Intel® Xeon Phi™ Processor 7200 Family Memory Management Optimizations

by

Steve H

Abstract

This paper examines software performance optimization for an implementation of a non-library version of DGEMM executing on the Intel® Xeon Phi™ processor (code-named Knights Landing, with acronym KNL) running the Linux* Operating System (OS). The performance optimizations will incorporate the use of C/C++ High Bandwidth Memory (HBM) application programming interfaces (APIs) for doing dynamic storage allocation from Multi-Channel DRAM (MCDRAM), `_mm_malloc` dynamic storage allocation calls into Double Data Rate (DDR) memory, high-level abstract vector register management, and data prefetching. The dynamic storage allocations will be used to manage tiled data structure objects that will accommodate the memory hierarchy of the Intel Xeon Phi processor architecture. The focus in terms of optimizing application performance execution based on storage allocation is to:

- Align the starting addresses of data objects during storage allocation so that vector operations on the Intel Xeon Phi processor will not require additional special vector alignment when using a vector processing unit associated with each hardware thread.
- Select data tile sizes and do abstract vector register management that will allow for cache reuse and data locality.
- Place select data structures into MCDRAM, through the HBM software package.
- Use data prefetching to improve timely referencing of the tiled data structures into the Intel Xeon Phi processor cache hierarchy.

These methodologies are intended to provide you with insight when applying code modernization to legacy software applications and when developing new software for the Intel Xeon Phi processor architecture.

Contents

1. Introduction 2

What strategies are used to improve application performance? 2

How is this article organized? 3

2. The Intel® Xeon Phi™ Processor Architecture 3

3. Why Does the Intel Xeon Phi Processor Need HBM? 5

How does a software application distinguish between data assigned to DDR versus MCDRAM in flat mode? 7

How does a software application interface to MCDRAM? 8

4. Prefetch Tuning 9

5. Matrix Multiply Background and Programming Example 10

6. Performance Results for a C/C++ Implementation of DGEMM Based on Intel® Math Kernel Library/DGEMM 17

7. Conclusions 26

8. References 26

1. Introduction

The information in this article might help you achieve better execution performance if you are optimizing software applications for the Intel® Xeon Phi™ processor architecture (code-named Knights Landing [1]) that is running the Linux* OS. The scenario is that optimization opportunities are exposed from using profiling analysis software tools such as Intel® VTune™ Amplifier XE [2], and/or Intel® Trace Analyzer and Collector [3], and/or MPI Performance Snapshot [4] where these software tools reveal possible memory management bottlenecks.

What strategies are used to improve application performance?

This article examines memory management, which involves tiling of data structures using the following strategies:

- **Aligned data storage allocation.** This paper examines the use of the _mm_malloc intrinsic for dynamic storage allocation of data objects that reside in Double Data Rate (DDR) memory.
- **Use of Multi-Channel Dynamic Random-Access Memory (MCDRAM).** This article discusses the use of a 16-gigabyte MCDRAM, which is High-Bandwidth Memory (HBM) [1]. MCDRAM on the Intel Xeon Phi processor comprises eight devices (2 gigabytes each). This HBM is integrated on-the Intel® Xeon Phi™ processor package and is connected to the Knights Landing die via a proprietary on-package I/O. All eight MCDRAM devices collectively provide an aggregate Stream triad benchmark bandwidth of more than 450 gigabytes per second [1].
- **Vector register management.** An attempt will be made to manage the vector registers on the Intel Xeon Phi processor by using explicit compiler semantics including C/C++ Extensions for Array Notation (CEAN) [5].
- **Compiler prefetching controls.** Compiler prefetching control will be applied to application data objects to manage data look-ahead into the Intel Xeon Phi processor's L2 and L1 cache hierarchy.

Developers of applications for Intel Xeon Phi processor architecture may find these methodologies useful for optimizing programming applications, which exploit at the core level, hybrid parallel programming consisting of a combination of both threading and vectorization technologies.

How is this article organized?

Section 2 provides insight on the Intel Xeon Phi processor architecture and what software developer may want to think about in doing code modernization for existing applications or for developing new software applications. Part 3 examines storage allocations for HBM (MCDRAM). In this article and for the experiments, data objects that are not allocated in MCDRAM will reside in DDR. Section 4 examines prefetch tuning capabilities. Part 5 provides background material for the matrix multiply algorithm. Section 6 applies the outlined memory management techniques to a double-precision floating-point matrix multiply algorithm (DGEMM), and works through restructuring transformations to improve execution performance. Part 7 describes performance results.

2. The Intel® Xeon Phi™ Processor Architecture

A Knights Landing processor socket has at most 36 active tiles, where a tile is defined as consisting of two cores (Figure 1) [1]. This means that the Knights Landing socket can have at most 72 cores. The two cores within each tile communicate with each other via a 2D mesh on-die interconnect architecture that is based on a Ring architecture (Figure 1) [1]. The communication mesh consists of four parallel networks, each of which delivers different types of packet information (for example, commands, data, and responses) and is highly optimized for the Knights Landing traffic flows and protocols. The mesh can deliver greater than 700 gigabytes per second of total aggregate bandwidth.

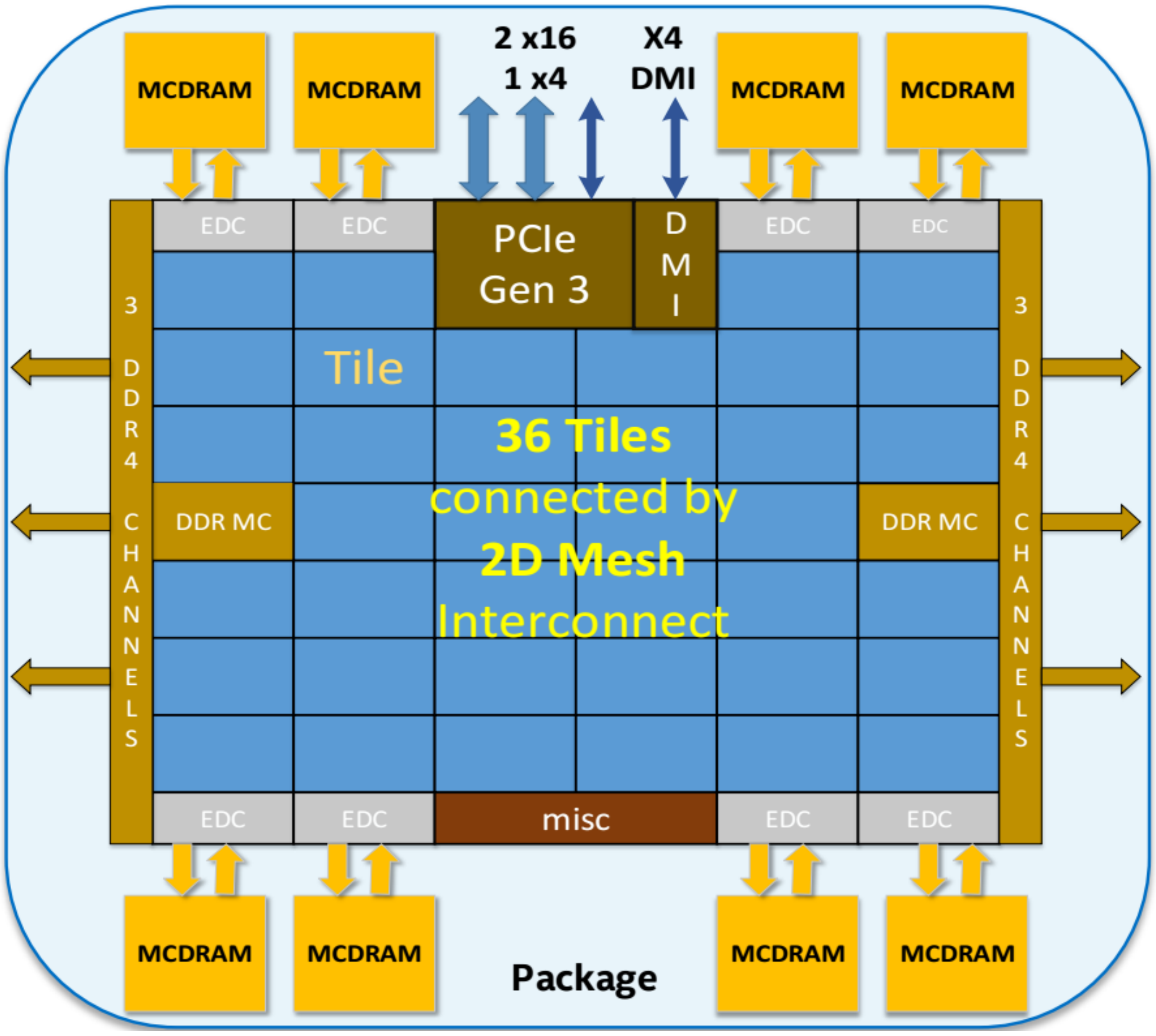

Figure 1. Intel® Xeon Phi™ processor block diagram showing tiles. (DDR MC = DDR memory controller, DMI = Direct Media Interface, EDC = MCDRAM controllers, MCDRAM = Multi-Channel DRAM) [1].

