

# A Performance Comparison of Linear Algebra Libraries for Sparse Matrix-Vector Product

**Douglas A. Augusto**    **Luiz M. Carvalho**  
DMA – IM – UFRJ      DMA – IME – UERJ  
daa@labma.ufrj.br    luizmc@ime.uerj.br

**Paulo Goldfeld**    **Albert E. F. Muritiba**    **Michael Souza**  
DMA – IM – UFRJ      DEMA – UFC              DEMA – UFC  
goldfeld@ufrj.br      einstein@ufc.br          michael@ufc.br

**Abstract:** In this work we evaluate the computing performance of some well-known linear algebra libraries when performing the sparse matrix-vector product using different matrix formats. We use as benchmark real-world sparse matrices that came from different domains. The results are carried out and compared on CPU and GPU processors.

**Keywords:** *Sparse Matrix-vector product, Computational Linear Algebra, Linear Algebra libraries*

## 1 Introduction

This document presents a performance evaluation of the parallel product of sparse matrices by dense vectors (SpMv) of three sparse matrix formats: *COOrdinate sparse matrix* (COO), *Compressed Sparse Row* (CSR) and *HYBrid* (HYB).

The SpMv product is a critical hotspot in many high-level algorithms, such as for instance iterative solvers, and thus a lot of effort has been put in order to accelerate the operation, particularly concerning the parallel efficiency [3], which has become increasingly important. With the deep architectural change brought by the recent emergence of massively parallel accelerators, new parallelization and optimization strategies had to be developed to properly exploit those new processors. In this context, the goal of this work is to present an evaluation and analysis of how state-of-the-art linear algebra libraries perform with respect to the sparse matrix-vector product on both CPU and GPU architectures, using different sparse matrix formats. In a nutshell, we want to measure how fast each library-format combination computes the product  $\mathbf{y} = \mathbf{Ax}$ , where  $\mathbf{y}$  and  $\mathbf{x}$  are dense vectors and  $\mathbf{A}$  a sparse matrix.

The investigated linear algebra packages are Intel's *Math Kernel Library (MKL)*<sup>1</sup>, the open source CUDA-based *CUSP*<sup>2</sup> C++ sparse matrix library, NVIDIA's CUDA *cuSPARSE*<sup>3</sup>, and finally the OpenCL-based open source *ViennaCL* library<sup>4</sup>.

This paper is organized as follows. The sparse matrix formats considered in this work are presented in Section 2. Section 3 briefly describes the assessed parallel linear algebra packages. The computational experiments followed by an analysis of the obtained results are presented in Section 4. Finally, Section 5 concludes the study and points out some directions of future work.

<sup>1</sup><http://software.intel.com/en-us/intel-mkl>

<sup>2</sup><http://cusplibrary.github.com>

<sup>3</sup><http://developer.nvidia.com/cusparse>

<sup>4</sup><http://viennacl.sf.net>

## 2 Sparse Matrix Formats

There are many ways to store the non-zeroes of sparse matrices, differing mainly with respect to generality, compactness, and efficiency. Being generic means that the format can be used with roughly the same efficiency upon different matrix structures; in other words, a generic format does not take advantage of a particular structure, for example, diagonal matrices—the formats covered in this work are classified as generic. Compactness regards how compact is the representation of the non-zeroes elements of a sparse matrix, that is, how many storage space is necessary to represent it. Usually, the more compact is the representation, the better it is; however, the storage efficiency may conflict with the efficiency in which the matrix operations are performed. Finally, efficiency has to do with how fast the matrix operations are carried out on a certain processor architecture, and can vary depending on the data structure used to hold the matrix components and how they are organized.

As one can guess, generality, compactness and efficiency are usually conflicting objectives. This explains why there are several sparse matrix formats out there; basically each one tries to fulfill a particular niche defined by an instantiation of the these three aspects.

In this paper we will focus on three frequently used sparse matrix formats, *Coordinate Sparse Matrix*, *Compressed Sparse Row*, and the *Hybrid Sparse Matrix* format.

