# On the Design, Development, and Analysis of Optimized Matrix-Vector Multiplication Routines for Coprocessors

Khairul Kabir[1], Azzam Haidar[1], Stanimire Tomov[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee Knoxville
[2] Oak Ridge National Laboratory
[3] University of Manchester

**Abstract.** The dramatic change in computer architecture due to the manycore paradigm shift, made the development of numerical routines that are optimal extremely challenging. In this work, we target the development of numerical algorithms and implementations for Xeon Phi coprocessor architecture designs. In particular, we examine and optimize the general and symmetric matrix-vector multiplication routines (gemv/symv), which are some of the most heavily used linear algebra kernels in many important engineering and physics applications. We describe a successful approach on how to address the challenges for this problem, starting from our algorithm design, performance analysis and programing model, to kernel optimization. Our goal, by targeting low-level, easy to understand fundamental kernels, is to develop new optimization strategies that can be effective elsewhere for the use on many-core coprocessors, and to show significant performance improvements compared to the existing state-of-the-art implementations. Therefore, in addition to the new optimization strategies, analysis, and optimal performance results, we finally present the significance of using these routines/strategies to accelerate higher-level numerical algorithms for the eigenvalue problem (EVP) and the singular value decomposition (SVD) that by themselves are foundational for many important applications.

## 1 Introduction

As the era of computer architectures dominated by serial processors closes, the scientific community has produced a consensus for the need to redesign numerical libraries to meet the new system design constraints and revolutionary levels of parallelism and heterogeneity. One approach, from the early days of the multicore architectures, was to redesign the higher-level algorithms, e.g., LAPACK [4], to use tile operations [6, 7, 8]. To provide parallelism in these algorithms, the computation is expressed as a Directed Acyclic Graph (DAG) of tasks on small matrices/tiles with labeled edges designating data dependencies, and a run-time system schedules the DAG's execution over the cores insuring that data dependancies are not violated. Performance relied on fast sequential implementations of the Basic Linear Algebra Subprograms (BLAS) interface [12].

When manycore accelerators entered the HPC field, it become apparent that braking the uniformity of the computation is not advantegeous for GPUs. Instead, hybrid approaches were developed [3, 16, 25, 26, 28], where there is still a DAG and scheduling (for both GPUs and CPUs), but SIMD tasks on large data that are suitable for GPUs, e.g., GEMM, remain coarse grained and are scheduled as single tasks for parallel execution through parallel BLAS implementations. This highlighted the interest in parallel BLAS, and subsequently to parallel BLAS implementations in CUBLAS [1] and MAGMA BLAS [2]. Hybrid approaches are also very suitable for the more recent many-core coprocessors, e.g., as evident from the MAGMA MIC's extension of MAGMA for the Xeon Phi coprocessors [13, 18].

An extension to the parallel BLAS in many HPC applications, that is a subject of current interest and development, is the use of batched BLAS [9,10,17,19]. Batched algorithms address one of the significant challenges in HPC today – that numerous important applications tend to be cast in terms of a solution to many small matrix operations: they contain very large majority of computations that consist of a large number of small matrices, which cannot be executed efficiently on accelerated platforms except in large groups, or "batches". Indeed, batched representation of computational tasks is pervasive in numerical algorithms for scientific computing. In addition to dense linear algebra routines and applications, batched LA can naturally express various register and cache blocking techniques for sparse computations [20], sparse direct multifrontal solvers [29], high-order FEM [11], and numerous applications in astrophysics [23], hydrodynamics [11], image processing [24], signal processing [5], and big data, to name just a few. Moreover, blocking for cache reuse - the most basic technique to accelerate numerical algorithms from the fundamental dense matrix-matrix product, to sparse matrix-vector (SpMV), to more complex linear or eigenvalue solvers – is often synonymous to a batched representation of algorithms.

To enable the effective use of parallel BLAS and batched BLAS-based computational approaches, new parallel BLAS algorithms and optimization strategies must be developed. In this work, we target the development of these foundational numerical algorithms, optimization strategies, and implementations for the Xeon Phi coprocessors, known also as Intel's many integrated core architectures (MIC). In particular, we examine and optimize the general and symmetric matrix-vector multiplication routines (gemv/symv), which are some of the most heavily used linear algebra kernels in many important engineering and physics applications. Our goal, by targeting low-level, easy to understand fundamental kernels, is to develop optimization strategies that can be effective elsewhere, and in particular for batched approaches for HPC applications on manycore coprocessors. Therefore, we developed new optimization strategies (and analysis) to obtain optimal performance. Finally, we illustrate the need and the significance of using these routines/strategies to accelerate higher-level numerical algorithms for the EVP and SVD problems that by themselves are foundational for many important applications.