Each core has two Vector Processing Units (VPUs) and 1 megabyte of level-2 (L2) cache that is shared by the two cores within a tile (Figure 2) [1]. Each core within a tile has 32 kilobytes of L1 instruction cache and 32 kilobytes of L1 data cache. The cache lines are 512-bits wide, implying that a cache line can contain 64 bytes of data.

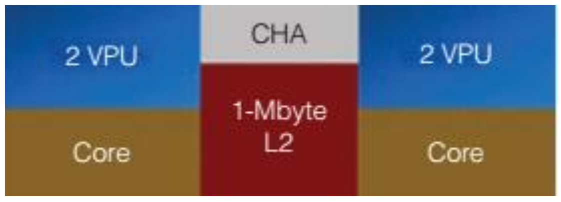

Figure 2. Intel® Xeon Phi™ processor illustration of a tile from Figure 1 that contains two cores (CHA = Caching/Home Agent, VPU = Vector Processing Unit) [1].

In terms of single precision and double-precision floating-point data, the 64-byte cache lines can hold 16 single-precision floating-point objects or 8 double-precision floating-point objects.

Looking at the details of a core in Figure 2, there are four hardware threads (hardware contexts) per core [1], where each hardware thread acts as a logical processor [6]. A hardware thread has 32 512-bit-wide vector registers (Figure 3) to provide Single Instruction Multiple Data (SIMD) support [6]. To manage the 512-bit wide SIMD registers (ZMM0-ZMM31), the Intel® Advanced Vector Extensions 512 (Intel® AVX-512) instruction set is used [7]. For completeness in regard to Figure 3, the lower 256-bits of the ZMM registers are aliased to the respective 256-bit YMM registers, and the lower 128-bits are aliased to the respective 128-bit XMM registers.

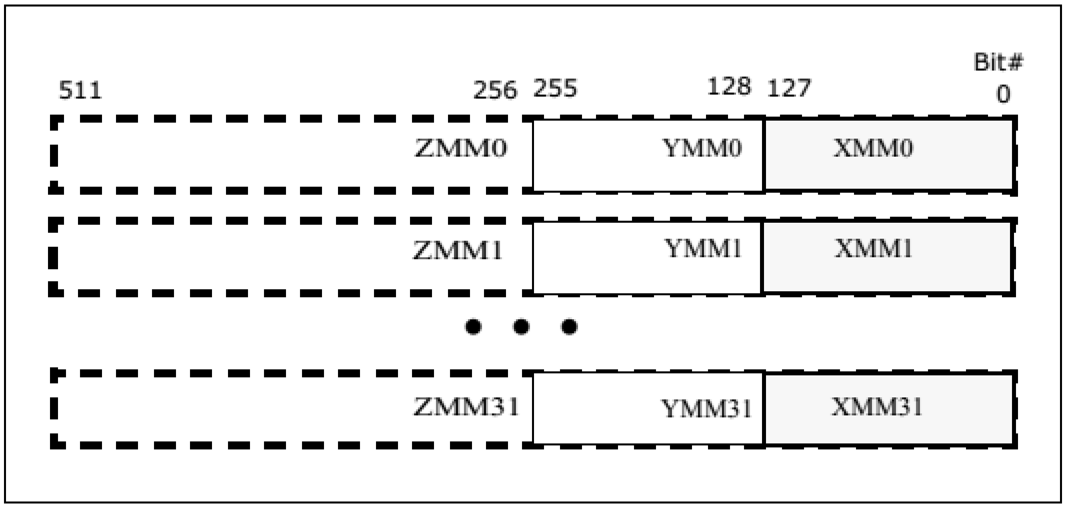

Figure 3. 512-bit-wide vectors and SIMD register set [6].

The rest of this article focuses on the instructions that support the 512-bit wide SIMD registers (ZMM0-ZMM31). Regarding the Intel AVX-512 instruction set extensions, a 512-bit VPU also supports Fused Multiply-Add (FMA) instructions [6], where each of the three registers acts as a source and one of them also functions as a destination to store the result. The FMA instructions in conjunction with the 512-bit wide SIMD registers can do 32 single-precision floating-point computations or 16 double-precision floating-point operations per clock cycle for computational semantics such as:

$$C_{ij} = C_{ij} + A_{ip} \times B_{pj}$$

where subscripts “i”, “j”, and “p” serve as respective row and column indices for matrices **A**, **B**, and **C**.

3. Why Does the Intel Xeon Phi Processor Need HBM?

Conventional Dynamic Random-Access Memory (DRAM) and Dual-Inline Memory Modules (DIMMs) cannot meet the data-bandwidth consumption capabilities of the Intel Xeon Phi processor [8]. To address this “processor to memory bandwidth” issue there are two memory technologies that can be used that place the physical memory closer to the Knights Landing processor socket, namely [8]:

- **MCDRAM:** This is a proprietary HBM that physically sits atop the family of Intel Xeon Phi processors.
- **HBM:** This memory architecture is compatible with the Joint Electron Device Engineering Council (JEDEC) standards [9], and is a high-bandwidth memory designed for a generation of Intel Xeon Phi processors, code named Knights Hill.

From a performance point of view, there is no conceptual difference between MCDRAM and HBM.

For the Intel Xeon Phi processor, MCDRAM as shown in Figure 4 has three memory modes [1]: cache mode, flat mode, and hybrid mode. When doing code modernization for existing applications or performing new application development on Intel Xeon Phi processor architecture, you may want to experiment with the three configurations to find the one that provides the best performance optimization for your applications. Below are some details about the three modes that may help you make informed decisions about which configuration may provide the best execution performance for software applications.

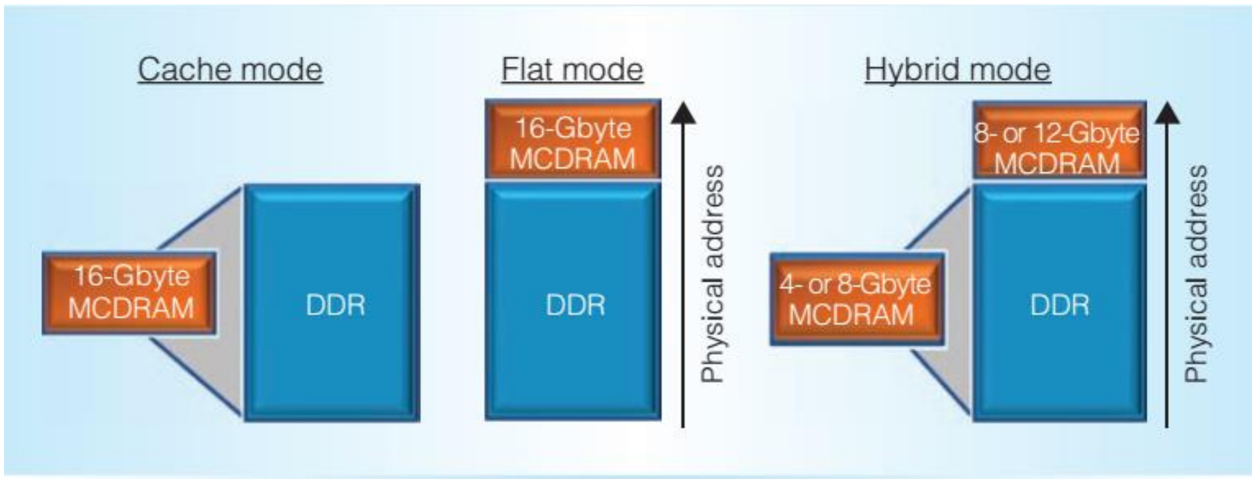

Figure 4. The three MCDRAM memory modes—cache, flat, and hybrid—in the Intel® Xeon Phi™ processor. These modes are selectable through the BIOS at boot time [1].

The **cache** mode does not require any software change and works well for many applications [1]. For those applications that do not show a good hit rate in MCDRAM, the other two memory modes provide more user control to better utilize MCDRAM.

In **flat** mode, both the MCDRAM memory and the DDR memory act as regular memory and are mapped into the same system address space [1]. The flat mode configuration is ideal for applications that can separate their data into a larger, lower-bandwidth region and a smaller, higher bandwidth region. Accesses to MCDRAM in flat mode see guaranteed high bandwidth compared to cache mode, where it depends on the hit rates. Unless the data structures for the application workload can fit entirely within MCDRAM, the flat mode configuration requires software support to enable the application to take advantage of this mode.

For the **hybrid** mode, the MCDRAM is partitioned such that either a half or a quarter of the MCDRAM is used as cache, and the rest is used as flat memory [1]. The cache portion will serve all of the DDR memory. This is ideal for a mixture of software applications that have data structures that benefit from general caching, but also can take advantage by storing critical or frequently accessed data in the flat memory partition. As with the flat mode, software enabling is required to access the flat mode section of the MCDRAM when software does not entirely fit into it. Again as mentioned above, the cache mode section does not require any software support [1].

How does a software application distinguish between data assigned to DDR versus MCDRAM in flat mode?

