performance can be improved by reducing memory access and by improving cache reuse [10-14]. In the case of nested loops, one commonly used strategy for optimizing cache and register reuse is to interchange the ordering of the loops where possible to reduce the stride of memory access [14]. By reducing the stride of memory access, the applications can utilize more data from each cache line that is transferred from or to memory.

Another useful strategy for cache reuse is blocking or tiling the loops. With this strategy, a single loop can be replaced by a pair of loops such that, the inner loop in the pair iterates over a block of the iteration in the original loop while using the same loop stride that was specified in the original loop. However, the outer loop in the pair iterates with a stride that is equal to the size of the block iterated over by the inner loop.

Some optimizations that we have discussed so far can be done either automatically by the compiler or explicitly by the programmer. However, there are additional types of optimizations that are too complex for the compiler. Some examples of such transformations are: reorganization of the data layout by changing array-of-structures to structure-of-arrays, complicated forms of cache-blocking, converting data access from main memory to on-the-fly recalculation, allocating and initializing memory close to the place in the application where it is used, and deallocating memory soon after its last use [11]. These memory optimization strategies will be part of ICAT in future.

5 ADVANCED VECTORIZATION ADVISOR

Modern compilers automatically vectorize loops if they recognize an opportunity to use SIMD instructions. Programmers can also expose the vectorization opportunity by using specific code annotations (*e.g.*, `#pragma vector always`).

However, there are loops that cannot be vectorized by the compilers due to data-dependencies - real or assumed by the compiler. Having switch statements, pointer aliasing, certain types of if-statements, and function calls in a loop can also prevent it from vectorizing. Non-contiguous memory accesses with a non-unit stride or indirect addressing also creates difficulties for the compiler [15-16]. There are also situations in which the static-code analysis by the compiler can be inconclusive. In such situations, the developers can add annotations in their code to provide guidance to the compiler.

ICAT will be extended to allow users to interactively add vectorization related advice to their code. For example, it can advise the developers to add the following hint before the loops that should not be vectorized: `#pragma novector`, or add the following hint before the loops in which the compiler should ignore any assumed dependencies: `#pragma ivdep`. Restructuring the existing code for explicitly prefetching data, inlining small functions, SIMD enabled functions, scalar-to-vector conversions, and improving memory alignment of data structures will also be supported by ICAT.

6 ICAT IN ACTION

ICAT is a command-line tool that is invoked as shown in Figure 3. Before invoking ICAT, the application that needs to be ported to KNL must be compiled with the "-g" flag. As shown in Figure 3, ICAT prompts the user to select one or all of the following options: memory mode advisor, cluster mode advisor, vectorization advisor, code adaptation advisor, and memory optimization advisor.

```
c561-052$ bash -i ./icat.sh
--------------------------------------------------------------------------
-------- Welcome to ICAT :: Interactive Code Adaptation Tool --------
--------------------------------------------------------------------------

Step 1
Purpose         : Acknowledge usage of compiler option '-h'
Question        : Please acknowledge that you have compiled the code with the '-g' option

                  Answer with y/n (y is the default) :: y
                  You have answered with             :: y

Step 2
Purpose         : Provide the name of the executable, the path, and optionally the program a
Question        : Name of the executable?                       testProgram
                  Path to the executable? You may use . (dot)   .
                  Command line arguments, separated by commas?


Step 3
Purpose         : Select advice topic
Question        : Please select from one of these options

Option          : Advice                Description
-----------------------------------------------------
1               : Memory mode           Exploit memory hierarchies
2               : Code adaptation       Assign individual arrays to different memory types
3               : Cluster mode          Exploit clustering of cores
4               : Vectorization mode    Enable vector instructions
5               : Memory optimization   Is this the AoS to SoA transformation?
6               : All                   Get all available advice at once

0               : Quit ICAT


                  Answer with a number between 0 and 6 (0 is the default)       :: 6
                  You have selected option                                      :: 6
```

Figure 3: Invoking ICAT

In the scenario shown in Figure 3, the user chooses option number "**6**" to run all advisors. As a first step, ICAT runs the memory mode advisor. It prompts the user for the name of the application executable that was compiled with "-g" flag and the complete path to it. Without any user interaction ICAT then profiles the application using `perf`,

and if needed, with Vtune as well. After analyzing the memory usage characteristics of an application, ICAT generates a `reports` directory and saves the recommendations in this directory. A sample report is presented in Figure 4 and it shows that ICAT provides condensed information about the memory usage pattern of the application. It suggests the memory mode in which the application should be run and the appropriate commands to start the application.

```
ICAT will determine best memory and
clustering modes and will check for
vectorization opportunities
Please enter the executable name (only):
circuit_serial
Please enter complete path to folder
containing executable:
/home1/01698/rauta/sample_code
Please enter any program arguments:

Profiling program...

Report generated.
----------------------
```