## 2 Background

This paper addresses two kernels – the general and the symmetric matrix-vector multiplications (gemv and symv) – which are crucial for the performance of linear solvers as well as EVP and SVD problems. A reference implementation for a generic matrix-vector multiplication kernel is straight-forward because of the data parallel nature of the computation. However, achieving performance on accelerators or coprocessors is challenging, as evident from the results on current state-of-the-art implementations. For example, even though Intel optimized dgemv in their recent release of MKL, its performance is highly nonuniform, reaching up to about 37-40 Gflop/s for only particular matrix sizes and data alignments. Performance, when matrix size is not multiple of the cache line (8 double precision numbers), drops by about 10 Gflop/s, or 30% of the peak obtained. Furthermore, a sequence of calls to dgemv with "transpose" and "Non transpose" have shown a drop in the performance as well at about 10 Gflop/s. In addition to the issues for the dgemv kernel, the irregular data access patterns in the symv routine bring further challenges for its design and optimization. For example, the current MKL dsymv achieves the same performance as the dgemv ($\approx$ 37-40 Gflop/s) while in theory it should be twice faster.

In this paper, we describe the optimizations performed to both the gemv and symv routines to make them reach their theoretical peak performances on coprocessors. Our gemv kernel is not affected by neither the matrix size nor the sequence of calls. It achieves uniform performance that matches the peaks of the MKL's gemv. This improvement was important to speed up many algorithms, and in particular, the reduction to bidiagonal form which is a major component for SVD.

An optimality analysis for the symv routines shows (see Section 6) that this kernel should achieve twice the performance of the gemv routine. We developed an algorithm (and its implementation) that exploits cache memory to read small blocks of the matrix in cache and reuse them in the computation involving their symmetric counterparts. This implementation divides the main memory reads in half and our experiments show that it reaches to around 50-55 Gflop/s for specific blocking sizes that allow each small block to fit into the L2 cache of a corresponding core of the coprocessor. Even though this new symv kernel brings an excellent improvement over the contemporary MKL, it is still less than what the performance bound analysis shows as being possible. This motivated us to look for further improvements that lead to the development of a second algorithm (and it implementation) that reuses the data loaded into the L1 cache level, as well as from registry, to reach to around 65 Gflop/s. We should mention that both of our symv implementations incur memory overheads of less than one percent (about 0.78%) of the matrix size. We also show the impact that this optimization have on the tridiagonal reduction which is the most time consuming component of the symmetric eigenvalue problem.

## 3 Contribution

The evolution of semiconductor technology is dramatically transforming the balance of future computer systems, producing unprecedented changes at every level of the platform pyramid. From the point of view of numerical libraries, and the myriad of applications that depend on them, three challenges stand out: 1) the need to exploit unprecedented amounts of parallelism; 2) the need to maximize the use of data locality; and 3) the need to cope with component heterogeneity. Besides the software development efforts that we investigate to accomplish an efficient implementation, we highlight our main contributions related to the algorithm's design and optimization strategies towards addressing these challenges on the MIC architecture:

***Exploit unprecedented amounts of parallelism***: Clock frequencies are expected to stay constant, or even decrease to conserve power; consequently, as we already see, the primary method of increasing computational capability of a chip will be to dramatically increase the number of processing units (cores), which in turn will require an increase of orders of magnitude in the amount of concurrency that routines must be able to utilize. We developed MIC-specific optimization techniques that demonstrate how to use the many (currently 60) cores of the MIC to get optimal performance. The techniques and kernels developed are fundamental and can be used elsewhere;

***Hierarchical communication model that maximize the use of data locality***: Recent reports (e.g., [15]) have made it clear that time per flop, network bandwidth (between parallel processors), and network latency are all improving, but at exponentially different rates. So an algorithm that is computation-bound and running close to peak today may be communication-bound in the near future. The same holds for communication between levels of the memory hierarchy. We demonstrate that related to the latter, performance is indeed harder to get on new manycore architectures unless hierarchical communications are applied. Hierarchical communications to get top speed now are needed not only for Level 3 BLAS but also for Level 2 BLAS, as we show. Only after we developed and applied multilevel cache blocking, our implementations reached optimal performance;

***Performance bounds analysis***: We study and demonstrate the maximal performance bounds that could be reached. The performance bounds allow us to ascertain the effectiveness of our implementation and how close it approaches the theoretical limit. We developed and demonstrated this use of performance bound analysis not only for the low-level kernels considered, but also for the higher-level algorithms that use them as building blocks.

## 4 Experimental Testbed

All experiments are done on an Intel multicore system with two 8-core Intel Xeon E5-2670 (Sandy Bridge) CPUs, running at 2.6 GHz. Each CPU has a 24

MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1 caches. The system is equipped with 52 GB of memory. The theoretical peak in double precision is 20.8 Gflop/s per core, giving 332 Gflop/s in total. The system is equiped with one Intel Xeon-Phi KNC 7120 coprocessor. It has 15.1 GB, runs at 1.23 GHz, and yields a theoretical double precision peak of 1,208 Gflop/s. We used the MPSS 2.1.5889-16 software stack, the icc compiler that comes with the composer_xe_2013_sp1.2.144 suite, and BLAS implementation from MKL (Math Kernel Library) 11.01.02 [21].

