

TR-2014-03

Characterization of Intel Xeon Phi for Linear Algebra  
Workloads

Omkar Deshmukh, Dan Negrut

May 21, 2014

## Abstract

This study focuses on applicability of Intel Xeon Phi coprocessor for some of the Basic Linear Algebra Subprograms (BLAS) subroutines. Based on Many Integrated Core (MIC) architecture, the vector processing unit (VPU) in Xeon Phi coprocessor provides data parallelism at a very fine grain, working on 512 bits of 16 single-precision floats or 32-bit integers at a time. In our work we analyze how well suited it is for workloads such as, LU factorization, solving linear systems, sparse-matrix vector multiplication etc. as implemented in Intel Math Kernel Library (MKL), optimized for MIC architecture. With theoretical peak double precision performance of 1 TFlops, the Intel Xeon Phi 5110P variant was able to reach up to 80% of the peak working with dense matrices. However, there was heavy degradation in performance when it came to sparse matrix workloads, to a degree where multi-core CPU based execution was an order of magnitude faster. In this report, we present our findings for these experiments and architectural artifacts that guide the performance.

## 1. Introduction and Motivation

In form of Xeon Phi, Intel has come up with a solution for co-processor based accelerated computing currently dominated by General Purpose GPU based programming (GPGPU). In terms of architecture, and hence the level of parallelism offered, the Many Integrated Core (MIC) based Xeon Phi lie somewhere in between traditional multi-core CPUs and massively parallel GPUs. It has 60 cores running slightly older P54C core architecture, with beefed up vector processing unit (VPU). Physically, the VPU is an extension to the P54C core and communicates with the core to execute the VPU ISA implemented in Xeon Phi coprocessor.

Each core supports 4 hardware threads. The VPU receives its instructions from the core arithmetic logic unit (ALU) and receives the data from the L1 cache by a dedicated 512-bit bus. It is fully pipelined and can execute most instructions with 4-cycle latency with single-cycle throughput. Each VPU has 128 entry 512-bit vector registers divided up among the threads, thus getting 32 entries per thread [1]. Each vector can contain 16 single-precision floats or 32-bit integer elements; or eight 64-bit integer or double-precision floating point elements. This style of SIMD level parallelism distributed over 60 cores offers ample of opportunities to accelerate data parallel workloads. In reference to numerical computing, we will look at how well the card performs for some common BLAS subroutines, over varying data set sizes and with different threading configurations.

## 2. Methodology

The Xeon Phi is x86 compatible, meaning most programs are meant to work with slight modifications and compiler that supports the MIC architecture. There are two modes of operation - Offload and Native. As the name suggest, the offload mode offload certain parts of the program to the co-processor. Unlike, CUDA based NVidia GPUs, it is not required to separate kernels that run on the co-processor. However, the programming model is very much similar. Computation is offloaded to the threads running on co-processor with OpenMP style compiler directives [2]. This means there is explicit transfer of input and output data over PCIe interface. The thread control and data movement instructions are encapsulated in compiler directives. In native mode, the entire executable is compiled and executed on the coprocessor. This is realized by a minimalistic Linux based operating system that runs on the card. This mode requires that all the required input files and libraries be present the coprocessor.

In terms of parallelism, the work can be shared with up to 60 cores each with its own VPU. A finer, per-core level, parallelism is achieved by running 4 threads per core each of which can use the 512-bit wide VPU. Thus, even though it establishes a certain hierarchy in term of thread management, the level of parallelism offered is more towards coarse grained as finer levels of parallelism by explicitly controlling VPU usage on thread-by-thread basis is limited to using intrinsics. For our experiments we mainly worked the implementation of BLAS package that is part of Intel MKL version 11.1. This serves two purposes - MKL is industry standard for numerical and scientific computing for CPU based workloads. It also gives us an opportunity to work at

higher levels of abstraction than programming with intrinsics, allowing for more optimization opportunities for Intel C compiler (ICC) and with minimal programmer efforts, which Intel markets as one of the unique selling propositions for this particular product.

### 3. Implementation details

We conducted our study by analyzing following workloads:

- Benchmarks
- LU factorization
- Linear system solver
- Sparse matrix-vector multiplication

The Intel Math Kernel Library (MKL) has implementations of this routines that are optimized for MIC architecture. The code was compiled with version 11.1 of MKL and the compiler used was Intel C Compiler (ICC). ICC supports offload and thread control pragmas for Xeon Phi. Below we discuss the implementation in more details.

#### 3.1 Benchmarks

We studied benchmark performance in form of two application – Dense system solver that is part of LINPACK and double precision matrix-matrix multiplication (*dgemm*) micro-benchmark. The LINPACK benchmark was supplied as a binary which is part of MKL distribution. It varies the input size length, while keeping the other parameters such as number of threads fixed. To complement this, we wrote a micro-benchmark that had MKL *dgemm* routine operating over fixed size input but with varying number of threads.

#### 3.2 LU Factorization

We worked with factorization of both dense and banded matrices with routines *dgetrf* and *dgbtrf* respectively. The routine computes the LU factorization of a general m-by-n matrix A as

$$A = P*L*U$$

, where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if  $m > n$ ) and U is upper triangular (upper trapezoidal if  $m < n$ ). The routines uses partial pivoting, with row interchanges. The approximate number of floating-point operations with dense matrices (*dgetrf*) for real flavors is:

$$(2/3)n^3 \quad \text{If } m = n,$$

$$(1/3)n^2 \quad (3m-n) \text{ If } m > n,$$

$$(1/3)m^2 \quad (3n-m) \text{ If } m < n.$$

, where  $m$  and  $n$  are horizontal and vertical dimensions respectively. For banded matrices (*dgbrtf*), the routine forms the LU factorization of a general  $m$ -by- $n$  band matrix  $A$  with  $kl$  non-zero sub-diagonals and  $ku$  non-zero super-diagonals. The total number of floating-point operations for real flavors varies between approximately  $2n(ku+1)kl$  and  $2n(kl+ku+1)kl$ . The input matrix needs to be stored in band storage format. The band storage scheme is illustrated by the following example, when  $m = n = 6$ ,  $kl = 2$ ,  $ku = 1$ :

