Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi*

Erik Saule[1], Kamer Kaya[1], Ümit V. Çatalyürek[1,2]

[1] The Ohio State University, Dept. Biomedical Informatics
[2] The Ohio State University, Dept. Electrical and Computer Engineering
{esaule,kamer,umit}@bmi.osu.edu

Abstract. Intel Xeon Phi is a recently released high-performance coprocessor which features 61 cores each supporting 4 hardware threads with 512-bit wide SIMD registers achieving a peak theoretical performance of 1Tflop/s in double precision. Its design differs from classical modern processors; it comes with a large number of cores, the 4-way hyperthreading capability allows many applications to saturate the massive memory bandwidth, and its large SIMD capabilities allow to reach high computation throughput. The core of many scientific applications involves the multiplication of a large, sparse matrix with a single or multiple dense vectors which are not compute-bound but memory-bound. In this paper, we investigate the performance of the Xeon Phi coprocessor for these sparse linear algebra kernels. We highlight the important hardware details and show that Xeon Phi's sparse kernel performance is very promising and even better than that of cutting-edge CPUs and GPUs.

1 Introduction

Given a large, sparse, $m \times n$ matrix $\mathbf{A}$, an input vector $\mathbf{x}$, and a cutting edge shared-memory manycore architecture Intel Xeon® Phi, we are interested in analyzing the performance of computing $\mathbf{y} \leftarrow \mathbf{Ax}$ in parallel. The computation, known as the sparse-matrix vector multiplication (SpMV), and with some variants, such as the sparse-matrix matrix multiplication (SpMM), they form the computational core of many applications involving linear systems, eigenvalues, and linear programs, i.e., most large scale scientific applications. For this reason, they have been extremely intriguing in the context of high performance computing (HPC). Efficient shared-memory parallelization of these kernels is well studied [1,2,3,9,19], and there exist several techniques such as prefetching, loop transformations, vectorization, register, cache, TLB blocking, and NUMA optimization, which have been extensively investigated to optimize the performance [7,11,12,19]. In addition, company-built support is available for many

* This work was partially supported by the NSF grants CNS-0643969, OCI-0904809 and OCI-0904802. We would like to thank NVIDIA for the K20 cards, Intel for the Xeon Phi prototype, and the Ohio Supercomputing Center for access to Intel hardware.

shared-memory architectures, such as Intel's MKL and NVIDIA's cuSPARSE. Popular 3rd party libraries such as OSKI [18] and pOSKI [8] also exist.

Intel Xeon Phi is a new coprocessor with many cores, hardware threading capabilities, and wide vector registers. Although Intel Xeon Phi has been released recently, performance evaluations already exist in literature [6,15,17]. Eisenlohr et al. investigated the behavior of dense linear algebra factorization on Xeon Phi [6] and Stock et al. proposed an automatic code optimization approach for tensor contraction kernels [17]. For sparse, irregular data, we evaluated the scalability of graph algorithms, coloring and breadth first search (BFS) [15].

Although similar to BFS, SpMV and SpMM are different kernels, in terms of synchronization, memory access, and load balancing. The irregularity and sparsity of SpMV-like kernels create several problems for accelerators. In this paper, we analyze how Xeon Phi performs on SpMV and SpMM. [4] studied the performance of a Conjugate Gradient application which uses SpMV, however this study concerns only a single matrix and is application oriented.

Having 61 cores and hyperthreading capability can help the Intel Xeon Phi to saturate the memory bandwidth during SpMV, which is not the case for many cutting edge processors. Yet, our analysis showed that the memory latency, not the memory bandwidth, is the bottleneck and the reason for not reaching to the peak performance. We observed that the performance of the SpMV kernel highly depends on the nonzero pattern of the matrix and its sparsity: when the nonzeros in a row are aligned and packed in cachelines in the memory, the memory accesses are much faster. We investigate two existing approaches for densifying the computation (namely the reverse Cuthill-McKee ordering *RCM* [5] and dense register blocking). This paper presents concise results and a more detailed version of our work can is available as a technical report [16].

Section 2 presents a brief architectural overview of the Intel Xeon Phi coprocessor. Section 3 describes the sparse-matrix multiplication kernels. In Sections 4 and 5, we conduct analyze Xeon Phi's performance on these kernels using 22 matrices from UFL Sparse Matrix Collection[3]. Section 6 shows that Xeon Phi's sparse matrix performance is better than that of four modern architectures: two dual Intel Xeon processors, X5680 (Westmere) E5-2670 (Sandy Bridge), and two NVIDIA Tesla® GPUs C2050 and K20. Section 7 concludes the paper.