## 5 The general matrix-vector multiplication routine gemv

Level 2 BLAS routines are of low computational intensity and therefore DLA algorithms designers try to avoid them. There are techniques that can replace Level 2 BLAS operations with Level 3 BLAS. For example, in factorizations like LU, QR, and Cholesky, the application of consecutive Level 2 BLAS operations that occur in the algorithms can be delayed and accumulated so that at a later moment the accumulated transformation can be applied at once as a Level 3 BLAS [4]. This approach totally removes Level 2 BLAS from Cholesky, and reduces its amount to $O(n^2)$ in LU, and QR, thus making it asymptotically insignificant compared to the total $O(n^3)$ amount of operations for these factorizations. The same technique can be applied to the two-sided factorizations [14], but in contrast to the one-sided, a large fraction of the total number of floating point operations (flops) still remains Level 2 BLAS. For example, the block Hessenberg reduction has about 20% of its flops in Level 2 BLAS, while both the bidiagonal and tridiagonal reductions have 50% of their flops in Level 2 BLAS [27]. In practice, the flops in Level 2 BLAS do not scale well on current architectures and thus can significantly impact the total execution time. Therefore the availability of their efficient implementations is still crucial for the performance of two sided factorization in current architecture. This section considers the Xeon Phi implementation of one fundamental Level 2 BLAS operation, namely the matrix-vector multiplication routine for general dense matrices (gemv). The gemv multiplication routine performs one of:

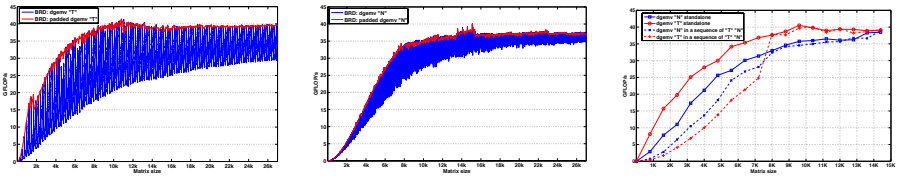
$$y := \alpha Ax + \beta y, \quad \text{or} \quad y := \alpha A^T x + \beta y, \tag{1}$$

where A is an M by N matrix, x and y are vectors, and $\alpha$ and $\beta$ are scalars.

### 5.1 Effect of the matrix size on the MKL gemv performance

The gemv performance peak on the Xeon-Phi coprocessor is as expected – achieving around 37-40 GFlop/s in double precision for both of its transpose and non-transpose cases, which translate to a bandwidth of about 160 GB/s. Achieving this bandwidth is what is expected on the Phi coprocessor [22]. However, this peak performance is obtained only on particular matrix sizes and data alignments. In reality, applications that exclusively rely on the gemv, e.g., the bidiagonal reduction (BRD), show a much lower performance. Our analysis shows

that in the particular case of the BRD (see equation (7)), performance must be about twice the performance of the gemv, while experiments show that the BRD attains less than 37-40 GFlop/s. A detailed analysis of the gemv kernel shows that its performance indeed highly depends on the location of the data in the memory, and in particular, on its alignment. We benchmarked gemv on matrices of consecutively increasing sizes from 1 to 27 K, similarly to the way that the BRD reduction calls it. We found out that its performance fluctuates as shown in Figure 1 (the blue curves) according to the offset from which the matrix is accessed. The performance drops by about 15 Gflop/s for the transposed case when the matrix size in the "$n$" dimension is not multiple of 240 (as shown in Figure 1a) and falls by about 10 Gflop/s for the non-transposed case when the matrix size in the $m$ dimension is not multiple of 8, as depicted in Figure 1b. To resolve the dependance on the memory alignment and the matrix sizes, we developed two routines (for the transpose and non-transpose cases, respectively) that always accesses a matrix from its aligned data, performing a very small amount of extra work but keeping its performance stable at its peak. The red curves in Figure 1 show our improvement. The algorithms are described in subsection 5.3 below.



(a) Performance of dgemv_T w/o the proposed virtual padding.

(b) Performance of dgemv_N w/o the proposed virtual padding.

(c) Performance of a sequence of dgemv_T and dgemv_N calls.

**Fig. 1.** Performance obtained from the dgemv routine on matrices of consecutively increasing sizes.

### 5.2 Effect of the sequence of gemv calls

After getting an optimal performance for the gemv's both transpose and non-transpose cases, as described in Section 5.1, we tested their use in real-world applications. For the BRD reduction for example, performance is improved for all sizes and reaches its theoretical peak for large matrix sizes. However, the performance for small sizes, in particular less than $8K$, is not as expected. The detailed experiments depicted in Figure 1c show that performance of gemv drops by 10 Gflop/s when called in a sequence of non-transpose followed by transpose cases for matrices of size less than $8K$. We believe that this is related to different parallelization grid used for each case of gemv (transpose vs. non-transpose) and thus this is the overhead of switching between the two different grids of

cores. The overhead probably exists always for larger sizes but its effect is less evident because the cost of the gemv is dominant. To overcome this drawback, we introduce another optimization technique and use it to develop a new gemv routine, described in detail in the following sections.