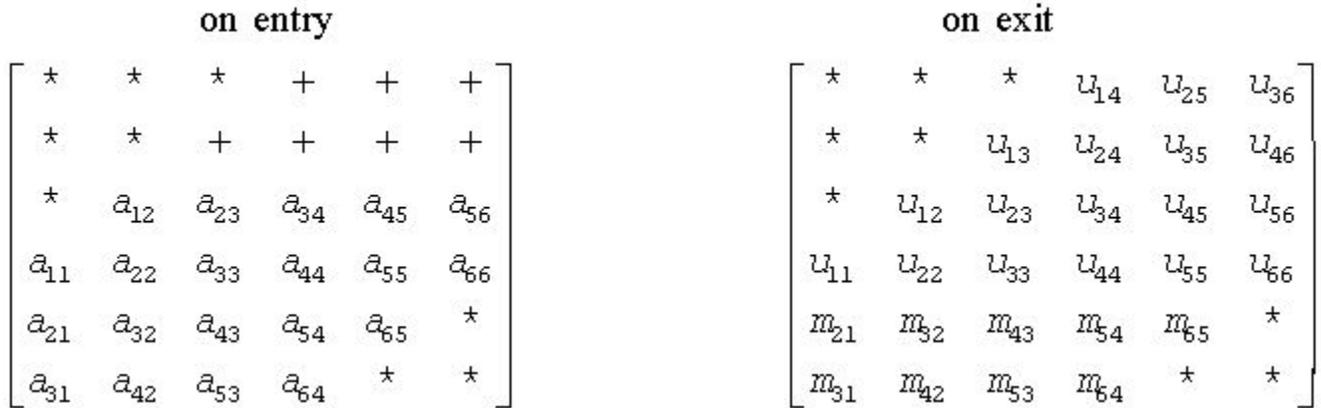


Figure 1 - LU Factorization Illustration

When running the experiments, the matrices were generated randomly with default seed. To give the fair idea of number crunching capabilities, timing measurements included only the computational parts of the routine and not the data movement overhead. Following code snippet show one such setup for the routine *dgbrtf*.

```
#pragma offload_transfer target(mic)
#pragma offload target(mic:0) in(n, kl, ku, lda) inout(a0:length(size_a) ALLOC)
in(ipiv:length(n) ALLOC)
{
    start = dsecnd();
    info = LAPACKE_dgbrtf(LAPACK_ROW_MAJOR, n, n, kl, ku, a0, lda, ipiv);
    end = dsecnd();
}
```

Similar setup was done for dense matrices.

```
#pragma offload_transfer target(mic)
#pragma offload target(mic:0) in(n, lda) inout(a0:length(size_a) ALLOC)
in(ipiv:length(n) ALLOC)
{
    start = dsecnd();
    info = LAPACKE_dgetrf(LAPACK_ROW_MAJOR, n, n, a0, lda, ipiv);
    end = dsecnd();
}
```

### 3.3 Linear system solver

The *dgesv* and *dgbsv* routines compute the solution to the system of linear equations with a square matrix *A* and multiple right-hand sides. The routines solve for *X* in the system of linear equations  $A \cdot X = B$ , where *A* is an *n*-by-*n* matrix, the columns of matrix *B* are individual right-hand sides, and the columns of *X* are the corresponding solutions. For our experiments, number of right-hand sides was fixed to one, meaning *X* and *B* are column vectors. These routines act as drivers for LU factorization routines (describe above) followed by system solver phase (*dgetrs* / *dgbrs*). Thus, there are additional floating-point operations -  $2n^2$  for real flavors of one right-hand side vector *b* in case of dense matrices. The additional number of floating-point operations for one right-hand side vector is  $2n(ku + 2kl)$  for real flavors. For these programs, same set of compilation flags as with LU factorization was used. Following are the example usages:

```
#pragma offload target(mic:0) in(n, nrhs, lda, ldb) in(a:length(size_a) ALLOC)
in(ipiv:length(n) ALLOC) inout(b:length(n) ALLOC)
{
    start = dsecnd();
    info = LAPACKE_dgesv( LAPACK_ROW_MAJOR, n, nrhs, a, lda, ipiv, b, ldb );
    end = dsecnd();
}
```

```
#pragma offload target(mic:0) in(n, kl, ku, nrhs, lda, ldb) in(a:length(size_a) ALLOC)
in(ipiv:length(n) ALLOC) in(b:length(n) ALLOC)
{
    start = dsecnd();
    info = LAPACKE_dgbsv( LAPACK_ROW_MAJOR, n, kl, ku, nrhs, a, lda, ipiv, b, ldb );
    end = dsecnd();
}
```

### 3.4 Sparse matrix-vector multiplication

The routine *mkl\_dcsrsv* computes matrix - vector product of a sparse matrix, with the operation defined as

$$y := \alpha \cdot A \cdot x + \beta \cdot y$$

, where  $\alpha$  and  $\beta$  are scalars, *x* and *y* are vectors, *A* is an *m*-by-*k* sparse matrix in the CSR format [3]. The number of floating point operations is twice the number of non-zero elements in the sparse matrix. As the inputs, we used a set of matrices from The University of Florida Sparse Matrix Collection.

Given below is the usage:

```
time_start = dsecnd();
mkl_dcsrsv( &transa, (int*)&csrMatrix->num_rows, (int*)&csrMatrix->num_cols, &alpha,
matdescra,
csrMatrix->vals, csrMatrix->cols, csrMatrix->rows, &csrMatrix->rows[1], x, &beta, y );
time_end = dsecnd();
time = time_end - time_start;
```

In all the above runs, following were the specifications for hardware used:

Phi: Intel® Xeon Phi™ Coprocessor 5110P (8GB, 1.053 GHz, 60 core), all tests in offload mode.