2 The Intel Xeon Phi Coprocessor

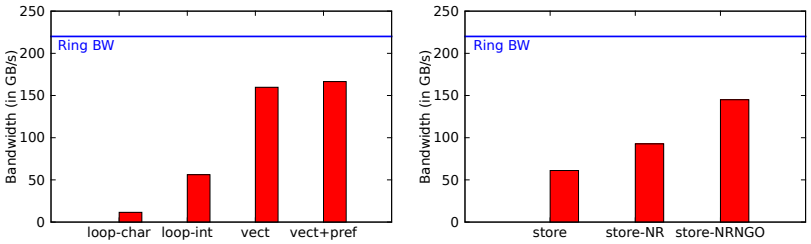
In this work, we use a pre-release KNC card SE10P. There are 61 cores clocked at 1.05GHz. Each core in the architecture has a 32kB L1 data cache, a 32kB L1 instruction cache, and a 512kB L2 cache. The architecture of a core is based on the Pentium architecture: though its design has been updated to 64-bit. A core can hold 4 hardware contexts at any time. A core never executes two instructions from the same hardware context consecutively: in other words, if a program only uses one thread, half of the clock cycles are wasted.

[3] http://www.cise.ufl.edu/research/sparse/matrices/

Most of the performance of the architecture comes from the vector processing unit (VPU). Each core has 32×512-bit SIMD registers which can be used as a vector of 8×64-bit or 16×32-bit values. The VPU can perform many basic instructions, such as addition or division, and mathematical operations, such as sine and sqrt, allowing to reach 8 double precision operations per cycle (16 single precision). The VPU can also perform both an addition and a multiplication simultaneously using a *Fused Multiply-Add* (FMA) instruction. Therefore, the peak performance of the SE10P card is 1.0248 Tflop/s in double precision (2.0496 Tflop/s in single precision) and half without FMA.

The card has 8 memory controllers; each can execute 5.5 billion transactions/second and has two 32-bit channels. Hence, the controllers can achieve an aggregated total bandwidth of 352GB/s. The cores' memory interface are 32-bit wide with two channels and the total bandwidth is 8.4GB/s per core. Thus, the cores can consume 512.4GB/s at most. However, the bandwidth between the cores and the memory controllers is limited by the ring network which can theoretically transport at most 220GB/s.

To better understand the performance of Intel Xeon Phi, we designed two simple benchmarks on read and write bandwidth. In both cases, each thread reads or writes large arrays into the memory multiple times. The read-bandwidth benchmark shows four configurations. The first two read the array one byte at a time or four bytes at a time; they are instruction bound and reach respectively 12GB/s and 60GB/s. The third benchmark (vect) uses SIMD instructions to process 64 bytes at a time; it reaches 171GB/s. The last one (vect+pref) adds prefetching instructions and obtains 177GB/s. Results are presented in Figure 1(a).



(a) Read Bandwidth (b) Write Bandwidth

Fig. 1. Benchmarking read and write bandwidth with various instructions. The ring bus theoretical maximal bandwidth is shown.

Figure 1(b) shows a similar benchmark for write operations. All three tested configurations use vectorized write instructions to overcome the instruction bound. The first benchmark uses a simple store operation and reaches 65GB/s. The second configuration disables the Read For Ownership protocol (which forces the processor to bring a cacheline into the cache before being able to write it) by using a No-Read hint (NR) and improves the write bandwidth to 99GB/s. The last configuration allows the write operations to be committed to the memory in

an arbitrary order using the Non Globaly Ordered write instructions (NRNGO); it yields 155GB/s.

More detailed experiments can be found in our report [16] which shows that when using vect+pref and store-NRNGO, the bandwidth scales sublinearly with the number of cores, indicating a contention on the memory subsystem.