### 5.3 A new MAGMA MIC gemv

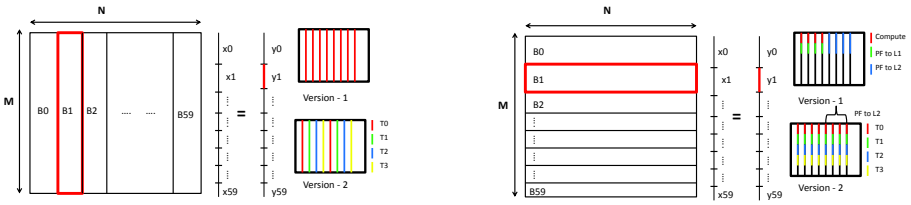
**Transpose matrix case:** The computation of the gemv routine for the transpose case can be parallelized in one-dimensional (1D) block-column fashion. In this parallelization model each thread processes its part of the gemv column by column, and thus for each column a dot product is performed. The accumulations are done in cache and the final, resulting vector $y$ is written once. Moreover, each thread reads data that is stored consecutively in memory, which will simplify the prefetching and vectorization process. To get good performance out of a MIC core, vectorization that takes advantage of the core's 16-wide SIMD registers is essential. Each core processes one block (or multiple, if we use 1D block cyclic distribution). The number of columns in $\text{block}_i$ can be set for example as:

$$columns\_in\_block_i = \frac{N}{num\_blocks} + (i < (N\%num\_blocks)\ ?\ 1 : 0), \quad (2)$$

where $num\_blocks$ is the number of blocks (e.g., 60 to correspond to 60 cores of a MIC) that we want N columns to split into, and $i = 1, \ldots, num\_blocks$ is the block index. We developed parametrized implementations and hand-tested them at first to get insight for the tuning possibilities. For example one parameter is number of threads per core. Figure 2a for example illustrates a distribution using one thread per core (displayed as version-1) and four threads per core (displayed as version-2). In this case, we found that both implementations provide the same performance. This is due to the fact that the gemv routine is memory bound and one thread per core is enough to saturate the bandwidth, thus increasing the number of threads does not affect the performance.

**Non-transpose matrix case:** Following the same strategy used for the transpose approach leads to poor performance for the non-transpose case. This is because the values of y need to be written multiple times in this case. Therefore, we can instead parallelize the algorithm in 1D block-row fashion. In this way each core processes its independent part of the gemv and thus the resulting vectors can be accumulated in cache and written to the main memory only once. To keep the blocks cache aligned, their size can be made to be a multiple of eight. For effective load balance we can think of the matrix as strips of size eight, and divide the strips among the block-rows equally. In this case the number of rows in $\text{block}_i$ can be set for example as:

$$\begin{aligned} m8\_strip &= (M + 7)/8 \\ rows\_in\_block_i &= [\frac{m8\_strip}{num\_blocks} + (i < (m8\_strip\%num\_blocks)\ ?\ 1 : 0)] \times 8 \end{aligned} \quad (3)$$




(a) Block algorithm for dgemv_T .

(b) Block algorithm for dgemv_N

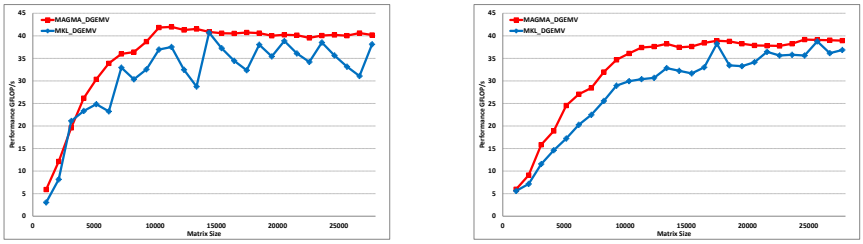
**Fig. 2.** Basic implementation of matrix-vector multiplication on Intel Xeon Phi

Dividing rows in block-rows like this has two advantages: first, every block except the last one will have elements multiple of eight, that is good for vectorization; and second, it helps keeping the blocks aligned with the cache sizes which is essential to reduce memory access time. When the matrix $A$ is not aligned for the cache size, we can increase the size of the first block in order to handle the unaligned portion, while making all the remaining blocks aligned. Compiler guided prefetching for this case is not enough to reach the same performance as for the transpose case. Prefetching to L1 and L2 cache plays important role here.