CPU: Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz, Dual socket chip, 20 cores up to 40 threads

Following compiler flags were used across all the tests:

```
-xHOST -O3 -mkl -parallel -opt-prefetch=2 -opt-streaming-stores auto -fp-model fast
```

## 4. Results

### 4.1 Benchmarks

As mentioned above, first benchmark implements dense system solver. The dimension of input matrix (N) was varied from 1000 to 45000. Since Xeon Phi has only 8 gigabytes of memory available, the N was limited to little over 28.5 elements. (28672 to be precise, adjusting for recommended padding of elements). For smaller sizes, such as  $N < 4000$ , the performance with CPU is better. However, for larger sizes, Xeon Phi offers better performance – reaching up to 756 GFlops for maximum possible input size (fig. 2). It can be seen that having larger data sets is important in getting good performance out of Xeon Phi, among other things.

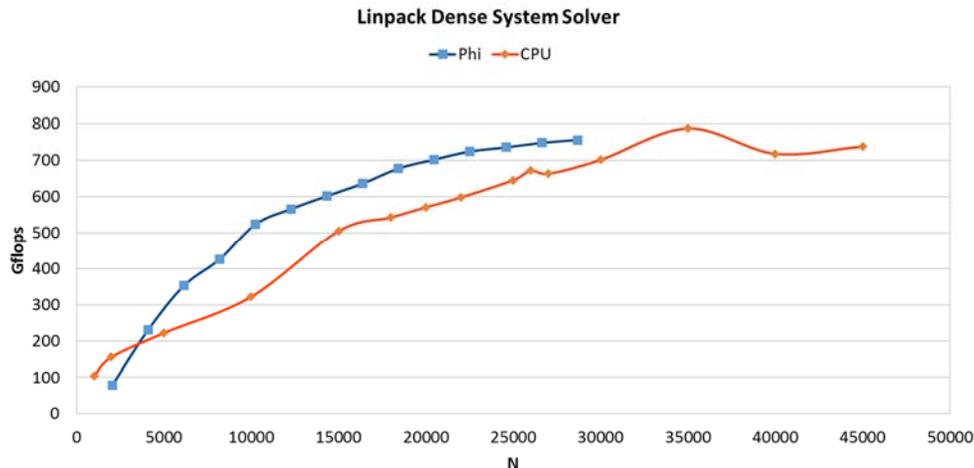


Figure 2 - Linpack Dense System Solver

For the *dgemm* micro-benchmark, we varied the no. of threads – 1 to 40 for CPU and 1 to 240 of Xeon Phi, while keeping input matrix size fixed to 8192 x 8192 elements. This result shows that having more number of threads helps with extracting more parallelism out of given problem. Another observation is that having correct number of threads is important for doing proper load balancing – performance drops for odd number of threads (fig. 3).

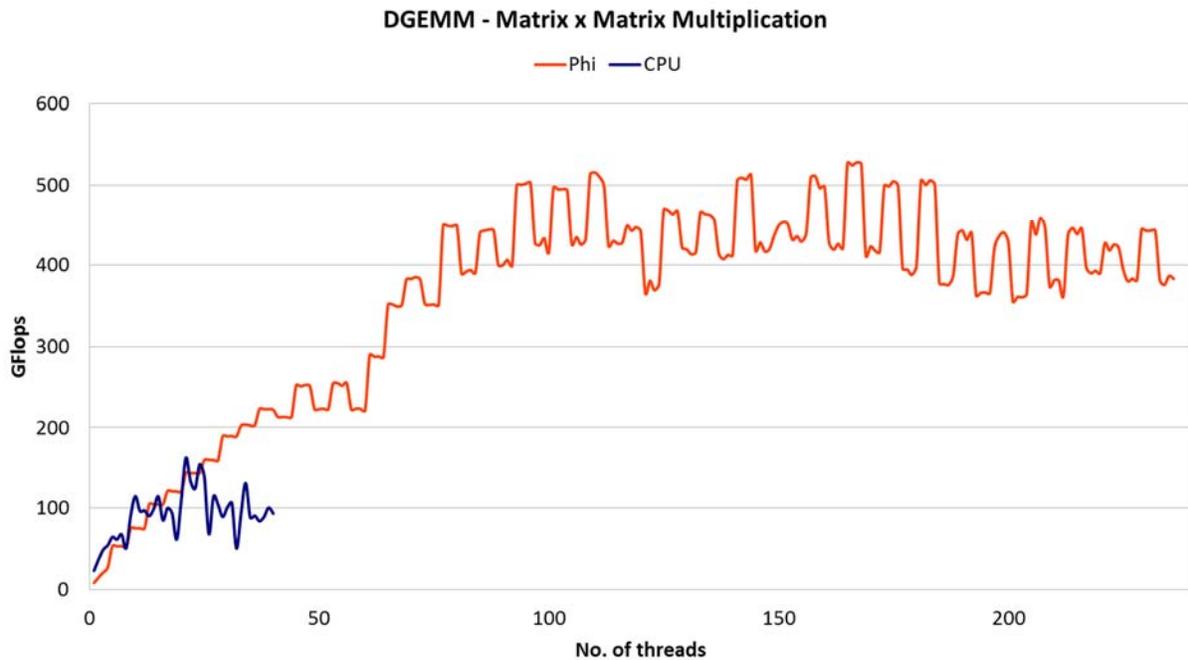


Figure 3 - Dense Matrix-Matrix Multiplication Microbenchmark

## 4.2 LU Factorization

For dense matrices, the performance of Xeon Phi is worse than that of CPU roughly by factor of 2. However, the difference is more pronounced in case of sparse matrices, with relative performance differing roughly by 15 to 20 times (fig. 5, 6). One subtle observation is that for certain value of N, there is noticeable drop GFlops rates. This is because improper leading dimension results in imbalanced partitioning of the data among threads. As for the effect of bandwidth variations, it can be seen that having larger bandwidths – in effect reducing the amount of sparsity – helps both CPU and Xeon Phi.