3 Sparse Multiplication Kernels

SpMV is in the form $\mathbf{y} \leftarrow \mathbf{Ax}$ where $\mathbf{A}$ is an $m \times n$ sparse matrix, and $\mathbf{x}$ and $\mathbf{y}$ are $n \times 1$ and $m \times 1$ column vectors. In this kernel, each nonzero is accessed, multiplied with an $\mathbf{x}$-vector entry, and the result is added to a $\mathbf{y}$-vector entry once. That is, there are two reads and one read-and-write per each nonzero accessed. Different from SpMV, in SpMM, $\mathbf{x}$ and $\mathbf{y}$ are $n \times k$ and $m \times k$ dense matrices. Hence, there are k reads and k read-and-writes per each nonzero accessed. Obtaining a good performance for SpMV is difficult on almost any architecture due to the sparsity pattern of $\mathbf{A}$ which yields a non-regular access to the memory. The amount of computation per nonzero is also very small. And most of the operations suffer from bandwidth limitation.

An $m \times n$ sparse matrix $\mathbf{A}$ with τ nonzeros is usually stored in the compressed row storage format CRS which uses three arrays:

- *cids*[.] is an integer array of size τ that stores the column ids for each nonzero in row-major ordering.
- *rptrs*[.] is an integer array of size $m+1$. For $0 \leq i < m$, *rptrs*[i] is the location of the first nonzero of the ith row in the *cids* array. The first element is *rptrs*[0] $= 0$, and the last element is *rptrs[m]* $= \tau$. Hence, all the column indices of row i are stored between *cids*[*rptrs*[i]] and *cids*[*rptrs*[$i+1$]] $- 1$.
- *val*[.] is an array of size τ. *val*[i] is the value of the ith nonzero.

There exist other sparse matrix representations [14], and the best storage format almost always depends on the pattern of the matrix and the kernel. In this work, we use CRS as it constitutes a solid baseline. Since $\mathbf{A}$ is represented in CRS, it is straightforward to assign a row to a single thread in a parallel execution. Each entry $\mathbf{y}_i$ of the output vector can be computed independently while streaming the matrix row by row. While processing a row i, multiple $\mathbf{x}$ values are read, and the sum of the multiplications is written to $\mathbf{y}_i$. Hence, there are one multiply and one add operation per nonzero, and the total number of floating point operations is 2τ.

4 SpMV on Intel Xeon Phi

For the experiments, we use a set of 22 matrices given in Table 1. The matrices are taken from the UFL Sparse Matrix Collection with one exception *mesh_2048* which corresponds to a 5-point stencil 2048×2048 mesh in 2D. We used the CRS representation, store all the scalar values in double precision, and all the indices via 32-bit integers. In the rest of the paper, the matrices are ordered from 1 to 22 by increasing number of nonzero entries. We repeated each operation 70 times and compute the averages of the last 60 operations. Caches are flushed between each measurement.

Table 1. Properties of the matrices used in the experiments. All matrices are square.

#	name	#row	#nonzero
1	*shallow_water1*	81,920	204,800
2	*2cubes_sphere*	101,492	874,378
3	*scircuit*	170,998	958,936
4	*mac_econ*	206,500	1,273,389
5	*cop20k_A*	121,192	1,362,087
6	*cant*	62,451	2,034,917
7	*pdb1HYS*	36,417	2,190,591
8	*webbase-1M*	1,000,005	3,105,536
9	*hood*	220,542	5,057,982
10	*bmw3_2*	227,362	5,757,996
11	*pre2*	659,033	5,834,044
12	*pwtk*	217,918	5,871,175
13	*crankseg_2*	63,838	7,106,348
14	*torso1*	116,158	8,516,500
15	*atmosmodd*	1,270,432	8,814,880
16	*msdoor*	415,863	9,794,513
17	*F1*	343,791	13,590,452
18	*nd24k*	72,000	14,393,817
19	*inline_1*	503,712	18,659,941
20	*mesh_2048*	4,194,304	20,963,328
21	*ldoor*	952,203	21,723,010
22	*cage14*	1,505,785	27,130,349

4.1 Performance evaluation

The SpMV kernel is implemented in C++ using OpenMP and processes the rows in parallel. We tested our dataset with multiple scheduling policies when compiled with `-O1` and `-O3` (see [16] for more details). When compiled with `-O1`, the performance obtained varies from 1 to 13GFlop/s. When compiled with `-O3` the performance rises for all matrices and reaches 22GFlop/s on *nd24k*. In total, 5 matrices from our set achieve a performance over 15GFlop/s.

An interesting observation is that the difference on the performance is not constant; it depends on the matrix and not correlated to its size. Analyzing the compiled assembly code for the SpMV inner loop (which computes the dot product between a sparse matrix row and the dense input vector) gives an insight on why the performances differ. When `-O1` is used, the dot product is implemented in a simple way, one element at a time, with 3 memory indirections, one increment, one addition, one multiplication, one test, and one jump per nonzero.