Similarly to the transpose case, using one or four threads per core provides the same performance. Again, we developed parametrized implementation where one parameter is the number of threads per core. Figure 2b for example illustrates a distribution using one thread per core (displayed as version-1) and four threads per core (displayed as version-2). The thread processes four column together to reduce the write traffic for vector $y$. Before processing the eight elements (eight is the length of SIMD instruction for double precision) it prefetches next eight elements of $A$ from the same column to the L1 cache level and the next four columns of the same block-row to the L2 cache. In this way, when the code proceeds to process the next four columns, the data for them will be obtained from the L2 cache. Processing more than four columns does not improve the performance. For version-2 each thread handles four columns together and then the consecutive eight rows from the same column. Like version-1 each thread will prefetch its portion from the same columns to the L1 cache and from the next four columns to the L2 cache.

The blocking and prefetching technique for the transpose and non-transpose cases are described in Figures 2a and 2b, respectively.

Figures 3a and 3b show the performance comparison of magma_dgemv *vs.* mkl_dgemv. In both the transpose and non-transpose cases the techniques presented yield better performances than the MKL's dgemvs.




(a) Performance of magma_dgemv_T (b) Performance of magma_dgemv_N

**Fig. 3.** Performance of MAGMA MIC dgemv *vs. MKL* on Intel Xeon Phi

## 6 Our proposed symmetric matrix-vector multiplication routine symv

The symv multiplication routine performs:

$$y := \alpha Ax + \beta y, \tag{4}$$

where $\alpha$ and $\beta$ are scalars, x and y are vectors of size N, and A is an N by N symmetric matrix.

The performance of the MKL symv routine is as high as the performance of the gemv routine, and therefore can be further accelerated. Due to the fact that the symv routine accesses half of the matrix, meaning it needs only half of the data transfers, its performance (theoretically) should be twice the one of the gemv. The idea behind getting this acceleration is to reuse the data from half of the matrix to perform the entire multiplication. The traditional way to attain this objective is to divide the whole matrix into small blocks so that each block fits in the cache. For the diagonal blocks a symv kernel is used and for each of the non-diagonal blocks two call to the gemv routine, one for the transpose and one for the non-transpose case, are used. For high parallelism without need for synchronization, each core handles a block and its resulting vector is written independently in separate locations. Thus, a summation is taken at the end to get final $y$ result. As each block is brought to the cache once, this technique is expected to reach close to the theoretical bound which, as mentioned, is twice the performance of gemv.

We performed a set of experiments for different block sizes. In our timing, we ignored the overhead of the summation and the launching of the threads. We illustrate in Figure 5a the performance obtained for different block sizes, chosen so that the data of every block is memory aligned. The maximum performance achieved is around 54 Gflop/s for large matrix sizes and near 50 Gflop/s for smaller matrix sizes. When including the time for the summation, the later results decrease by about $5 - 10\%$. This symv implementation brings an excellent

improvement over the contemporary MKL (e.g., it is about 1.3 times faster). However, the performance is not optimal. This motivated us to search for other MIC-specific optimization techniques, leading to our second algorithm and implementation that adds one more level of blocking. In particular, we manage to reuse data from the L1 cache, which brings the performance up to the optimal level, i.e., twice the one for gemv.

In order to achieve the desired performance one must optimize both at the blocking and at the kernel levels. As there are sixty cores in a MIC we divided the whole matrix into $60 \times 60$ blocks. If (i, j) is the index of a block in a two dimensional grid and block_M×block_N is the block's dimension, block_M and block_N are computed as follows:

$$\begin{aligned} n8\_strip &= (N+7)/8 \\ block\_M &= [\frac{n8\_strip}{60} + (i < (n8\_strip\%60) \ ? \ 1 : 0)] * 8 \\ block\_N &= [\frac{n8\_strip}{60} + (j < (n8\_strip\%60) \ ? \ 1 : 0)] * 8. \end{aligned} \tag{5}$$

When the size of the matrix $A$ is multiple of 8, then both block_M and block_N are multiple of eight as well and all the blocks in the grid are aligned with the cache. When the size of $A$ is not multiple of 8, the non-aligned portion is added to block$(0, 0)$, making all the remaining blocks aligned and of sizes multiple of 8.

The symv computation is organized according to the description presented in Figure 4. Since the diagonal blocks require special attention because their lower or upper portion are accessed, and in order to enforce workload balance among the cores, we split the diagonal blocks over all the cores in a way that provides load balance. The non-diagonal blocks are as well distributed among the cores as described in Figure 4 in order to achieve load balance. The number inside each block indicates which core owns and processes that block. Since the gemv and the symv are memory bound, we found that one thread per core is the best configuration.