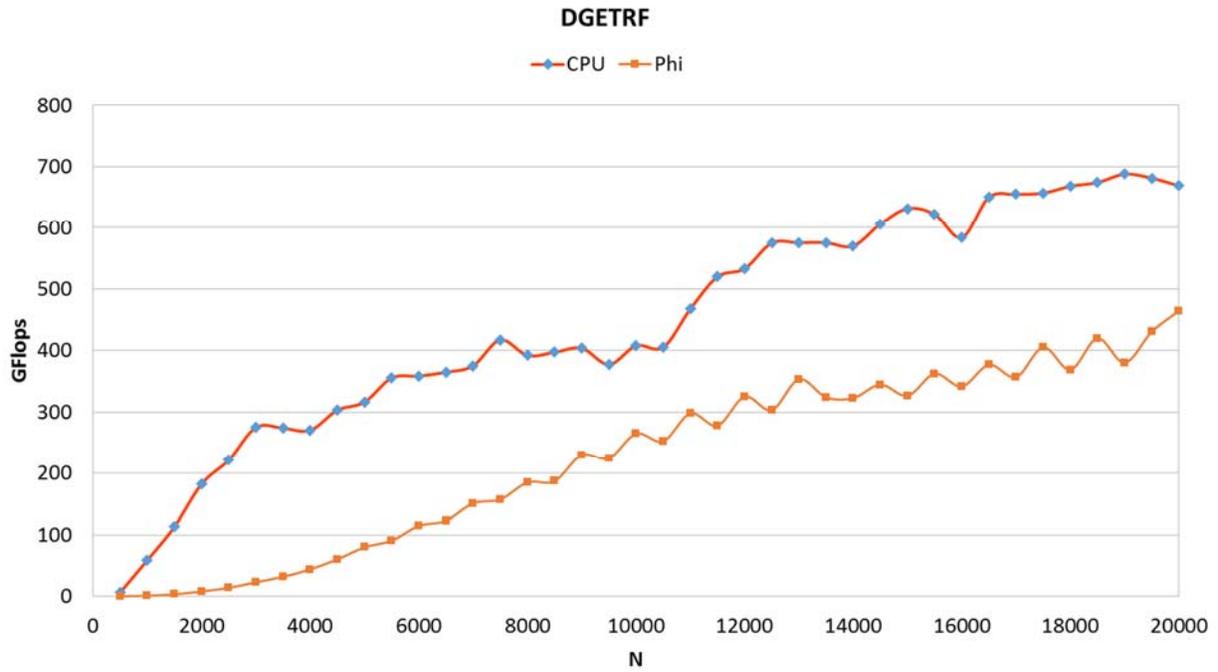


Figure 4 - LU Factorization: Dense Matrices

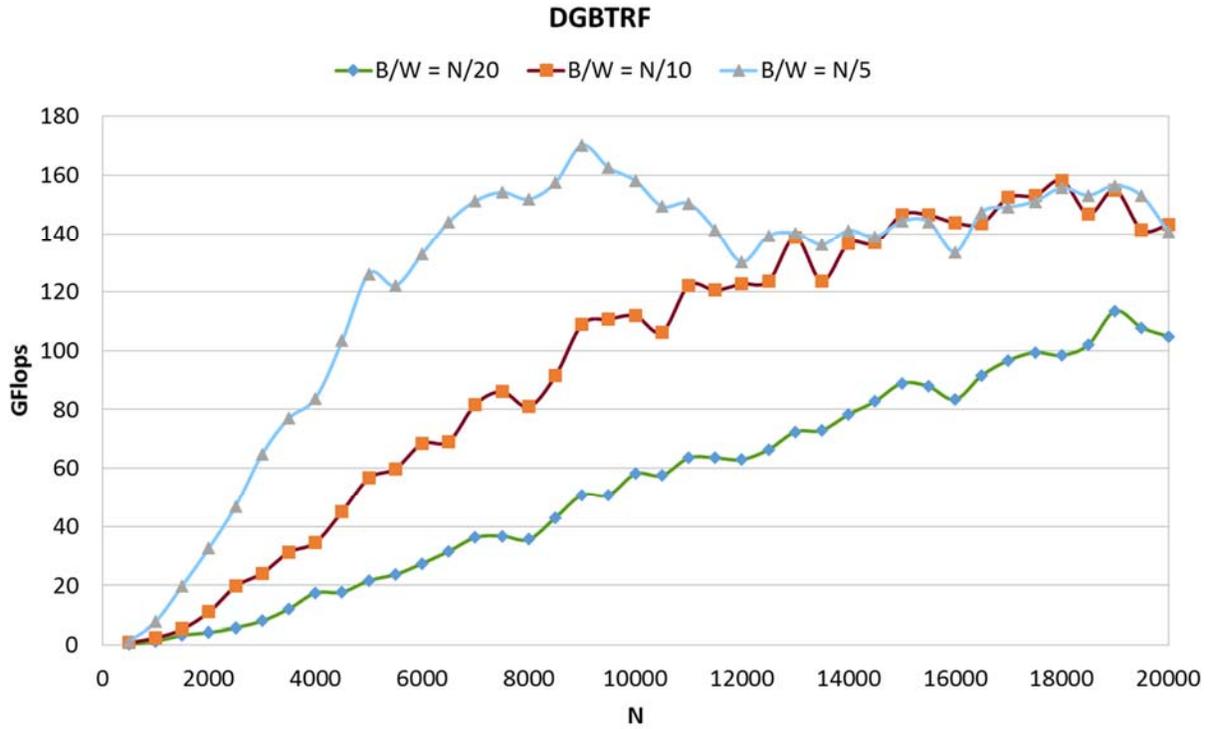


Figure 5 - LU Factorization on CPU: Banded Matrices

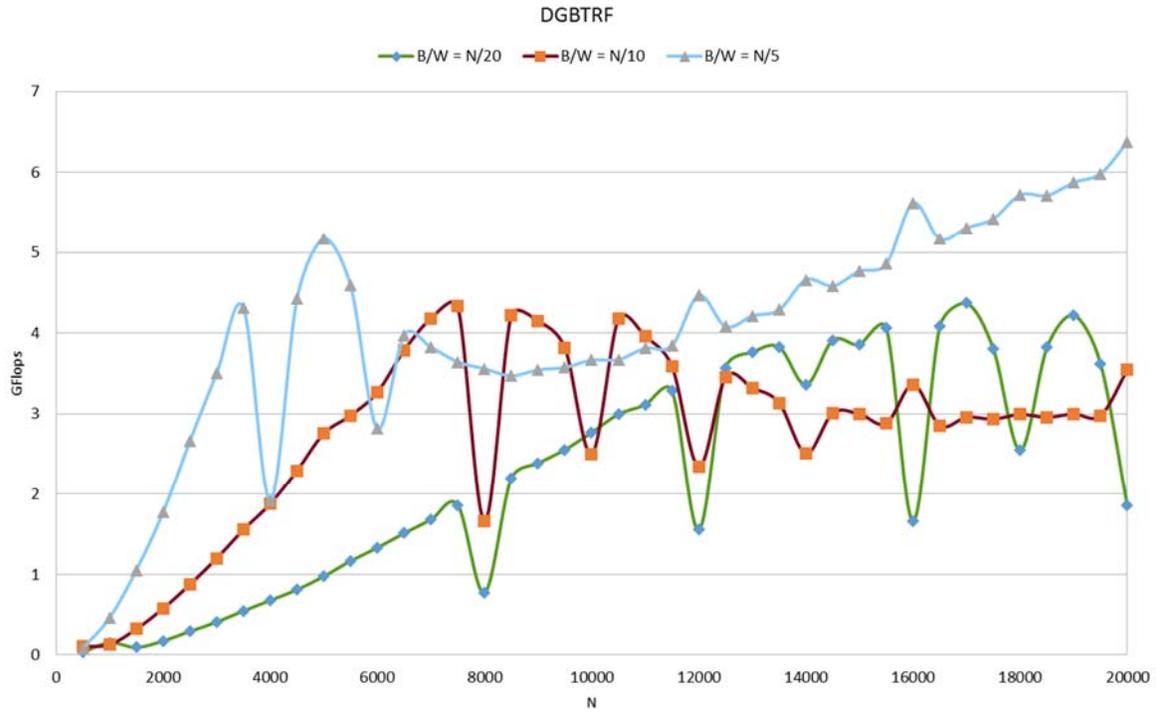


Figure 6 - LU Factorization on Phi: Banded Matrices

### 4.3 Linear system solver

For linear system solver, results' profile was similar to that of LU factorization. This is mainly because it only adds  $O(N^2)$  level of complexity to  $O(N^3)$  complex algorithm (fig. 7, 8).