The code generated in `-O3` is much more complex. It uses vectorial operations so as to make 8 operations at once. The compiled code loads 8 consecutive values of the sparse row in a 512-bit SIMD register in a single operation. Then it populates another SIMD register with the values of the input vector. Once populated, the two vectors are multiplied and accumulated with previous results in a single FMA. Populating the SIMD register with the appropriate values from $\mathbf{x}$ is non trivial since these values are not consecutive in memory. However, Xeon Phi offers an instruction, *vgatherd*, that allows to fetch multiple values at once. The instruction takes an offset vector, a pointer to the beginning of the array, and a destination register. In general, *vgatherd* needs to be called as many times as the number of cachelines the offset vector touches (indicated by a auxiliary bit-mask), since it can only simultaneously fetch the elements that are on the same cacheline. So overall, one FMA, two vector loads (one for the nonzero from the matrix and one for the column positions), one increment, one test, and some *vgatherd* are performed for each 8 nonzeros of the matrix.

Figure 2 shows the SpMV performance for each matrix with `-O1` and `-O3` as a function of the *useful cacheline density* (UCLD), a metric we devised for the analysis. For each row, we computed the ratio of the number of nonzeros on that row to the number of elements in the cachelines of the input vector due to that row. Then we took the average of these values to compute UCLD. For each matrix, there are two points in Figure 2, one for the performance in `-O1` (marked

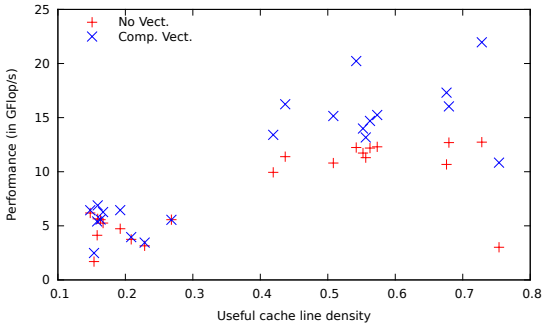


Fig. 2. The improvement of `-O3` (Comp. Vect.) is linked to cacheline density.

with '+'s) and one for the performance in `-O3` (marked with '×'s). These points are horizontally aligned for the same matrix, and their vertical distance represents the improvement on the performance for that matrix. The improvement with vectorization, and in particular with *vgatherd*, is significantly much higher when the UCLD is high.

4.2 Bandwidth considerations

The nonzeros in the matrix need to be transferred to the core before being processed. Assuming the access to the vectors do not incur any memory transfer, and since each nonzero takes 12 bytes (8 for the value and 4 for the column index) and incurs two floating point operations (multiplication and sum), the flop-to-byte ratio of SpMV is $\frac{2}{12} = \frac{1}{6}$. We saw that the sustained memory bandwidth is about 180GB/s, which indicates a maximum performance for the SpMV kernel of 30GFlop/s. This is not obtained by our previous experiments.

Assuming only 12 bytes per nonzero need to be transfered to the core gives only a *naive bandwidth* for SpMV: both vectors and the row indices also need to be transferred. For an $n \times n$ matrix with τ nonzeros, the actual minimum amount of memory that need to be transferred in both ways is $2 \times n \times 8 + (n + 1) \times 4 + \tau \times (8 + 4) = 4 + 20 \times n + 12 \times \tau$. Usually, 12τ dominates the equation, but for sparser matrices, $20n$ should not be ignored. The *application bandwidth*, which takes both terms into account, is a common alternative cross-architecture measure of performance on SpMV.

Figure 3 (left) shows that the naive approach which ignores a significant portion of the data for some matrices. The application bandwidth obtained ranges from 22GB/s to 132GB/s. Most matrices have a bandwidth below 100GB/s.