Each core computes the symmetric matrix-vector multiplication of its block by performing the gemv_N and gemv_T together, meaning it loads a column of $A$ and computes the multiplication for both the non-transpose and transpose cases, and then moves to the next column. We used the same prefetching technique as the one used in our gemv kernel for the non-transpose case. We prefetch the data of a block to the L2 cache and then every column is prefetched to the L1 cache where we perform all computations involving that data. This technique is illustrated in Figure 4. The corresponding portions of $x$ and $y$ of the matrix-vector multiplication of the red block in Figure 5b is shown in yellow for the non-transpose operation and in the purple color for the transpose operation. Finally, the resulting vector $y_i$ must be summed, and this is done in parallel by the 60 cores. Figure 5b shows the performance of our MAGMA MIC dsymv along with a comparison to the performance of the dsymv routine from the MKL Library. Using the above technique we can achieve almost twice the performance of gemv, which mean that the bound limit for this routine is reached.

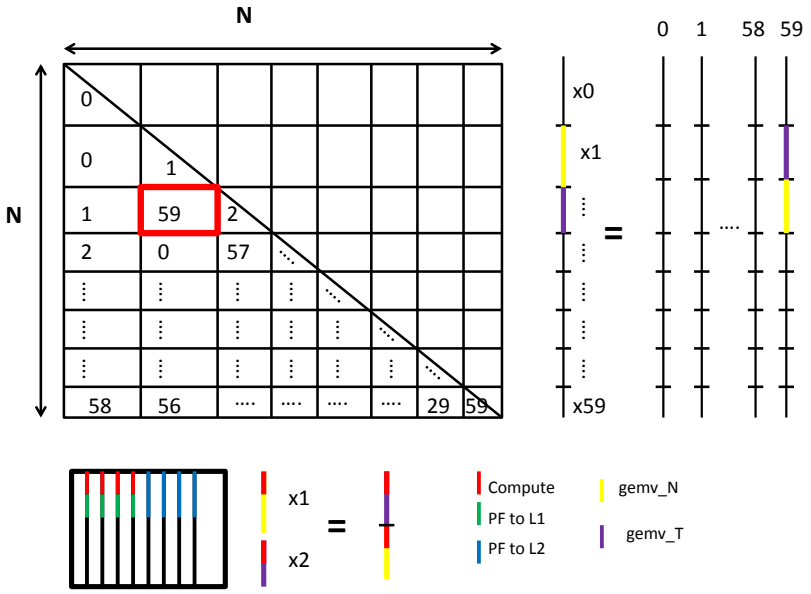
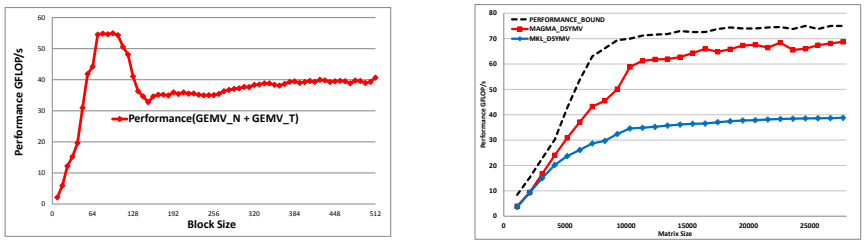



**Fig. 4.** Basic implementation of MAGMA symv on Intel Xeon Phi

## 7 Impact on Eigenvalue and Singular value Algorithms

Eigenvalue and singular value decomposition (SVD) problems are fundamental for many engineering and physics applications. The solution of these problems follow three phases. First phase is reducing the matrix to a condensed form matrix (e.g., tridiagonal form for symmetric eigenvalue problem, and bidiagonal form for singular value decomposition) that has the same eigen/singular-values as the original one. The reductions are referred to as two-sided factorizations, as they are achieved by two-sided orthogonal transformations. Then, in the second phase an eigenvalue (respectively, singular value) solver further computes the eigenpairs (respectively, singular values and vectors) of the condensed form matrix. Finally, in the third phase the eigenvectors (respectively, left and right singular vectors) resulting from the previous phase are multiplied by the orthogonal matrices used in the reduction phase. We performed a set of experiment in order to determine the percentage of time spent in each of these phases for the symmetric eigenvalue problem and the singular value decomposition problem. The results depicted in Figures 6a, and 6b show that the first phase is the most time consuming portion. It consists of more than 80% or 90% of the total time when all eigenvectors/singular vectors or only eigenvalues/singular values are computed, respectively. These observations illustrate the need to improve

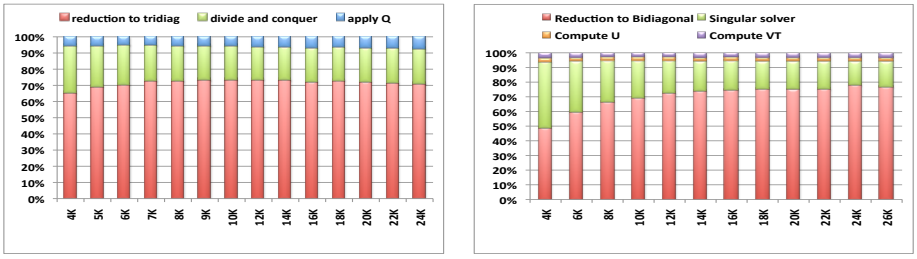



(a) Effect of the blocking size for the implementation of the symv routine as two gemv calls.