When MCDRAM is configured in flat mode, the application software is required to explicitly allocate memory into MCDRAM [1]. In a flat mode configuration, the MCDRAM is accessed as memory by relying on mechanisms that are already supported in the existing the Linux* OS software stack. This minimizes any major enabling effort and ensures that the applications written for flat MCDRAM mode remain portable to systems that do not have a flat MCDRAM configuration. This software architecture is based on the Non-Uniform Memory Access (NUMA) memory support model [10] that exists in current operating systems and is widely used to optimize software for current multi-socket systems. The same mechanism is used to expose the two types of memory on Knights Landing as two separate NUMA nodes (DDR and MCDRAM). This provides software with a way to address the two types of memory using NUMA mechanisms. By default, the BIOS sets the Knights Landing cores to have a higher affinity to DDR than MCDRAM. This affinity helps direct all default and noncritical memory allocations to DDR and thus keeps them out of MCDRAM.

On a Knights Landing system one can type the NUMA command:

```
numactl –H
```

or

```
numactl --hardware
```

and you will see attributes about the DDR memory (node 0) and MCDRAM (node 1). The attributes might look something like the following:

```
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 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76
77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101
102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158
159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196
197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215
216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234
235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253
254 255
node 0 size: 32664 MB
node 0 free: 30414 MB
node 1 cpus:
node 1 size: 16384 MB
node 1 free: 15958 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

There is an environment variable called `MEMKIND_HBW_NODES` which controls the binding of high bandwidth memory allocations to one of the two NUMA nodes listed above. For example, if this environment variable is set to 0, it will bind high bandwidth memory allocations to NUMA node 0. Alternatively, setting this environment variable to 1 will bind high bandwidth allocations to NUMA node 1.

How does a software application interface to MCDRAM?

To allocate critical memory in MCDRAM in flat mode, a high-bandwidth (HBW) malloc library is available that can be downloaded at reference [11] or by clicking here. This memkind library [11] has functions that can align data objects on, say, 64-byte boundaries. Such alignments can lead to efficient use of cache lines, the L2 and L1 caches, and the SIMD vector registers. Once the memkind library is installed, the `LD_LIBRARY_PATH` environment variable will need to be updated to include the directory path to the memkind library.

One other topic should be noted regarding huge pages. On Knights Landing, huge pages are managed by the kernel a bit differently when you try to perform memory allocation using them (memory pages of size 2 MB instead of the standard 4 KB) [12]. In those cases, huge pages need to be enabled prior to use. The content of the file called:

```
/proc/sys/vm/nr_hugepages
```

contains the current number of preallocated huge pages of the default size. If for example, you issue the Linux command on the Knights Landing system:

```
cat /proc/sys/vm/nr_hugepages
```

and the file contains a 0, the system administrator can issue the Linux OS command:

```
echo 20 > /proc/sys/vm/nr_hugepages
```

to dynamically allocate and deallocate default sized persistent huge pages, and thus adjust the number of default sized huge pages in the huge page pool to 20. Therefore, the system will allocate or free huge pages, as required. Note that one does not need to explicitly set the number of huge pages by echoing to the file `/proc/sys/vm/nr_hugepages` as long as the content of `/sys/kernel/mm/transparent_hugepage/enabled` is set to "always".

A detailed review for setting the environment variables `MEMKIND_HBW_NODES` and `LD_LIBRARY_PATH` in regard to the memkind library and adjusting the content of the file `nr_hugepages` is discussed in "Performance Results for a C/C++ Implementation of DGEMM Based on Intel® Math Kernel Library/DGEMM".

4. Prefetch Tuning

Compiler prefetching is disabled by default for the Intel Xeon Phi processor [13]. To enable compiler prefetching for Knights Landing architecture use the compiler options:

```
-O3 -xmic-avx512 -qopt-prefetch=<n>
```

where the values of meta-symbol *<n>* are explained in Table 1.

Table 1. Intel® C/C++ compiler switch settings for `-qopt-prefetch`

How does the `-qopt-prefetch=<n>` compiler switch work for the Intel® Xeon Phi™ processor architecture?	
Value of meta-symbol "<*n*>"	**Prefetch Semantic Actions**
0	This is the default and if you omit the `-qopt-prefetch` option, then no auto-prefetching is done by the compiler
2	This is the default if you use only `-qopt-prefetch` with no explicit "<*n*>" argument. Insert prefetches for direct references where the compiler thinks the hardware prefetcher may not be able to handle it
3	Prefetching is turned on for all direct memory references without regard to the hardware prefetcher
4	Same as n=3 (currently)
5	Additional prefetching for all indirect references (Intel® Advanced Vector Extensions 512 (Intel® AVX-512) and above) - Indirect prefetches (hint 1) is done using AVX512-PF gatherpf instructions on Knights Landing (not all cases, but a subset) - Extra prefetches issued for strided vector accesses (hint 0) to cover all cache-lines

The prefetch distance is the number of iterations of look-ahead when a prefetch is issued. Prefetching is done after the vectorization phase, and therefore the distance is in terms of vectorized iterations if an entire serial loop or part of a serial loop is vectorized. The Intel Xeon Phi processor also has a hardware L2 prefetcher that is enabled by default. In general, if the software prefetching algorithm is performing well for an executing application, the hardware prefetcher will not join in with the software prefetcher.

For this article the Intel C/C++ Compiler option:

```
-qopt-prefetch-distance=n1[,n2]
```

is explored. The arguments n_1 and n_2 have the following semantic actions in regard to the –`-qopt-prefetch-distance` compiler switch:

- The distance n_1 (number of future loop iterations) for first-level prefetching into the Intel Xeon Phi processor L2 cache.
- The distance n_2 for second-level prefetching from the L2 cache into the L1 cache, where $n_2 \leq n_1$. The exception is that n_1 can be 0 for values of n_2 (no first-level prefetches will be issued by the compiler).

Some useful values to try for n_1 are 0, 4, 8, 16, 32, and 64 [14]. Similarly, useful values to try for n_2 are 0, 1, 2, 4, and 8. These L2 prefetching values signified by n_1 can be permuted with prefetching values n_2 that control data movement from the L2 cache into the L1 cache. This permutation process can reveal the best combination of n_1 and n_2 values. For example, a setting might be:

```
-qopt-prefetch-distance=0,1
```

where the value 0 tells the compiler to disable compiler prefetching into the L2 cache, and the n_2 value of 1 indicates that 1 iteration of compiler prefetching should be done from the L2 cache into the L1 cache.

The optimization report output from the compiler (enabled using `-opt-report=<m>`) will provide details on the number of prefetch instructions inserted by the compiler for each loop.

In summary, section 2 discussed the Intel Xeon Phi processor many-core architecture, including the on-die interconnection network for the cores, hardware threads, VPUs, the L1 and L2 caches, 512-bit wide vector registers, and 512-bit wide cache lines. Part 3 examined MCDRAM, and the memkind library for helping to establish efficient data alignment of data structures (memory objects). This present section discussed prefetching of these memory objects into the cache hierarchy. In the next section, these techniques will be applied so as to optimize an algorithm [15] such as a double-precision version of matrix multiply. The transformation techniques will incorporate using a high-level programming language in an attempt to maintain portability from one processor generation to the next [16].

5. Matrix Multiply Background and Programming Example

Matrix multiply has the core computational assignment:

$$C_{ij} = C_{ij} + A_{ip} \times B_{pj}$$

A basic matrix multiply loop structure implemented in a high-level programming language might look something like the following pseudo-code:

```
integer i, j, p;

for p = 1:K
    for j = 1:N
        for i = 1:M
           $C_{ij} = C_{ij} + A_{ip} \times B_{pj}$
        endfor
    endfor
```