The application bandwidth is computed assuming that every single byte of the problem is transferred exactly once. This assumption is (mostly) true for the matrix and the output vector. However, it does not hold for the input vector for two reasons: first, it is unlikely that each vector element will be used by only a single core's threads, some element will be transferred to multiple cores. Furthermore, a core's cache is only 512kB, and elements of the input vector may be transfered multiple times to the same core. We analytically computed the number of cachelines accessed by each core assuming that chunks of 64 rows are distributed in a round-robin fashion. We performed the analysis assuming an infinite

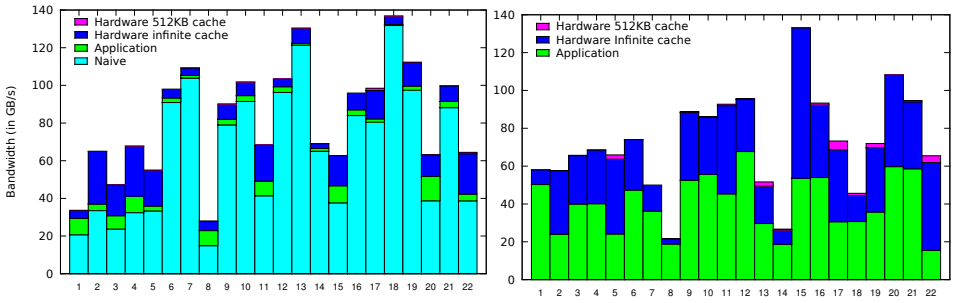


Fig. 3. The achieved bandwidth for SpMV (left) and SpMM (right).

cache and with a 512kB cache. We computed the effective memory bandwidth of SpMV and display them as the top two stacks of the bars in Figure 3 (left). Three observations are striking: first, the difference between the application bandwidth and estimated actual bandwidth is greater than 10GB/s on 10 instances and more than 20GB/s on three of them. The highest difference is seen on *2cubes_sphere* (#2) where the amount of data transferred is 1.7 times larger than the application bandwidth. Second, there is no significant difference between the assumed infinite cache and 512kB cache bandwidth. That is, no cache thrashing occurs. Finally, even when we take the actual memory transfers into account, the obtained bandwidth is still way below the architecture peak bandwidth.

4.3 Effect of matrix ordering

A widely-used approach to improve SpMV performance is ordering the rows and columns to make the matrix more suitable for the kernel. Such permutations are used in sparse linear algebra for multiple purposes such as improving numerical stability and preconditioning. Here, we employ the reverse Cuthill-McKee algorithm (RCM) [5]. RCM has been widely used for minimizing the maximum distance between the nonzeros and the diagonal of the matrix, i.e., the *bandwidth of the matrix*. We expect that such a densification of the nonzeros can improve both the UCLD of the matrix and reduce the number of times the vector needed to be transfered from the main memory to the core caches.

RCM improves the performance of only 4 matrices by more than 2GFlop/s and most of the matrices benefit less. The performance of 8 matrices degrade. Hence, RCM ordering was not able to significantly improve SpMV on Intel Xeon Phi (see [16] for more details).

4.4 Effect of register blocking

One limitation in the original SpMV implementation is that only a single nonzero is processed at a time. Register blocking helps us to process all the nonzeros within a region at once. The region should be small enough that the data associated with it can be stored in the registers so as to minimize memory accesses [7]. Assuming a regular partitioning $\mathbf{A}$ to blocks of size $a \times b$, we use a dense block representation for the blocks containing at least one nonzero. We represent this list of non-empty blocks via CRS. One dimension of the blocks is set to 8 to

leverage the Xeon Phi architecture which naturally align on 512 bits, the other dimension varies from 1 to 8. To perform the multiplication, each dense block is loaded into the registers in packs of 8 values allowing to use Fused Multiply-Add operations. Register blocking typically helps the performances by reducing three quantities: 1) the matrix size in memory, 2) the number of instructions to perform the multiplications, 3) the number of load instructions to the vector.

Overall, we could not observe a constant improvement for register blocking on Xeon Phi. The best scheme with 8×1 blocks improved the performance on only 8 instances compared to the original implementation (detailed results in [16]). Register blocking allows to reach a much higher utilization of the hardware (the effective memory bandwidth is over 160GB/s) but this does not compensate the large increase in matrix sizes. Indeed, the matrices we used have a low locality leading to a sharp increase in the size of the matrix when encoded using dense tiles. There is almost no reduction of the load instruction of the vector since the *vgatherd* instruction already reads the input vector per batch.

5 SpMM on Intel Xeon Phi

One idea to obtain more performance is to increase the flop-to-byte ratio by performing more than one SpMV at a time. Many applications can take the advantage of using multiple vectors at once, e.g., graph based recommendation systems [10] or eigensolvers (by the use of the LOBPCG algorithm) [20]. Multiplying several vectors by the same matrix boils down to multiplying a sparse matrix by a dense matrix, which we refer to as SpMM. All the statements above are also valid for existing cutting-edge processors and accelerators. However, with its large SIMD registers, Xeon Phi is expected to perform significantly better.