(b) Performance of the second approach of the symv routine.

**Fig. 5.** Performance of MAGMA dsymv on Intel Xeon Phi

the reduction phase. It is challenging to accelerate the two-sided factorizations on new architectures because they are rich in Level 2 BLAS operations, which are bandwidth limited and therefore do not scale on recent architectures. For that, we focus in this section on the reduction phase and study its limitation. Furthermore, we present the impact of our optimized kernel on accelerating it on Intel Xeon-Phi coprocessor architectures.




(a) Execution time profile of the symmetric eigenvalue routine dsyevd.

(b) Execution time profile of the SVD routine dgesvd.

**Fig. 6.** Percentage of the time spent in each of the three phase of the symmetric eigenvalue and singular-value problem .

### 7.1 Performance Bound Analysis

In order to evaluate the performance behavior of the two-sided factorizations and to analyze if there are opportunities for improvements, we conduct a computational analysis of the reduction to condensed forms for the two-sided reductions

(TRD and BRD). The total cost for the reduction phase can be summarized as follow:

| **For Tridiagonal:** | **For Bidiagonal:** |
|---|---|
| $\approx \frac{2}{3}n^3_{\mathsf{symv}} + \frac{2}{3}n^3_{\mathsf{Level\ 3}}$ | $\approx \frac{4}{3}n^3_{\mathsf{gemv}} + \frac{4}{3}n^3_{\mathsf{Level\ 3}}$ |
| $\approx \frac{4}{3}n^3.$ | $\approx \frac{8}{3}n^3.$ |

According to these equations we derive below the maximum performance $P_{max}$ that can be reached by any of these reduction algorithms. In particular, for large matrix sizes $n$, $P_{max} = \frac{number\ of\ operations}{minimum\ time\ t_{min}}$, and thus $P_{max}$ is expressed as:

**For Tridiagonal:**

$$\frac{\frac{4}{3}n^3}{\frac{2}{3}n^3 * \frac{1}{P_{symv}} + \frac{2}{3}n^3 * \frac{1}{P_{Level3}}}$$

$$\frac{2 * P_{Level3} * P_{symv}}{P_{Level3} + P_{symv}} \qquad (6)$$

$$\approx 2P_{symv}$$
$$when \quad P_{Level3} \gg P_{symv}.$$

**For Bidiagonal:**

$$\frac{\frac{8}{3}n^3}{\frac{4}{3}n^3 * \frac{1}{P_{gemv}} + \frac{4}{3}n^3 * \frac{1}{P_{Level3}}}$$

$$\frac{2 * P_{Level3} * P_{gemv}}{P_{Level3} + P_{gemv}} \qquad (7)$$

$$\approx 2P_{gemv}$$
$$when \quad P_{Level3} \gg P_{gemv}.$$

The performance of the Level 2 BLAS routines such as the matrix-vector multiplication (symv or gemv) is memory bound and very low compared to the Level 3 BLAS routines which can achieve close to the machine's peak performance. For example, on the Intel Xeon Phi system the performance of dgemv is about 40 Gflop/s, while for dgemm is about 1000 Gflop/s. Thus, one can expect from Equations (6,7) that the performance of the reduction algorithms are bound by the performance of the Level 2 BLAS operations. This proves that the performance behavior for these algorithms is dictated by the matrix-vector Level 2 BLAS routines, and this is one example of why it is very important to optimize them.

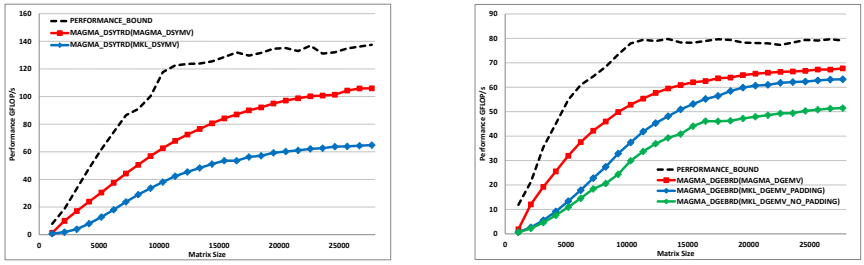
### 7.2 Impact on the Tridiagonal reduction

Figure 7a shows the performance for the tridiagonal reduction using the Xeon Phi. The MAGMA implementation using the MKL symv routine is much slower than when using our proposed symv implementation. In particular MAGMA with the new symv optimization is about 1.6× faster than MAGMA using the MKL symv, and reaches 78% of the theoretical performance bound derived from equation 6.