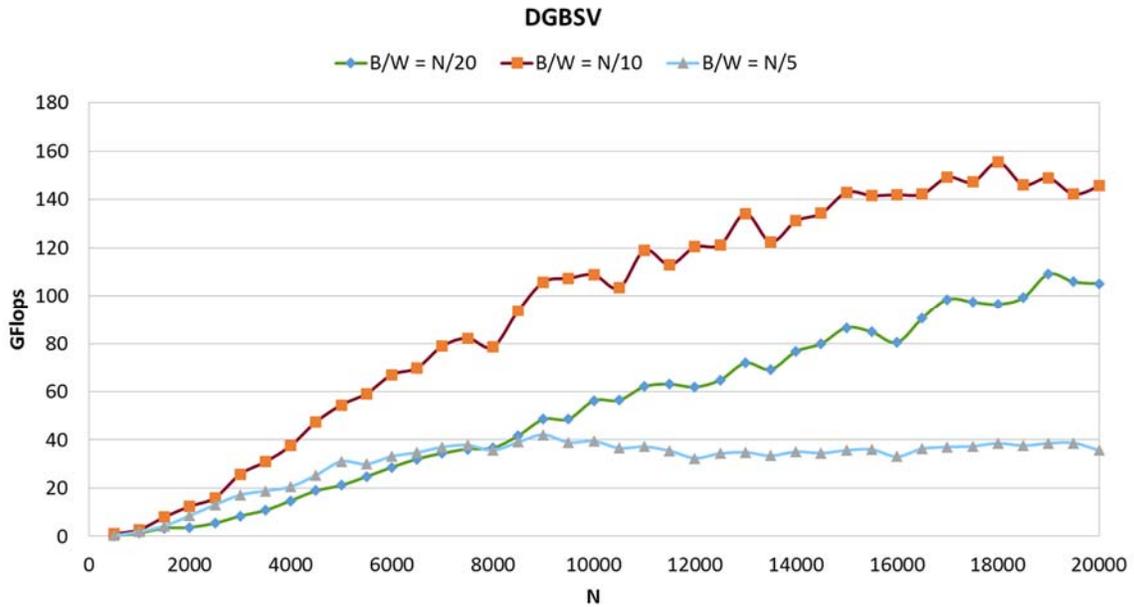


Figure 7 - Linear system solver on CPU: Banded Matrices

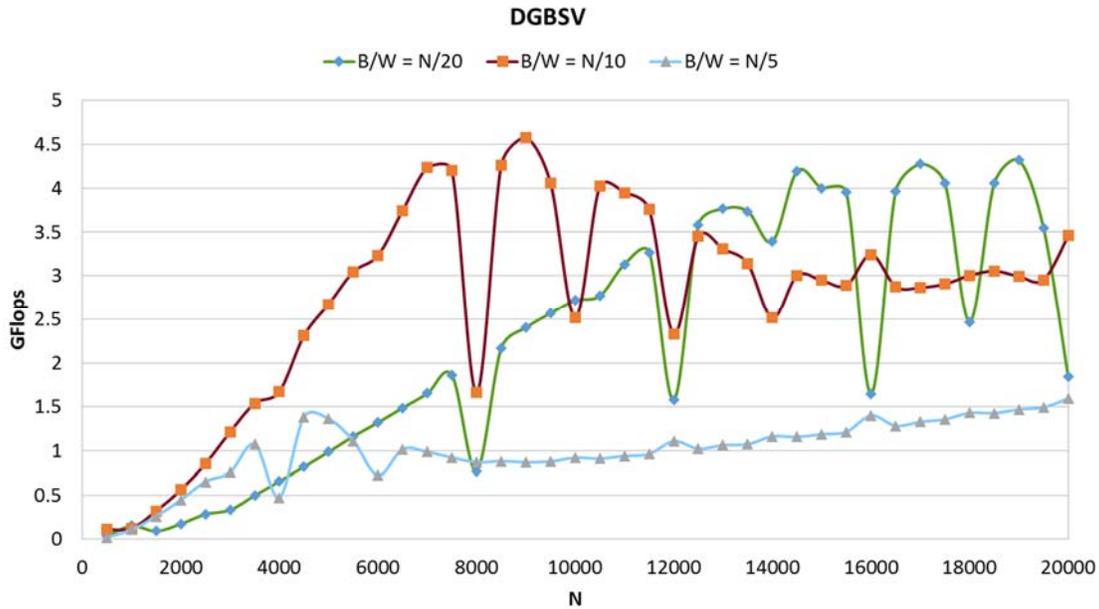


Figure 8 - Linear system solver on Phi: Banded Matrices

#### 4.4 Sparse matrix-vector multiplication

As opposed to results published in, we did comparison of Xeon Phi with CPU for similar set of input matrices. We found out that results to be a mixed bag (fig. 9). Depending upon the type of matrix, about half of the cases resulted in Xeon Phi outperforming the CPU. Generally speaking, denser matrices performed better, though it remains to be seen precisely which parameters and input configurations guide the overall performance. We include this in our future work.

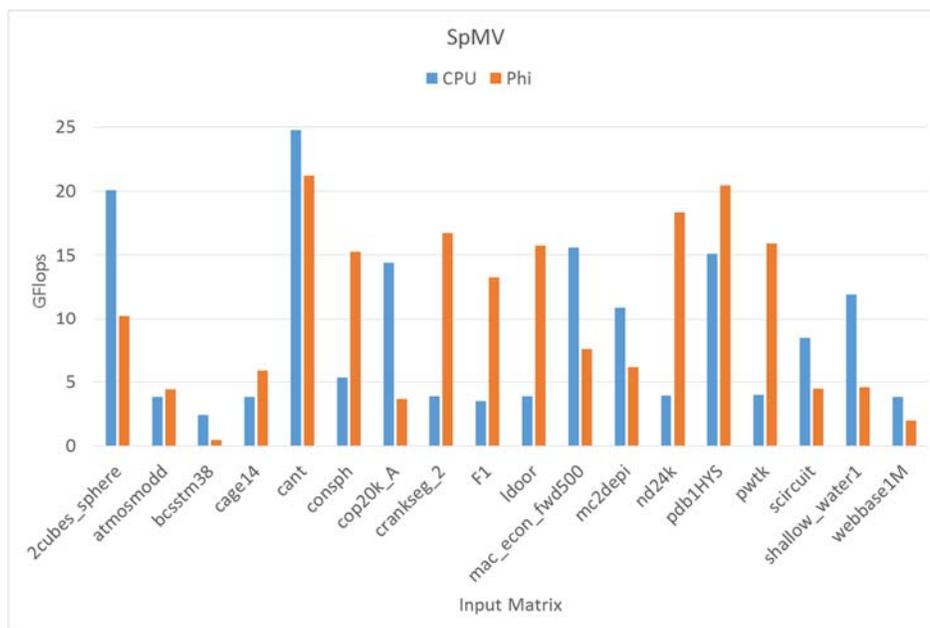


Figure 9 - SpMV Comparison

## 4.5 Effects of optimizations

Intel recommends certain optimization guidelines to improve performance of MKL on Xeon Phi coprocessors, which are specific to Intel MIC Architecture. Performance of many Intel MKL routines can potentially improve when input and output data reside in memory allocated with 2MB pages. This enables one to address more memory with less pages and thus reduce the overhead of translating between virtual and physical memory addresses compared to memory allocated with the default page size of 4K. For thread binding it is recommended that to use compact binding and fine level of granularity [5]. As far as data alignment and leading dimensions are concerned, following criteria needs to be met:

- Align the first element of the input data on 64-byte boundaries
- For two- or higher-dimensional single-precision transforms, use leading dimensions (strides) divisible by 8 but not divisible by 16
- For two- or higher-dimensional double-precision transforms, use leading dimensions divisible by 4 but not divisible by 8

More information can be found in [4].

As can be seen in figure 10, there 2MB pages provide only marginal improvement over 4K pages. The effect of leading dimensions on the other hand is much more pronounced. The effects more fine level of variation can be seen in figure 11.