Figure 4: Specifying the input program name and the path to it

```
c561-003$ cat
reports/circuit_serial_memory_advisor_repo
rt.txt
--- circuit_serial Characteristics ---
Memory usage: 0.120438
Cache Miss Rate: -0

----- Recommendations -----
Application fits into HBM.

Mode to use: If numactl is available, use
the Flat-Mode with all allocations to HBM.

If numactl is not available, then use the
Cache-Mode. However, note that the cache
misses in the Cache-Mode are more
expensive than reading data from DDR4 in
Flat-Mode.

Memory Allocation: HBM

To execute the application in Flat-Mode:
Use command < numactl --membind=1
./run-app> if it is serial, or < ibrun
--membind=1 ./run-app > if it is parallel.

To execute the application in Cache-Mode:
Use the command that you normally use,
that is, < ./run-app > if it is serial or
< ibrun ./run-app > if it is parallel.

In general, to determine <NUMA_NODE> in
the command < numactl --membind=NUMA_NODE
>, run the command < numactl -H > and look
for the node that does not have any cores

--End ICAT report for circuit_serial--
```

Figure 5: Sample memory mode recommendation

On the basis of the memory usage information ICAT performs further analysis to suggest an appropriate cluster mode for the application. As shown in Figure 6, it prompts the user for information on the parallel programming model used by the application, and generates a report with advice on the cluster mode to use. A sample cluster mode report is shown in Figure 7. In this example ICAT recommends the quadrant mode.

```
Profiling program...
What is the programming model used in your
application?
1. OpenMP
2. MPI
3. OpenMP+MPI
4. None of the above/serial
4
Report generated.
----------------------

OTHER INFORMATION NOT INCLUDED HERE
----------------------
```

Figure 6: Specifying the input for the cluster mode advisor

The code adaptation advisor checks the report generated by the memory advisor to detect if any code adaptation might be needed or not. For our example application, as shown in Figure 8, no code adaptation is recommended.

```
c561-003$ cat
reports/circuit_serial_clustering_advisor_
report.txt

--- circuit_serial Recommendations ---

 Clustering mode to use: Quadrant

--End ICAT report for circuit_serial--
```

Figure 7: Sample cluster mode recommendation

```
Either the source code modification is not
needed or the Memory Advisor report for
circuit_serial does not exist in the
subdirectory named reports. However, if
you would like to test how our source code
modification script works, press 2, else
press 3.
3
Please choose from the following ICAT
options:
1. Run Memory Mode Advisor
2. Run Cluster Mode Advisor
3. Run Vectorization Advisor
4. Run Code Adaptation Advisor
5. Run Memory Optimization Advisor
6. All
7. Quit ICAT
7
```

Figure 8: Code adaptation advisor

To demonstrate how the code adaptation advisor changes the input code, let us consider a different and trivial example that is shown in Figure 9. If the memory allocation for the arrays named **`buffer`** and **`duffer`** should be done on MCDRAM, then the `malloc` calls for allocating memory for them should be replaced with the calls to the function `hbw_malloc`. The calls to function `free` should be replaced with calls to the function `hbw_free`, and the header file for the HBWMALLOC interface should be included in the code. It is also advised to insert appropriate checks (conditional statements) to ensure the availability of MCDRAM by calling the `hbw_check_available` function. ICAT can make all the aforementioned modifications or insertions in the code. The steps for modifying code using ICAT and the resulting code snippet are shown in Figure 10. ICAT also inserts code to ensure that the size of memory allocated from MCDRAM is less than or equal to 16 GB. At present, the vectorization and memory optimization advisors are under development.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (){
  int i, n, *buffer, *duffer;
  i=4816000;
  buffer = (int*)malloc(i*sizeof(int));
  duffer = (int*)malloc(i*sizeof(int));
  for (n=0; n<i; n++)
    buffer[n]=rand()%26 + 100;
  for (n=0; n<i; n++)
    buffer[n]=rand()%26 + 120;
  usleep(1000);
  for (n=0; n<i; n++)
    duffer[n]=buffer[n]+(buffer[n]+n);
  free(buffer);
  free(duffer);
  return 0;
}
```

Figure 9: Sample code for adaptation

7 CONCLUSION

In this paper we presented a high-level tool named ICAT that assists users to take advantage of the MCDRAM on KNL nodes. ICAT internally utilizes Vtune and `perf` to analyze an application provided by the user. ICAT guides the users to select the most efficient memory and cluster modes. It also helps with reengineering source code to optimally utilize the MCDRAM for allocating bandwidth-critical data structures. In the future versions of ICAT, support for memory optimization and advanced vectorization will be added.

While there are ongoing efforts to provision the middleware for managing multiple-levels of memory hierarchies, higher level tools like ICAT have the potential for helping users in efficiently taking advantage of the memory hierarchies without feeling overwhelmed with their low-level architectural details, and without learning multiple tools like Vtune, and `perf`.