To implement $\mathbf{Y} \leftarrow \mathbf{AX}$, we encode the dense $m \times k$ input matrix $\mathbf{X}$ in row-major, so each row is contiguous in memory. To process a row $\mathbf{A}_{i*}$ of $\mathbf{A}$, a temporary array of size k is first initialized to zero. Then for each nonzero in $\mathbf{A}_{i*}$, a row $\mathbf{X}_{j*}$ is streamed to be multiplied by the nonzero and the result is accumulated into the temporary array. We developed three variants of that algorithm: the first variant is generic and relies on compiler vectorization. The second is tuned for values of k which are multiple of 8 and uses FMA to perform the multiplications and additions of 8 at a time. The temporary values are kept in registers by taking the advantage of the large number of SIMD registers available on Xeon Phi. The third variant also uses Non-Globally Ordered write instructions with No-Read hint (NRNGO).

We experimented with $k = 16$. Manual vectorization doubles the performance allowing to reach more than 60GFlop/s in 11 instances. The use of NRNGO write instructions provides significant performance improvements. The achieved performance peaks on the matrix *pwtk* matrix at 128GFlop/s. Figure 3 (right) shows the bandwidth achieved by the best implementation (complete results are in [16]). The application bandwidth is computed assuming the matrix and vector are transferred only once. It surpasses 60GB/s in only 1 instance. Since there are 16 input vectors, the overhead induced by transferring the values in $\mathbf{X}$ to multiple cores is comparable to the application bandwidth. The impact of having a finite cache is mostly negligable.

6 Against other architectures

We compare the performance of Xeon Phi with 4 other architectures including 2 GPU configurations and 2 CPU configurations. We used two CUDA-enabled cards from NVIDIA: an NVIDIA Tesla C2050 (448 CUDA Cores @ 1.15GHz, 2.6GB memory @ 1.5GHz, ECC on, CUDA 4.2) and an NVIDIA Tesla K20 (2,496 CUDA Cores @ 0.71GHz, 4.8GB memory @ 2.6GHz, ECC on, CUDA 5.0). For both GPU configurations, we use the CuSparse library. We also use two Intel CPU systems: the first has a dual Intel Xeon X5680 (Westmere: 6 cores @ 3.33Ghz, no hyperthreading, 12MB shared L3 cache). The second has a dual Intel Xeon E5-2670 (Sandy Bridge: 8 cores @ 2.6GHz, hyperthreading enabled, 20MB shared L3 cache). The codes for both CPU architectures are compiled with the `icc 13.0` with `-O3` optimization flag. The implementation used is the same as the one used on Xeon Phi except the vector optimizations in SpMM where the instruction sets differ.

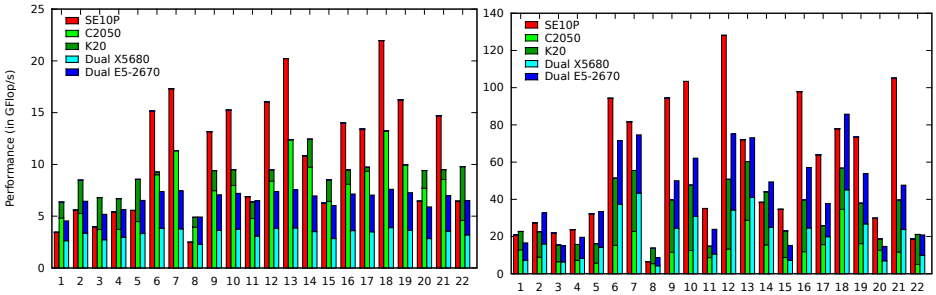


Fig. 4. Architectural comparison between a Intel Xeon Phi coprocessor (Pre-release SE10P), two NVIDIA GPUs (C2050 and K20) and two dual CPU architectures (Intel Xeon X5680 and Intel Xeon E5-2670) for SpMV (left) and SpMM (right).

Results of the experiments are presented in Figure 4. We present the configurations as stacked bar charts: K20 on top of C2050 and E5-2670 on top of X5680. Figure 4 (left) shows the SpMV results: E5-2670 appears to be roughly twice faster than X5680. It reaches a performance between 4.5 and 7.6GFlop/s and achieves the highest performance for one instance. For GPU architectures, K20 is faster than the C2050. It performs better for 18/22 instances. It obtains between 4.9 and 13.2GFlop/s and the highest performance on 9 instances. Xeon Phi reaches the highest performance on 12 instances and it is the only architecture which obtains more than 15GFlop/s. Furthermore, it does it for 7 instances.