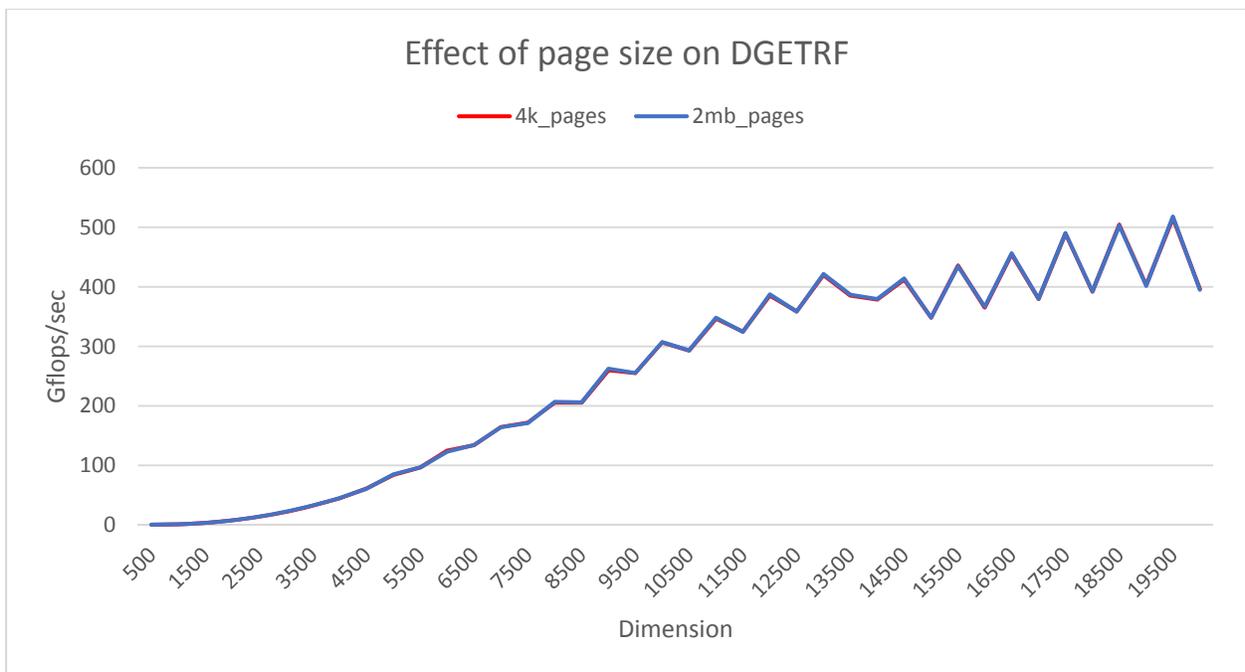


Figure 10 - Effect of Page Size on Performance

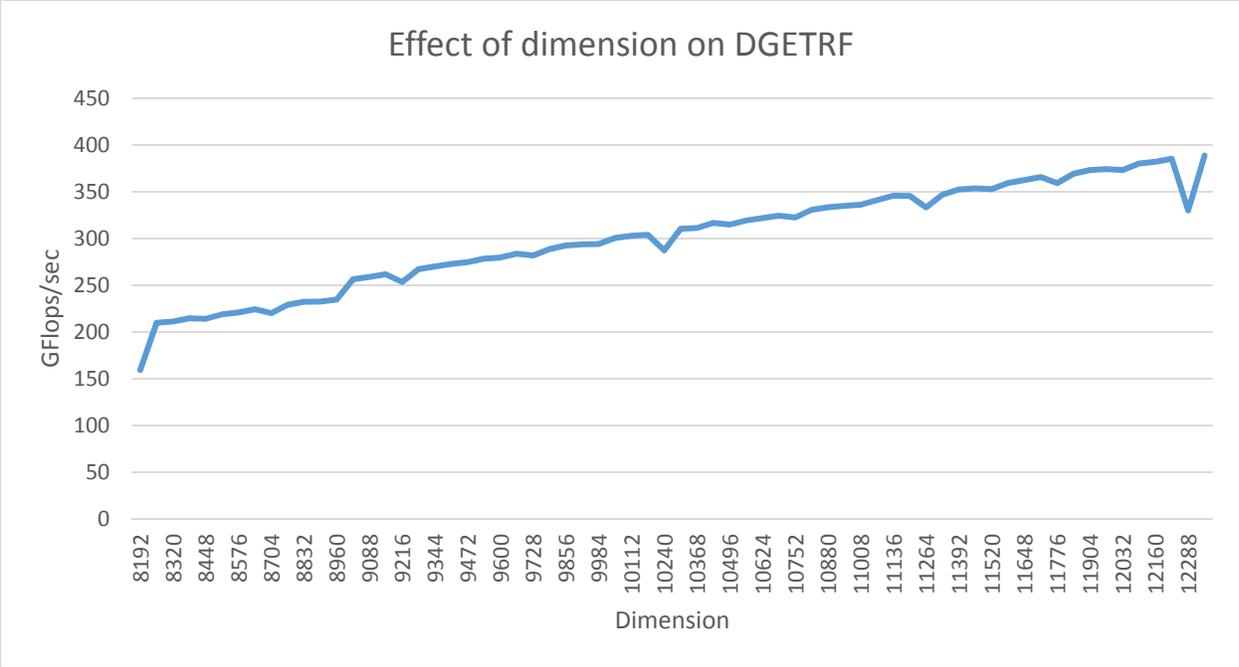
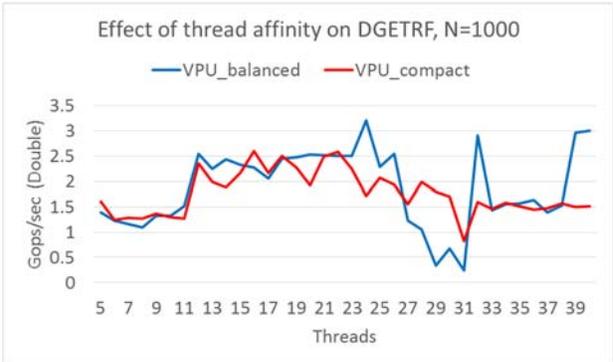
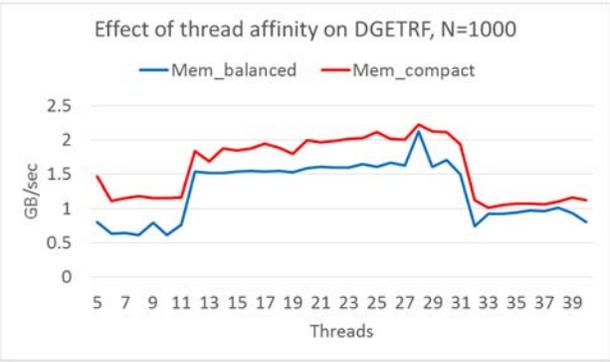