```
Code Adaptation Advisor for Using HBM.
...
Line # '7' of your program contains a call
to malloc that can be replaced with a call
to hbw_malloc to take advantage of MCDRAM.
Do you want to replace the calls to
malloc?(type y for yes, and n for no)y

Lines with call to function free that is
paired with the recently replaced malloc
call, will be replaced with hbw_free.
…
Program modification is complete. The
modified program is presented below.
#include <hbwmalloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (){
  int i, n, *buffer, *duffer; i=4816000;
  int checkHBMAvailability =
     hbw_check_available();
  int array_size = (i *sizeof(int));
  if (checkHBMAvailability == 0 &&
     array_size < 17179869185){
     buffer = (int*)hbw_malloc (i
                        *sizeof(int));
  } else{
     buffer = (int*)malloc(i*sizeof(int));
  }
 //other code-memory allocation for duffer
  for (n=0; n<i; n++)
    buffer[n]=rand()%26 + 100;
  for (n=0; n<i; n++)
    buffer[n]=rand()%26 + 120;
  usleep(1000);
  for (n=0; n<i; n++)
    duffer[n]=buffer[n]+(buffer[n]+n);
  if (checkHBMAvailability == 0 &&
                 array_size < 17179869185){
     hbw_free(buffer);
  } else{   free(buffer); }
 //other code-memory allocation for duffer
  return 0;
}
```

Figure 10: Code updating using ICAT

ACKNOWLEDGMENTS

We are grateful to Tiffany Connors for her contributions to the ICAT codebase. We are very grateful to the National Science Foundation for grant # 1642396, ICERT REU program (National Science Foundation grant # 1359304), XSEDE (National Science Foundation grant # ACI-1053575), and TACC for providing resources required for this project.

REFERENCES

[1] MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing Processors: Developer's Guide. 2016. Accessed on March 10th 2017: https://colfaxresearch.com/knl-mcdram/

[2] Clustering Modes in Knights Landing Processors. 2016. Accessed on March 10th 2017: https://colfaxresearch.com/knl-numa/

[3] James Jeffers, James Reinders, and Avinash Sodani. 2016. Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2. Elsevier Science & Technology Books, 662. ISBN: 0128091940, 9780128091944

[4] Vtune Performance Profiler. 2017. Accessed on March 10th, 2017: https://software.intel.com/sites/default/files/managed/d7/ba/intel-vtune-amplifier-2017-product-brief.pdf

[5] perf: Linux Profiling with Performance Counters. Accessed on March 10th, 2017: https://perf.wiki.kernel.org/index.php/Main_Page

[6] Sunny G. Using The AutoHBW Library with Jemalloc and Memkind. 2015. Accessed on March 10th, 2017: https://software.intel.com/en-us/articles/using-autohbw-with-jemalloc-and-memkind-library

[7] NUMACTL. Accessed on March 10th, 2017: http://linuxcommand.org/man_pages/numactl8.html

[8] Memkind Heap Manager. Accessed on March 10th, 2017: http://memkind.github.io/memkind/memkind_arch_20150318.pdf

[9] HBWMALLOC. Accessed on March 10th, 2017: https://www.mankier.com/3/hbwmalloc

[10] Charles Yount and Alejandro Duran. 2016. Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling. In Proceedings of the 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS '16). IEEE Press, Piscataway, NJ, USA, 65-75. DOI: https://doi.org/10.1109/PMBS.2016.12

[11] Raghunandan Mathur, Hiroshi Matsuoka, Osamu Watanabe, Akihiro Musa, Ryusuke Egawa, and Hiroaki Kobayashi. 2015. A Case Study of Memory Optimization for Migration of a Plasmonics Simulation Application to SX-ACE. *2015* Third International Symposium on Computing and Networking (CANDAR), Sapporo, 2015, 521-527. DOI:10.1109/CANDAR.2015.105

[12] Ranjith Subramanian, Yannis Smaragdakis, and Gabriel H. Loh. 2006. Adaptive Caches: Effective Shaping of Cache Behavior to Workloads. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39). IEEE Computer Society, Washington, DC, USA, 385-396. DOI: http://dx.doi.org/10.1109/MICRO.2006.7

[13] Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. 2004. Improving Data Locality by Array Contraction. IEEE Transactions on Computers, Vol. 53(9), 1073-1084. DOI: 10.1109/TC.2004.62

[14] Markus Kowarschik, Christian Weiß. 2002. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In Algorithms for Memory Hierarchies, Advanced Lectures, 213 - 232.

[15] A Guide to Vectorization with Intel C++ Compilers. Acessed on March 10th, 2017: https://software.intel.com/sites/default/files/m/4/8/8/2/a/31848-CompilerAutovectorizationGuide.pdf

[16] Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processors. 2016. Accessed on March 10th 2017: https://colfaxresearch.com/knl-avx512/