Figure 4 (right) shows the SpMM results: E5-2670 gets twice the performance of X5680, which is similar to their relative SpMV performances. The K20 is often more than twice faster than C2050, which is much better compared with their relative performances in SpMV. The Xeon Phi coprocessor gets the best performance in 14 instances. Intel Xeon Phi is the only architecture which achieves more than 100GFlop/s. Furthermore, it reaches more than 60GFlop/s on 9 instances. The CPU configurations reach more than 60GFlop/s on 6 instances while the GPU configurations never achieve that performance.

7 Conclusion and Future Work

In this work, we analyze the performance of Intel Xeon Phi coprocessor on SpMV and SpMM. These sparse algebra kernels have been used in many important applications. The analysis gives the first absolute performance results of Xeon Phi. Overall, the performance we obtained is very promising. When compared with cutting-edge processors and accelerators, its SpMV, and especially SpMM, performance are superior thanks to its wide registers and vectorization capabilities.

In particular, we showed that the sparse matrix kernels we investigated are latency bound. Our experiments suggested that having a relatively small 512kB L2 cache per core is not a problem for Intel Xeon Phi. However, having 61 cores induces a significant data transfer overhead due to accessing similar parts of $\mathbf{x}$ and $\mathbf{X}$ from multiple cores, especially in SpMM. We linked the performance of SpMV to the efficacy of the *vgatherd* instruction which allows efficient memory loads. The classical techniques to improve the performance of SpMV appeared to bring little improvements on Xeon Phi. As a future work, we are planning to investigate matrix storage schemes, intra-core locality, and data partitioning to improve the performance of Xeon Phi.

References

1. N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proc. High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.
2. A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proc. SPAA '09*, pages 233–244, 2009.
3. A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS*, 2011.
4. T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. Openmp programming on intel xeon phi coprocessors: An early performance comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, Nov. 2012.
5. E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. ACM national conference*, pages 157–172, 1969.
6. J. Eisenlor, D. E. Hudak, K. Tomko, and T. C. Prince. Dense linear algebra factorization in OpenMP and Cilk Plus on Intel MIC: Development experiences and performance analysis. In *TACC-Intel Highly Parallel Computing Symp.*, 2012.
7. E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *Proc. of ICCS*, pages 127–136, 2001.
8. A. Jain. pOSKI: An extensible autotuning framework to perform optimized spmvs on multicore architecture. Master's thesis, UC Berkeley, 2008.
9. M. Krotkiewski and M. Dabrowski. Parallel symmetric sparse matrix-vector product on scalar multi-core CPUs. *Parallel Comput.*, 36(4):181–198, Apr. 2010.
10. O. Küçüktunç, K. Kaya, E. Saule, and Ü. V. Çatalyürek. Fast recommendation on bibliographic networks. In *Proc. ASONAM'12*, Aug 2012.
11. J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *Int. J. High Perform. Comput. Appl.*, 18(2), May 2004.
12. R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Appl. Algebra Eng., Commun. Comput.*, 18(3):297–311, May 2007.
13. S. Potluri, K. Tomko, D. Bureddy, and D. K. Panda. Intra-MIC MPI communication using MVAPICH2: Early experience. In *TACC-Intel Highly Parallel Computing Symp.*, 2012.
14. Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.
15. E. Saule and Ü. V. Çatalyürek. An early evaluation of the scalability of graph algorithms on the Intel MIC architecture. In *IPDPS Workshop MTAAP*, 2012.
16. E. Saule, K. Kaya, and U. V. Catalyurek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. Technical Report arXiv:1302.1078, ArXiv, Feb. 2013.
17. K. Stock, L.-N. Pouchet, and P. Sadayappan. Automatic transformations for effective parallel execution on intel many integrated core. In *TACC-Intel Highly Parallel Computing Symp.*, 2012.
18. R. Vuduc, J. Demmel, , and K. Yelic. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005, J. of Physics: Conference Series*, 2005.
19. S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. SC '07*, 2007.
20. Z. Zhou, E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, J. P. Vary, and Ü. V. Çatalyürek. An out-of-core eigensolver on SSD-equipped clusters. In *Proc. of IEEE Cluster*, Sep 2012.