Figure 11 - Effect of Leading dimension on Performance

To understand the effect of thread binding, we chose a small matrix input size with  $N = 1000$ . Here we see that, compact bindings offer better utilization of available memory bandwidth. However, due sharing of single VPU among multiple thread, the overall utilization of VPU capacity is less compared to balanced bindings. As a results, net performance in terms GFlops is better with balanced bindings (fig. 12).



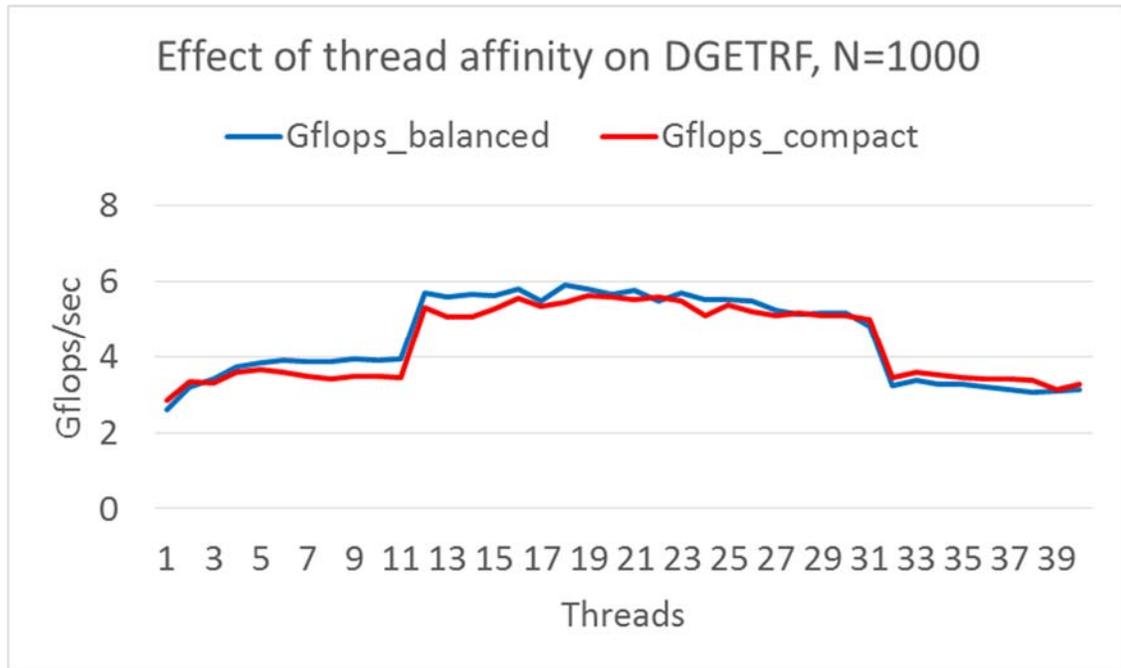


Figure 12 - Effect of Thread Affinity on Performance

## 5. Conclusion and Future Work

It can be safely said that as far as linear algebraic workloads are concerned, Xeon Phi is well suited only for certain subset of it. Particularly, it works well when it comes to dense matrices in general. However, the performance drops by a large magnitude when it comes to sparse matrices. The main reason behind this is the problem shifts its nature from compute-bound to memory-bound, resulting in reduced VPU usage. We believe the MIC architecture offers breadth of programming models that accelerated the adoption of this platforms. However, as far as the performance is concerned, there are large gaps that it needs to fill in. The P54C core is outdated, with no support for modern instruction level parallelism features such as out-of-order execution. Additionally, it is still dependent on data transfers over PCIe, which turns out to be a bottleneck if there is a lot of data movement involved. The next version of Xeon Phi has potential to address both these issues, as it is supposed to come out with more up-to-date Atom based core and support to run as a standalone machine as opposed to a co-processor [6]. In light of such advancements, we believe it will be interesting to revisit this type of study to understand what benefits these improvements can provide for scientific computing community.

## References

- [1] Intel® Xeon Phi Coprocessor Vector Microarchitecture  
<http://software.intel.com/sites/default/files/article/393199/intel-xeon-phi-coprocessor-vector-microarchitecture.pdf>
- [2] Best Known Methods for Using OpenMP on Intel Many Integrated Core (Intel® MIC) Architecture  
<https://software.intel.com/en-us/articles/best-known-methods-for-using-openmp-on-intel-many-integrated-core-intel-mic-architecture>
- [3] The Intel Math Kernel Library Sparse Matrix Vector Multiply Format Prototype Package  
<https://software.intel.com/en-us/articles/the-intel-math-kernel-library-sparse-matrix-vector-multiply-format-prototype-package>
- [4] Improving Performance on Intel Xeon Phi Coprocessors  
[https://software.intel.com/sites/products/documentation/doclib/mkl\\_sa/11/mkl\\_userguide\\_lnx/GUID-7F67CA7F-0B2D-4584-AE90-63A64A58F0EC.htm](https://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mkl_userguide_lnx/GUID-7F67CA7F-0B2D-4584-AE90-63A64A58F0EC.htm)
- [5] OpenMP Thread Affinity Control  
<https://software.intel.com/en-us/articles/openmp-thread-affinity-control>
- [6] Knights Landing Details  
<http://www.realworldtech.com/knights-landing-details/>