### 7.3 Impact on the Bidiagonal reduction

Figure 7b shows the performance for the bidiagonal reduction on the Xeon Phi. Similarly to the tridiagonal factorization, the MAGMA bidiagonal reduction using our proposed gemv shows better performance than when using the gemv

routine from the MKL library combined with our proposed fix described in Section 5.1. In particular we are reaching 85% of the theoretical performance bound that we derived in equation 7.




(a) Performance of MAGMA Tridiagonal Reduction Routine dsytrd.

(b) Performance of MAGMA Bidiagonal Reduction Routine dgebrd.

**Fig. 7.** Impact of the proposed symv and gemv routine on the reduction algorithms for eigenvalue and singular value problem.

## 8 Conclusions

We developed MIC-specific optimization techniques that demonstrate how to use the many (currently 60) cores of the MIC to get optimal performance. The techniques and kernels developed are fundamental and can be used elsewhere. For example, we showed that hierarchical communications to get top speed now are needed not only for Level 3 BLAS but also for Level 2 BLAS – indeed, only after we developed and applied multilevel cache blocking, our implementations reached optimal performance. Further, the new gemv kernel handles unaligned general matrices efficiently and its use in higher-level routines, like the bidiagonal reduction, does not require additional optimizations techniques, like padding for example. The impact of our optimizations are clearly visible in performance of the bidiagonal reduction. Finally, our new symv is almost 2× faster than MKL's symv. Optimization in symv makes the tridiagonal reduction 1.6× faster than using MKL's symv.

## Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, the Department of Energy, and Intel. The results were obtained in part with the financial support of the Russian Scientific Fund, Agreement N14-11-00190.

## References


1. CUDA CUBLAS Library. `https://developer.nvidia.com/cublas`.
2. MAGMA. `http://icl.cs.utk.edu/magma/`.
3. Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.
4. E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999.
5. M.J. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, 2012.
6. George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Hérault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2), 2012.
7. Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.
8. Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multicore architectures. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 116–125, New York, NY, USA, 2007. ACM.
9. T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, and J. Dongarra. LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In *Proceedings of 16th IEEE International Conference on High Performance and Communications (HPCC 2014)*, August 2014.
10. T. Dong, A. Haidar, S. Tomov, and J. Dongarra. A fast batched Cholesky factorization on a GPU. In *Proceedings of 2014 International Conference on Parallel Processing (ICPP-2014)*, Septembe 2014.
11. Tingxing Dong, Veselin Dobrev, Tzanio Kolev, Robert Rieben, Stanimire Tomov, and Jack Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.
12. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):18–28, March 1990.
13. Jack Dongarra, Mark Gates, Azzam Haidar, Yulu Jia, Khairul Kabir, Piotr Luszczek, and Stanimire Tomov. Portable hpc programming on intel many-integrated-core hardware with magma port to xeon phi. In *PPAM 2013*, Warsaw, Poland, 09/2013 2013.
14. Jack J. Dongarra, Danny C. Sorensen, and Sven J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215 – 227, 1989. Special Issue on Parallel Algorithms for Numerical Linear Algebra.

15. Samuel H. Fuller and Editors; Committee on Sustaining Growth in Computing Performance; National Research Council Lynette I. Millett. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011.
16. Azzam Haidar, Chongxiao Cao, Asim Yarkhan, Piotr Luszczek, Stanimire Tomov, Khairul Kabir, and Jack Dongarra. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, Washington, DC, USA, 2014. IEEE Computer Society.
17. Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *International Journal of High Performance Computing Applications*, doi:10.1177/1094342014567546, 02/2015.
18. Azzam Haidar, Jack Dongarra, Khairul Kabir, Mark Gates, Piotr Luszczek, Stanimire Tomov, and Yulu Jia. Hpc programming on intel many-integrated-core hardware with magma port to xeon phi. *Scientific Programming*, 23, 01/2015 2015.
19. Azzam Haidar, Piotr Luszczek, Stanimire Tomov, and Jack Dongarra. Towards batched linear solvers on accelerated hardware platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, 02/2015 2015. ACM, ACM.
20. Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, February 2004.
21. Intel. Math kernel library. `https://software.intel.com/en-us/en-us/intel-mkl/`.
22. John McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. (`http://www.cs.virginia.edu/stream/`).
23. O.E.B. Messer, J.A. Harris, S. Parete-Koon, and M.A. Chertkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.
24. J.M. Molero, E.M. Garzón, I. García, E.S. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS.
25. S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010.
26. S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proceedings of the 2010 IEEE International Parallel & Distributed Processing Symposium, IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. `http://dx.doi.org/10.1109/IPDPSW.2010.5470941`DOI: 10.1109/IPDPSW.2010.5470941.
27. Stanimire Tomov, Rajib Nath, and Jack Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Comput.*, 36(12):645–654, December 2010.
28. V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing 08*. IEEE, 2008.
29. Sencer N. Yeralan, Tim A. Davis, and Sanjay Ranka. Sparse mulitfrontal QR on the GPU. Technical report, University of Florida Technical Report, 2013.