```
        endfor
```

where matrix ***A*** has dimensions M × K, matrix ***B*** has dimensions K × N, and matrix ***C*** has dimensions M × N. For the memory offset computation for matrices ***A***, ***B***, and ***C*** we will assume column-major-order data organization.

For various processor architectures, software vendor libraries are available for performing matrix multiply in a highly efficient manner. For example, matrix multiplication for the above can be computed using DGEMM which calculates the product for a matrix **C** using double precision matrix elements [17]. Note that a DGEMM core solver for implementing the above algorithm *may* be implemented in assembly language (e.g., DGEMM for the Intel® Math Kernel Library [17]), where an assembly language solution may not be *necessarily* portable from one processor architecture to the next.

In *regard* to this article, the focus is to do code restructuring transformations to achieve code modernization performance improvements using a high-level programming language. The reason for using matrix multiply as an example in applying the high-level memory-allocation-optimization techniques is that the basic algorithm is roughly four lines long and is easily understandable. Additionally, it is hoped that after you see a before and after of the applied restructuring transformations using a high-level programming language, you will think about associating restructuring transformations of a similar nature to the applications that you have written in a high-level programming language which are targeted for code modernization techniques.

Goto et al. [15] have looked at restructuring transformations for the basic matrix multiply loop structure shown above in order to optimize it for various processor architectures. This has required organizing the ***A***, ***B***, and ***C*** matrices from the pseudo-code example above into sub-matrix tiles. Figure 5 shows a tile rendering abstraction, but note that the access patterns required in Figure 5 are different from those described in reference [15]. For the $\widetilde{\boldsymbol{A}}$ and $\widetilde{\boldsymbol{B}}$ tiles in Figure 5, data packing is done to promote efficient matrix-element memory referencing.

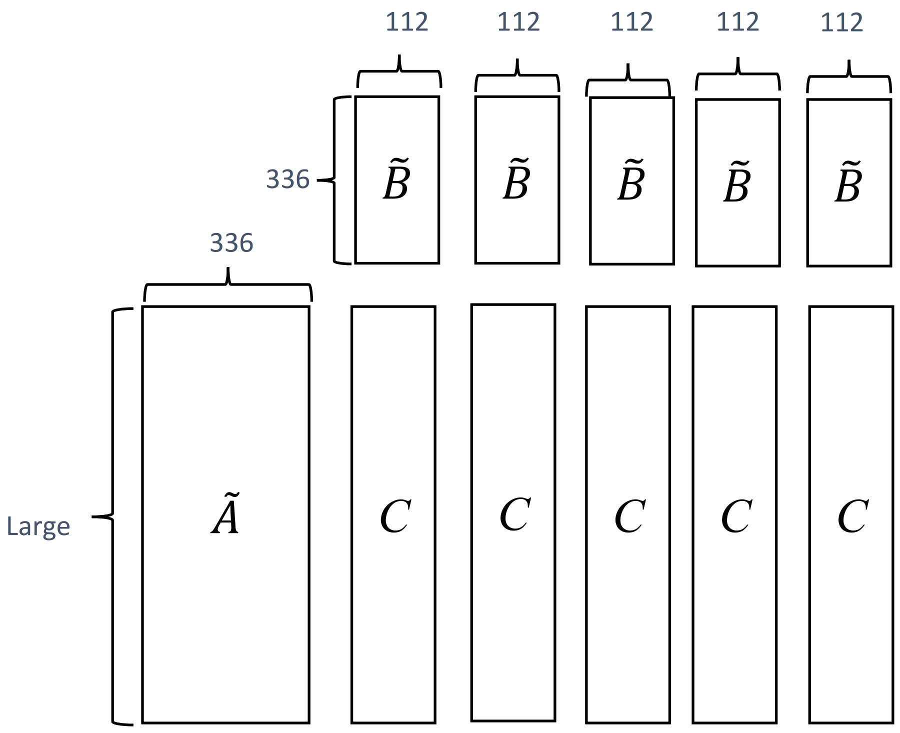

Figure 5. Partitioning of DGEMM for the Intel® Xeon Phi™ processor where buffer $\tilde{\boldsymbol{A}}$ is shared by all cores, and buffer $\tilde{\boldsymbol{B}}$ and sections of matrix ***C*** are not shared by all cores. The data partitioning is based on an Intel® Xeon Phi™ processor/DGEMM implementation from the Intel® Math Kernel Library [17].

Regarding the matrix partitioning in Figure 5 for the Intel Xeon Phi processor, $\tilde{\boldsymbol{A}}$ is shared by all the cores, and matrices $\tilde{\boldsymbol{B}}$ and ***C*** are not shared by all the cores. This is for a multi-threaded DGEMM solution. Parallelism for the Intel Xeon Phi processor can be demonstrated as threading at the core level (Figure 1), and then as shown in Figure 2, the VPUs can exploit vectorization with 512-bit vector registers and SIMD semantics.

For the sub-matrices in Figure 5 that are either shared by all of the Intel Xeon Phi processor cores (for example, sub-matrix $\tilde{\boldsymbol{A}}$), or for the sub-matrices that are not shared (for example, sub-matrices for $\tilde{\boldsymbol{B}}$ and partitions for matrix ***C***), the next question is: what memory configurations should you use (for example, DDR or MCDRAM)?

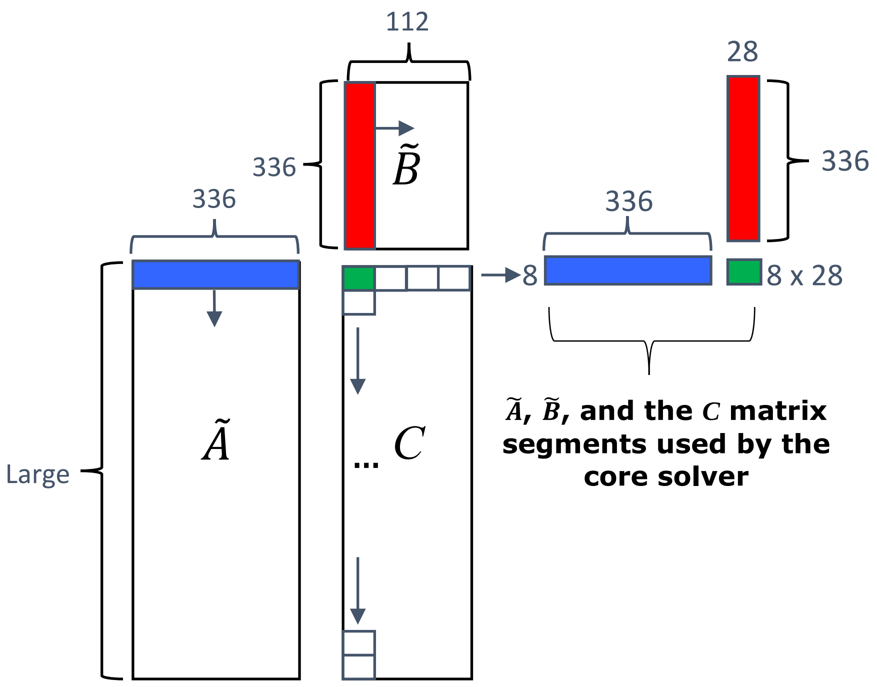
112
28
336
336
336
$\tilde{B}$
336
8
8 x 28
$\tilde{A}$
... C
Large
$\tilde{A}$, $\tilde{B}$, and the C matrix segments used by the core solver

Figure 6. DGEMM kernel solver for Intel® Xeon Phi™ processor with partitions for $\tilde{\boldsymbol{A}}$, $\tilde{\boldsymbol{B}}$, and ***C*** [17].

Recall that there is 16 gigabytes of multi-channel DRAM and therefore since sub-matrix $\tilde{\boldsymbol{A}}$ is shared by all the cores, it will be placed into MCDRAM using the flat mode configuration. In section 3, we examined HBM, where for MCDRAM there were three configurations: cache mode, flat mode, and hybrid mode (Figure 4). It was mentioned that the flat mode configuration is ideal for applications that can separate their data into a larger, lower-bandwidth region and a smaller, higher bandwidth region. Following this rule for flat mode, we will place $\tilde{\boldsymbol{A}}$ (Figure 6) into MCDRAM using the following "memkind" library prototype:

```
int hbw_posix_memalign(void **memptr, size_t alignment, size_t
     size);
```

where the alignment argument "`size_t alignment`" might have a value of 64, which is a power of 2 and allows the starting address of $\tilde{\boldsymbol{A}}$ to align on the beginning of a cache line.

In Figure 6, note that matrix ***C*** consists of a core partition that has 8 rows and 28 columns. From an abstraction point of view, the 8 double-precision matrix elements (64 bytes total) can fit into a 512-bit (64 byte) vector register. Also, recall that there are 32 512-bit vector registers per hardware thread. To reduce register pressure on a hardware thread, 28 of the vector registers will be used for the core solver on the right side of Figure 6.

Similarly, for the other tiling objects in Figure 6, the `_mm_malloc` intrinsic will be used to allocate storage in DDR memory on Knights Landing. The `_mm_malloc` function prototype looks as follows:

```
     void *_mm_malloc (size_t size, size_t align);
```

The `_mm_malloc` prototype also has a "`size_t align`" argument, which again is an alignment constraint. Using a value of 64 allows data objects that are dynamically allocated to have their starting address aligned on the beginning of a cache line.

For Figure 6, matrix $\widetilde{\boldsymbol{B}}$ will be migrated into the L2 cache.

To summarize, we have discussed the partitioning of the ***A***, ***B***, and ***C*** matrices into sub-matrix data tiles, and we have utilized 28 of the 32 512-bit vector registers. We looked at data storage prototypes for placing data into MCDRAM or DDR. Next, we want to explore how the data elements within the sub-matrices will be organized (packed) to provide efficient access and reuse.

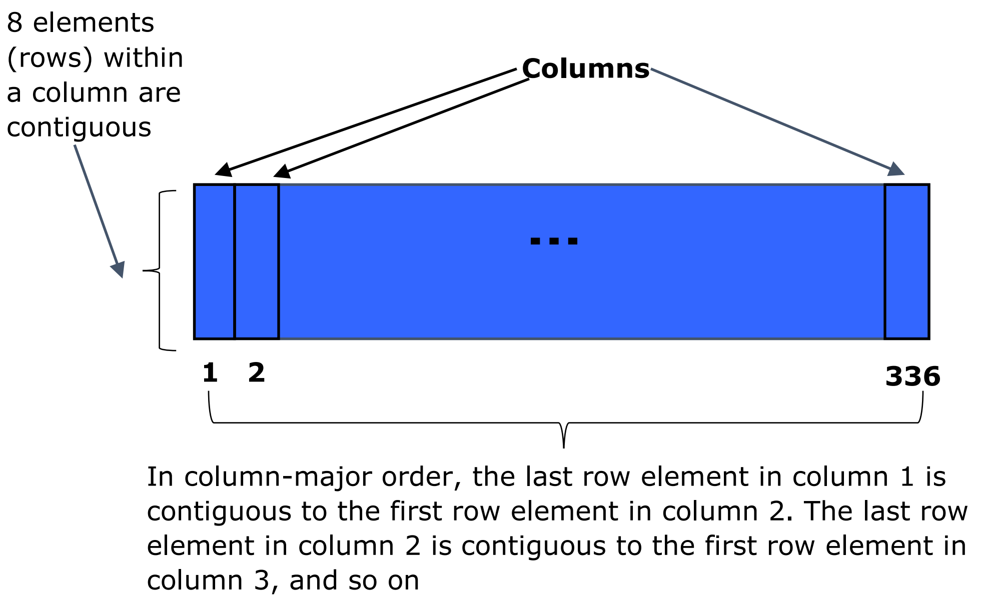
8 elements (rows) within a column are contiguous
Columns
...
1
2
336
In column-major order, the last row element in column 1 is contiguous to the first row element in column 2. The last row element in column 2 is contiguous to the first row element in column 3, and so on

Figure 7. Data element packing for 8 rows by 336 columns of matrix segment $\widetilde{\boldsymbol{A}}$ using column-major-order memory offsets [17].

Recall from Figure 5 and Figure 6 that matrix segment $\widetilde{\boldsymbol{A}}$ has a large number of rows and 336 columns. The data is packed into strips that have 8 row elements for each of the 336 columns (Figure 7) using column-major order memory offsets. The number of strips for matrix segment $\widetilde{\boldsymbol{A}}$ is equal to:

$$\text{Large-number-of-rows} / 8$$

Note that 8 double-precision row elements for each of the 336 columns can provide efficient use of the 512-bit wide cache lines for the L2 and L1 caches.

For matrix segment $\widetilde{\boldsymbol{B}}$ in Figure 5 and Figure 6, the 336 row by 112 column tiles are sub-partitioned into 336 rows by 28 column strips (Figure 8). In Figure 8, the matrix segment $\widetilde{\boldsymbol{B}}$

has strips that use row-major-order memory offsets and therefore the 28 elements in a row are contiguous. The 28 elements of a row for matrix segment $\widetilde{\boldsymbol{B}}$ correspond with the 28 elements for matrix ***C*** that are used in the core solver computation illustrated in the right portion of Figure 6.

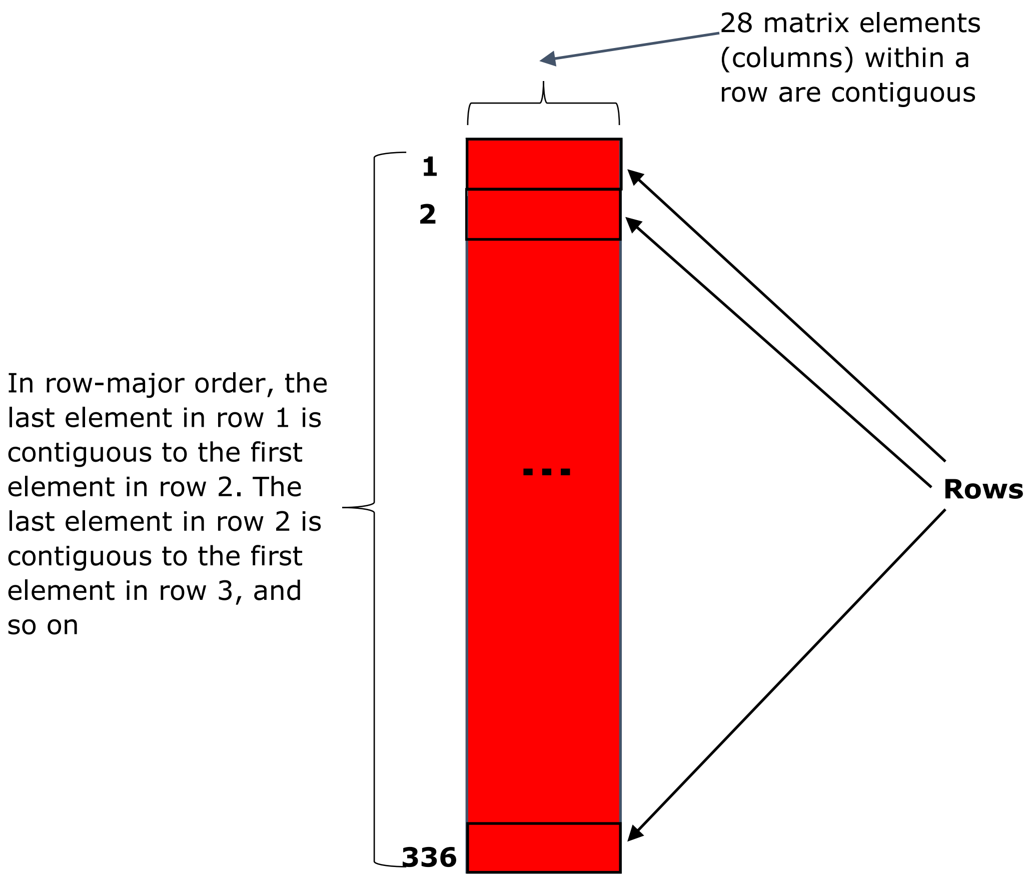
28 matrix elements (columns) within a row are contiguous
1
2
In row-major order, the last element in row 1 is contiguous to the first element in row 2. The last element in row 2 is contiguous to the first element in row 3, and so on
...
Rows
336

Figure 8. Data element packing for 336 rows by 28 columns of matrix segment $\widetilde{\boldsymbol{B}}$ using row-major-order memory offsets [17].

Figure 9 shows a column-major-order partition of matrix ***C*** that has 8 array elements within a column and there are 28 columns (in green). As mentioned earlier, the 8 contiguous double-precision array elements within a column will fit into a 512-bit (64 byte) vector register, and the 28 columns contain 8 row-elements each that can map onto 28 of the 32 possible vector registers associated with a hardware thread. In Figure 9, note that when the next 8 row by 28 column segment for matrix ***C*** (in white) is processed, the first element in each column is adjacent to the last element in each column with respect to the green partition. Thus, this column major ordering can allow the 8 row by 28 column segment (in white) to be efficiently prefetched for anticipated FMA computation.

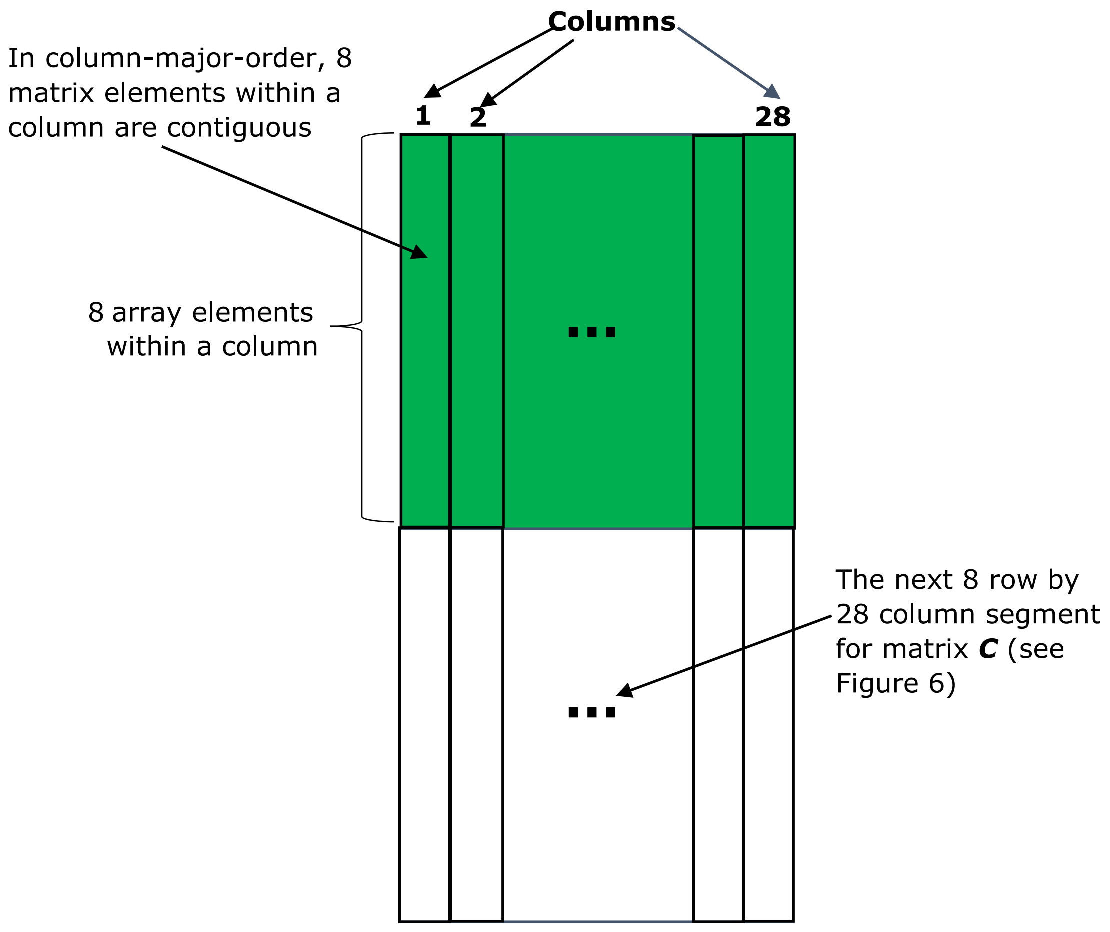
Columns
1
2
28
In column-major-order, 8 matrix elements within a column are contiguous
8 array elements within a column
...
The next 8 row by 28 column segment for matrix ***C*** (see Figure 6)
...

Figure 9. Data element organization for 8 rows by 28 columns of matrix segment ***C*** using column-major-order memory offsets [17].

In regard to Figures 7, 8, and 9, we have completed the data referencing analysis for the $\tilde{\boldsymbol{A}}$, $\tilde{\boldsymbol{B}}$, and ***C*** matrices, which are color coded to make it easy to associate the rectangular data tiling abstractions with the three matrix multiply data objects. Putting this all together into a core matrix multiply implementation, a possible pseudo-code representation for the core solver in Figure 6 that also reflects the data referencing patterns for Figures 7, 8 and 9, might look something like the following:

```
for ( jjc = … )
    for ( iir = … )
        for ( ppc = … )
            C[ir+iir:8,jc+jjc] += Ã[iir:8,ppc] × B̃[l,ppc,0];
            C[ir+iir:8,jc+jjc+1] += Ã[iir:8,ppc] × B̃[l,ppc,1];
     …
               $\boldsymbol{C}$[ir+iir:8,jc+jjc+26] += $\tilde{\boldsymbol{A}}$[iir:8,ppc] × $\tilde{\boldsymbol{B}}$[l,ppc,26];
               $\boldsymbol{C}$[ir+iir:8,jc+jjc+27] += $\tilde{\boldsymbol{A}}$[iir:8,ppc] × $\tilde{\boldsymbol{B}}$[l,ppc,27];
        endfor
    endfor
endfor
```

C/C++ Extensions for Array Notation (CEAN) [5] are used in the pseudo-code as indicated by the colon notation "…:8" within the subscripts. This language extension is used to describe computing 8 double-precision partial results for matrix $\boldsymbol{C}$ by using 8 double-precision elements of $\tilde{\boldsymbol{A}}$ and replicating a single element of $\tilde{\boldsymbol{B}}$ eight times and placing the same 8 values into a 512-bit vector register for the $\tilde{\boldsymbol{B}}$ operand to make it possible to take advantage of FMA computation. For matrix $\tilde{\boldsymbol{B}}$ the subscript "`l`" (the character `l` is the letter L) is used to reference the level (entire strip in Figure 8), where there are 4 levels in the $\tilde{\boldsymbol{B}}$ matrix, each containing 336 rows and 28 columns. This accounts for the value 112 (28 × 4) in Figure 6.

6. Performance Results for a C/C++ Implementation of DGEMM Based on Intel® Math Kernel Library/DGEMM

This section describes three DGEMM experiments that were run on a single Intel Xeon Phi processor socket that had 64 cores and 16 gigabytes of MCDRAM. The first experiment establishes a baseline for floating-point-operations-per second performance. Experiments 2 and 3 attempt to demonstrate increasing floating-point-operations-per-second performance. All three executables do storage allocation into MCDRAM or DDR using the respective function prototypes:

```
int hbw_posix_memalign(void **memptr, size_t alignment, size_t
size);
```

and

```
    void *_mm_malloc (size_t size, size_t align);
```

MCDRAM for these experiments was configured in flat mode. The executables were built with the Intel C/C++ Compiler. Cache management was used for the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices by transferring data into tile data structures that would fit into the L2 and L1 caches. MCDRAM was used for the $\tilde{A}$-tile. The other data structures that were allocated for this Intel® Math Kernel Library (Intel® MKL)/DGEMM implementation used DDR memory.

Please note that on your system, the floating-point-operations-per-second results will vary from those shown in Figures 10, 11, and 12. Results will be influenced, for example, by factors *such as* the version of the OS, the software stack component versions, the processor stepping, the number of cores on a socket, and the storage capacity of MCDRAM.

A shell script for running the experiment that resulted in the data for Figure 9 had the following arguments:

64 1 336 112 43008 43008 dynamic 2 <*path-to-memkind-library*>

- 64 defines the number of core threads.
- 1 defines the number of hardware threads per core that are to be used.
- 336 defines the number of columns for the $\tilde{A}$ and the number of rows for the $\tilde{B}$ tiling data structures. See Figures 5 and 6.
- 112 defines the number of columns for the $\tilde{B}$ data structure tile. Also, see Figures 5 and 6.
- 43008 is the matrix order.
- The second value 43008 refers to the number of rows for $\tilde{A}$.
- The values dynamic and 2 are used to control the OpenMP* scheduling [18] [19] (see Table 2 below).
- The meta-symbol <*path-to-memkind-library*> refers the directory path to the memkind library that is installed on the user's Knights Landing system.

The first experiment is based on the data tiling storage diagrams from the section 5. Recall that each 512-bit vector register for Intel Xeon Phi processor can reference eight double-precision floating-point operations, and there is also an opportunity to use the FMA vector instruction for the core computation:

```
for ( … )
     C[ir+iir:8,jc+jjc] += …
     C[ir+iir:8,jc+jjc+1] += …

     …

     C[ir+iir:8,jc+jjc+26] += …
     C[ir+iir:8,jc+jjc+27] += …
endfor
```

For the compilation of orig_knl_dgemm.c into an executable for running on the Intel Xeon Phi processor, the floating-point-operations-per-second results might look something like the following:

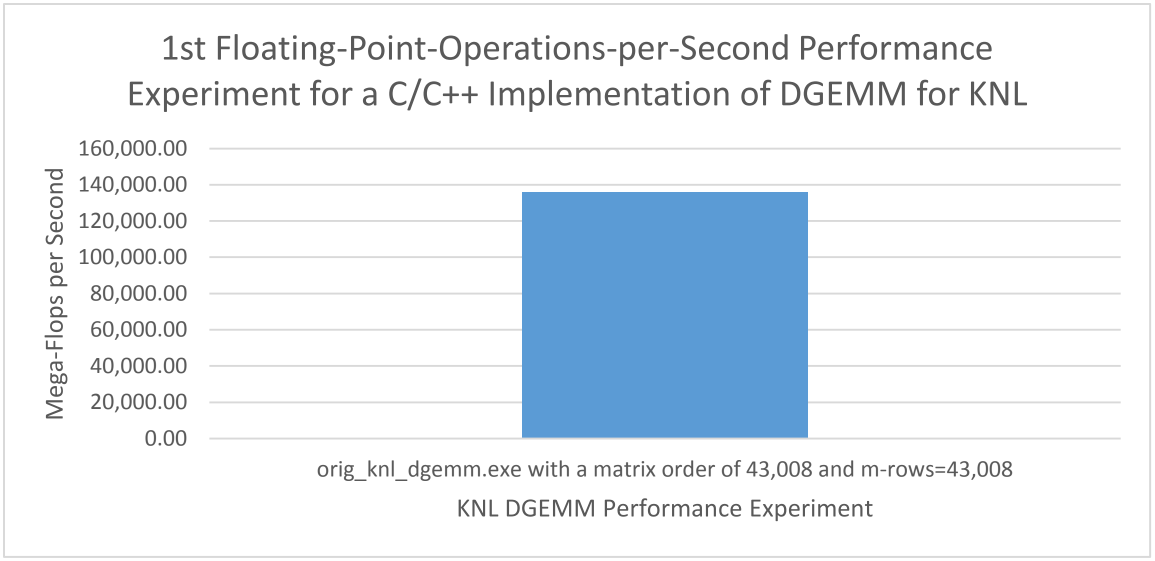

Figure 10. Intel® Xeon Phi™ processor result for the executable orig_knl_dgemm.exe using 64 core threads and 1 OpenMP* thread per core. The matrix order was 43,008. The $\tilde{A}$ tile data structure for the ***A***-matrix had 336 columns and the M-rows value was 43,008. 0 abstract vector registers were used for the matrix-multiply core solver.

In the next experiment, the source file called opt_knl_dgemm.c is used to build the executable called reg_opt_knl_dgemm.exe. In this file, references to the 28 columns of matrix **C** in the core solver are replaced with the following:

```
t0[0:8]  = C[ir+iir:8,jc+jjc];
t1[0:8]  = C[ir+iir:8,jc+jjc+1];

…

t26[0:8]  = C[ir+iir:8,jc+jjc+26];
t27[0:8]  = C[ir+iir:8,jc+jjc+27];

for ( … )
    t0[0:8]  += …
    t1[0:8]  += …

…

    t27[0:8]  += …
endfor

C[ir+iir:8,jc+jjc]  += …
C[ir+iir:8,jc+jjc+1]  += …

     …
```

```
C[ir+iir:8,jc+jjc+26] = t26[0:8];
C[ir+iir:8,jc+jjc+27] = t27[0:8];
```

The notion of using the array temporaries `t0` through `t27` can be thought of as assigning abstract vector registers in the computation of partial results for the core matrix multiply algorithm. For this experiment on the Intel Xeon Phi processor, the floating-point-operations-per-second results might look something like:

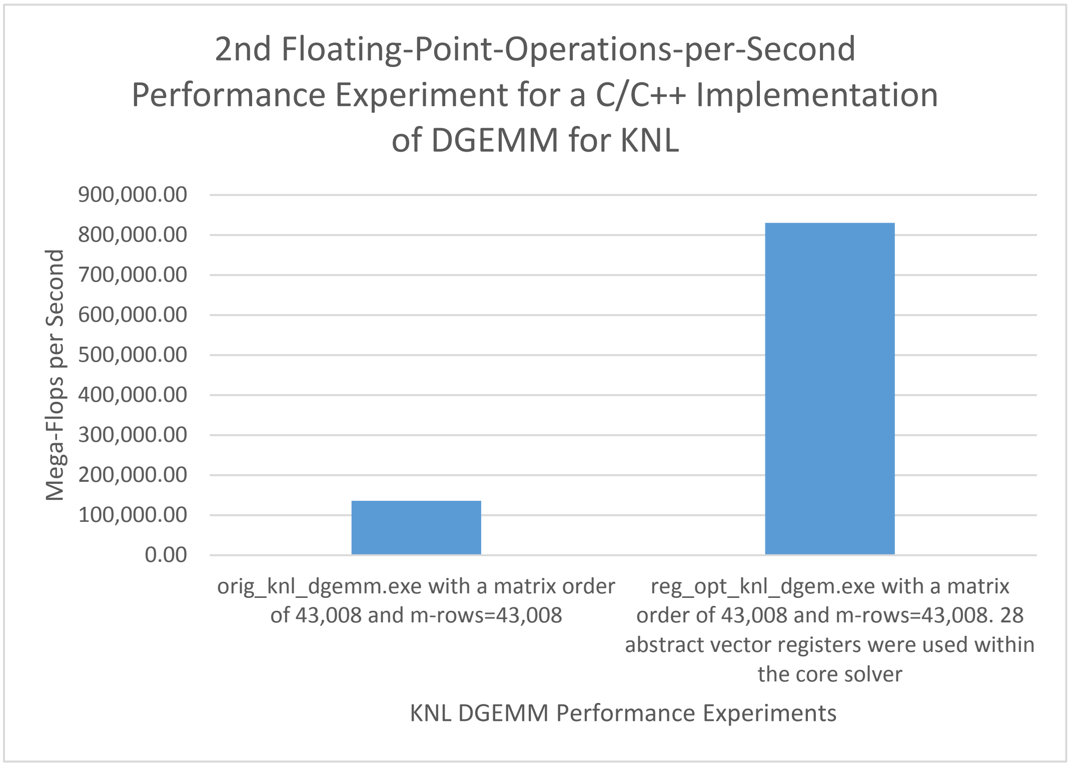

Figure 11. Intel® Xeon Phi™ processor performance comparison between the executable, orig_knl_dgemm.exe and the executable, reg_opt_knl_dgemm.exe using 64 core threads and 1 OpenMP* thread per core. The matrix order was 43,008. The $\tilde{A}$ tile data structure for the ***A***-matrix had 336 columns and the M-rows value was 43,008. The executable reg_opt_knl_dgemm.exe used 28 abstract vector registers for the matrix-multiply core solver

Note that in Figure 11, the result for the executable, orig_knl_dgemm.exe is compared with the result for the executable, reg_opt_knl_dgemm.exe (where 28 abstract vector registers were used). As mentioned previously, from an abstract vector register perspective, the intent was to explicitly manage 28 of the thirty-two 512 bit vector registers for a hardware thread within a Knights Landing core.

The last experiment (experiment 3) builds the Intel MKL/DGEMM executable called pref_32_0_reg_opt_knl_dgemm.exe using the Intel C/C++ compiler options `-qopt-`

`prefetch=2` and `-qopt-prefetch-distance=`*n_1*`,`*n_2* where *n_1* and *n_2* are replaced with integer constants. The `-qopt-prefetch-distance` switch is used to control the number of iterations of data prefetching that take place for the L2 and L1 caches on Knights Landing. The L2, L1 combination that is reported here is (32,0). Figure 12 shows a comparison of experiments 1, 2, and 3.

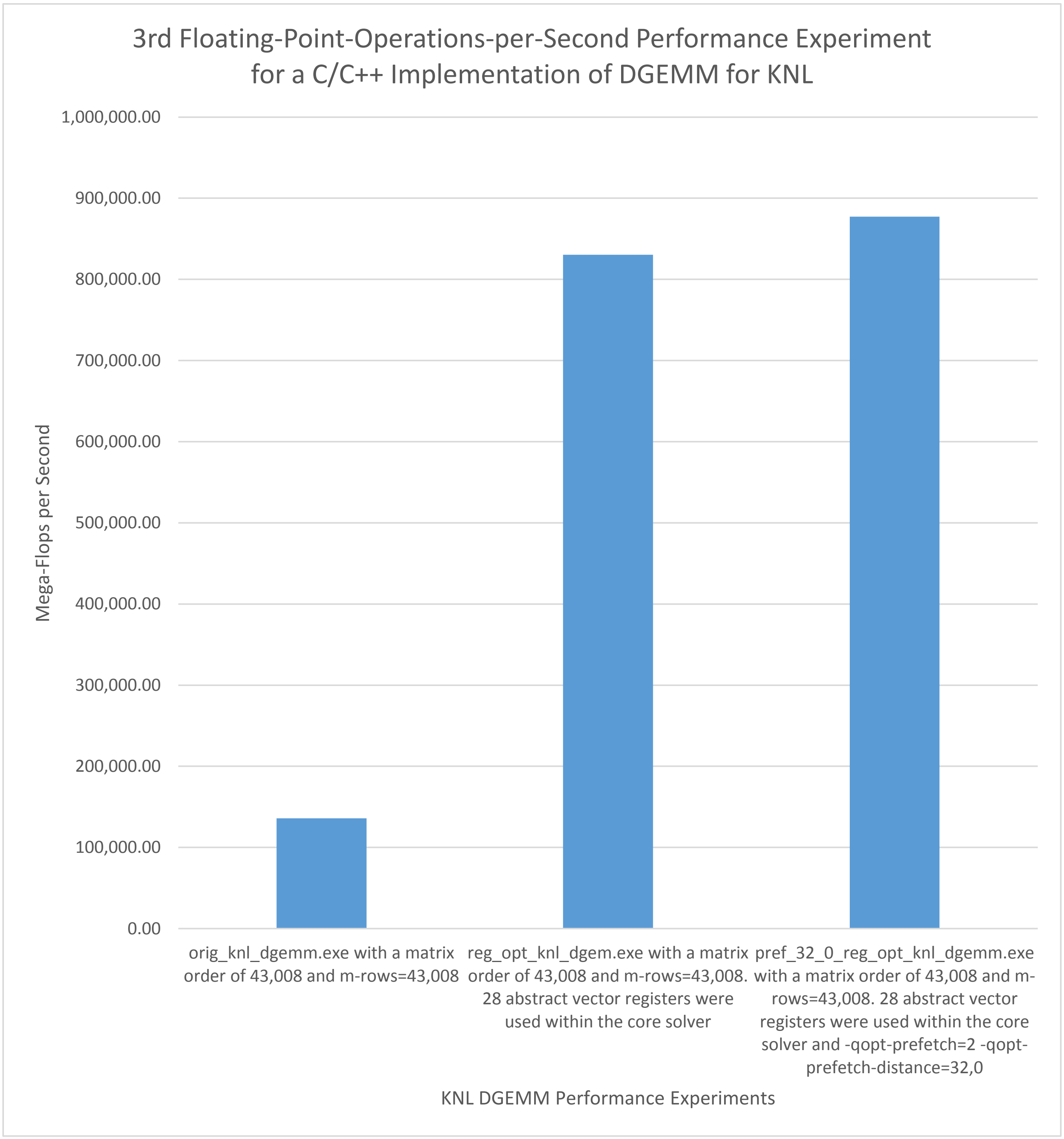

Figure 12. Intel® Xeon Phi™ processor performance comparisons for executables, orig_knl_dgemm.exe, reg_opt_knl_dgem.exe, and pref_32_0_reg_opt_knl_dgemm.exe. Each executable used 64 core threads and 1 OpenMP* thread per core. The matrix order was 43,008. The $\widetilde{\boldsymbol{A}}$ tile data structure for the $\boldsymbol{A}$-matrix had 336 columns and the M-rows

value was 43,008. The executables reg_opt_knl_dgemm.exe and pref_32_0_reg_opt_knl_dgemm.exe used 28 abstract vector registers for the matrix-multiply core solver. The executable pref_32_0_reg_opt_knl_dgemm.exe was also built with the Intel® C/C++ Compiler prefetch switches `-qopt-prefetch=2` and `-qopt-prefetch-distance=32,0`

For the three experiments discussed above, the user can download the shell scripts, makefiles, C/C++ source files, and a README.TXT file at the following URL:

Knights Landing/DGEMM Download Package

After downloading and untarring the package, note the following checklist:

1) Make sure that the HBM software package called memkind is installed on your Knights Landing system. Click here to retrieve the package, if it is not already installed.

2) Set the following environment variables:

```
export MEMKIND_HBW_NODES=1
export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:<path-to-memkind-library>/lib
```

where *<path-to-memkind-library>* is a meta-symbol and represents the directory path to the memkind library where the user has done their installation of this library.

3) Issue the command:

```
cat /proc/sys/vm/nr_hugepages
```

If it does not have a value of 20, then ask your system administrator to change this value on your Knights Landing system using root privileges. With a system administrator account, this can be done by issuing the command:

```
echo 20 > /proc/sys/vm/nr_hugepages
```

followed by the verification command:

```
cat /proc/sys/vm/nr_hugepages
```

As mentioned earlier in a subsection of Section 3, one does not need to explicitly set the number of huge pages by echoing to `/proc/sys/vm/nr_hugepages` as long as the content of `/sys/kernel/mm/transparent_hugepage/enabled` is set to "always".

4) Review the content of the README.TXT file that is within the directory `opt_knl_dgemm_v1` on your Knights Landing system. This read-me file contains information about how to build and run the executables from a host Knights Landing system. The README.TXT file should be used as a guide for doing your own experiments.

Once you complete the checklist on the Intel Xeon Phi processor system, you can source an Intel Parallel Studio XE Cluster Edition script called `psxevars.sh` by doing the following:

```
. <path-to-Intel-Parallel-Studio-XE-Cluster-Edition>/psxevars.sh
intel64
```

This script is sourced in particular to set up the Intel C/C++ compilation environment.

For experiment 1, issue a command sequence that looks something like the following within the directory `opt_knl_dgemm_v1`:

```
$ cd ./scripts
$ ./orig_knl_dgemm.sh <path-to-memkind-library>
```

The output report for a run with respect to the `scripts` sub-directory will be placed in the sibling directory called `reports`, and the report file should have a name something like:

```
orig_knl_dgemm_report.64.1.336.112.43008.43008.dynamic.2
```

where the suffix notation for the report file name has the following meaning:

- 64 defines the number of core threads.
- 1 defines the number of hardware threads per core that are to be used.
- 336 defines the number of columns for the $\tilde{\boldsymbol{A}}$ and the number of rows for the $\tilde{\boldsymbol{B}}$ tiling data structures. See Figures 5 and 6.
- 112 defines the number of columns for the $\tilde{\boldsymbol{B}}$ data structure tile. Also, see Figures 5 and 6.
- 43008 is the matrix order.
- The second value 43008 refers to the number of rows for $\tilde{A}$.
- The values dynamic and 2 are used to control the OpenMP scheduling (see below).

As mentioned earlier, OpenMP is used to manage threaded parallelism. In so doing, the OpenMP Standard [18] provides a scheduling option for work-sharing loops:

`schedule(`*`kind[, chunk_size]`*`)`

This scheduling option is part of the C/C++ directive: `#pragma omp parallel for`, or `#pragma omp for`, and the Fortran* directive: `!$omp parallel do`, or `!$omp do`. The `schedule` clause specifies how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of a team. Each thread executes its assigned chunk or chunks in the context of its implicit task. The *`chunk_size`* expression is evaluated using the original list items of any variables that are made private in the loop construct.

Table 2 provides a summary of the possible settings for the "kind" component for the "schedule" option.

Table 2. "kind" scheduling values for the OpenMP* schedule(kind[, chunk_size])directive component for OpenMP work sharing loops [18,19].

Kind	Description
`Static`	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop

	iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is the loop-count/number-of-threads. Set chunk to 1 to interleave the iterations.
`Dynamic`	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
`Guided`	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies the minimum size chunk to use. By default, the chunk size is approximately loop-count/number-of-threads.
`Auto`	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
`Runtime`	Uses the OMP_SCHEDULE environment variable to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

An alternative to using the scheduling option of the C/C++ directives:

`#pragma omp parallel for` or `#pragma omp for`

or the Fortran directives:

`!$omp parallel do` or `!$omp do`

is to use the OpenMP environment variable `OMP_SCHEDULE`, which has the options:

`type``[,`*`chunk`*`]`

where:

- *`type`* is one of `static`, `dynamic`, `guided`, or `auto`.
- *`chunk`* is an optional positive integer that specifies the chunk size.

Finally, experiments 2 and 3 can be launched from the scripts directory by using the commands:

```
./reg_opt_knl_dgemm.sh <path-to-memkind-library>
```

and

```
./pref_reg_opt_knl_dgemm.sh <path-to-memkind-library>
```

In a similar manner, the output report for each run with respect to the `scripts` sub-directory will be placed in the sibling directory called `reports`.

7. Conclusions

The experiments on an Intel Xeon Phi processor architecture using HBM library storage allocation along with MCDRAMM for a non-library C/C++ implementation of KNL/DGEMM indicate that data alignment, data placement, and management of the vector registers can help provide good performance on the Intel Xeon Phi processor. Management of the Intel Xeon Phi processor vector registers at the program-language-application-level was done with abstract vector registers. In general, you may want to use conditional compilation macros within your applications to control the selection of the high-bandwidth libraries for managing dynamic storage allocations into MCDRAM versus DDR. In this way, you can experiment with the application to see which storage allocation methodology provides the best execution performance for your application running on Intel Xeon Phi processor architectures. Finally, compiler prefetching controls were used for the L1 and L2 data caches. The experiments showed that making adjustments to prefetching further improved execution performance.

As mentioned earlier, the core solver for MKL DGEMM is written in assembly language, and when a user finds a need to use DGEMM as part of a software application programming solution, Intel® MKL DGEMM should be used. For completeness, the following URL provides performance charts for Intel® MKL DGEMM on Knights Landing:

https://software.intel.com/en-us/intel-mkl/benchmarks#DGEMM3

8. References

1. A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, Y. Liu, "KNIGHTS LANDING: SECOND GENERATION INTEL® XEON PHI PRODUCT," *IEEE MICRO*, March/April 2016, pp. 34-46.

2. "Intel® VTune™ Amplifier 2017," https://software.intel.com/en-us/intel-vtune-amplifier-xe.

3. "Intel® Trace Analyzer and Collector," https://software.intel.com/en-us/intel-trace-analyzer.

4. "Getting Started with the MPI Performance Snapshot," https://software.intel.com/en-us/articles/getting-started-with-the-mpi-performance-snapshot.

5. "C/C++ Extensions for Array Notations Programming Model," https://software.intel.com/en-us/node/522649

6. "Intel® 64 and IA-32 Architectures Software Developer Manuals", https://software.intel.com/en-us/articles/intel-sdm#combined.

7. "Intel® Architecture Instruction Set Extensions Programming Reference", https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference.

8. B. Brett, “Multi-Channel DRAM (MCDRAM) and High-Bandwidth Memory (HBM),” https://software.intel.com/en-us/articles/multi-channel-dram-mcdram-and-high-bandwidth-memory-hbm.

9. https://www.jedec.org/

10. http://man7.org/linux/man-pages/man7/numa.7.html

11. https://github.com/memkind/memkind

12. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt

13. “Intel® C++ Compiler 17.0 Developer Guide and Reference,” https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide.

14. R. Krishnaiyer, “Compiler Prefetching for the Intel® Xeon Phi™ coprocessor,” https://software.intel.com/sites/default/files/managed/54/77/5.3-prefetching-on-mic-update.pdf.

15. K. Goto and R. van de Geijn, “Anatomy of High-Performance Matrix Multiplication,” *ACM Transactions on Mathematical Software*, Vol. 34, No. 3, May 2008, pp. 1-25.

16. Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processors, https://colfaxresearch.com/knl-avx512, May 2016.

17. “Intel® Math Kernel Library (Intel® MKL)”, https://software.intel.com/en-us/intel-mkl/?cid=sem43700010399172817&intel_term=intel+mkl&gclid=CKK8mIfJ49ACFYlgfgodbzcD6A&gclsrc=aw.ds

18. “The OpenMP API Specification for Parallel Programming,” http://openmp.org/wp/openmp-specifications.

19. R. Green, “OpenMP Loop Scheduling,” https://software.intel.com/en-us/articles/openmp-loop-scheduling.

Notices