**COO** The *COOrdinate Sparse Matrix* (COO) format is the most direct representation of a sparse matrix. For each non-zero element it stores its row and column indices and, of course, its value. This leads to a conceptually simple representation but not the most compact one, requiring a total of  $(2 \times I_s + E_s) \times nnz$  bytes, where  $I_s$  and  $E_s$  are, respectively, the space in bytes required to store the indices and value of an element;  $nnz$  is the number of non-zeroes.

**CSR** The *Compressed Sparse Row* format, or simply CSR, is one of the most well-known formats used to store the non-zero elements of a sparse matrix. Its structure has three arrays: one holding the non-zero values, the other one the column index of each value and finally a third one which stores the indices of each row with respect to the array of non-zeroes. Therefore, the CSR storage requirement is  $(I_s + E_s) \times nnz + I_s \times (n + 1)$  bytes, where  $n$  is the number of rows.

The drawback of CSR is that it causes non-coalesced memory access when reading the non-zero elements by adjacent threads, which is suboptimal for architectures that take great advantage of coalesced accesses, such as GPUs. On the other hand, its linear access pattern fits well the cache hierarchy of CPUs, which translates in an efficient use of caches.

**HYB** Finally, the *HYBrid Sparse Matrix* (HYB) format mixes two storage schemes: the regular part of a sparse matrix are represented using the ELLPack storage format (ELL) and the irregular ones—the extra entries of the irregular rows—as COO [1]. Roughly speaking, the ELL format resembles a transposed version of CSR and, for efficiency, it packs each row as if all of them had the same number of non-zeroes—of course, this considerably increases the space requirements when the format is solely applied to sparse matrices that have a large variation with respect to the number of non-zeroes per row.<sup>5</sup> By properly assigning a representation scheme to different parts of the matrix, the HYB format exploits the efficiency of ELL whenever possible and falls back to COO otherwise in order to attain a good trade-off between efficiency and compactness. The necessary space taken by the HYB format depends on the particular matrix and how the implementation defines what is regular or not; therefore, it is  $n \times max_{nnz} \times (I_s + E_s) + (2 \times I_s + E_s) \times extra_{nnz}$  bytes, where  $n$  is the number of rows stored in ELL,  $max_{nnz}$  is the implementation-defined parameter that says how many (non-)zeroes entries per row will be packed in ELL, and  $extra_{nnz}$  is the sum of the extra non-zeroes of each row that could not fit into the array of  $max_{nnz}$  elements.

<sup>5</sup>This is why ELL is not considered a generic format [4].

### 3 Parallel Linear Algebra Libraries

Nowadays there is a significant amount of effort devoted to building high-level libraries for basic linear algebra operations (BLAS level 1, 2 and 3) that explore the power of parallel multi-core processors (CPUs) and/or many-core accelerators (such as GPUs).

The remaining of this section briefly presents some of the possibly most popular packages for linear algebra, which are then evaluated in terms of performance in Section 4. The majority of them support specific architectures (*MKL*, *CUSP* and *cuSPARSE*) whereas one, *ViennaCL*, can seamlessly support several architectures through the portability provided by the OpenCL backend.

**MKL** Intel *Math Kernel Library* is a proprietary package of mathematical routines, including a rich set of linear algebra, that are optimized for Intel processors. Applications that use MKL can be written in both C/C++ and FORTRAN languages. Although MKL does not support GPUs, it can be run on the massively parallel Intel's Many Integrated Core (MIC) architecture, which has a lot more cores and a substantially wider vector unit.

**cuSPARSE** *NVIDIA CUDA Sparse Matrix library* (cuSPARSE) is library designed for C or C++ that provides a collection of linear algebra subroutines, including sparse vector by dense vector, sparse matrix by dense vector, sparse matrix by multiple dense vectors, sparse matrix by sparse matrix addition and multiplication, sparse triangular solve and tri-diagonal solver. It supports dense and several sparse matrix formats (COO, CSR, CSC, ELL, HYB and Blocked CSR), providing also matrix format conversion routines.

**CUSP** CUSP is an open-source high-level C++ template library for sparse matrix computations using CUDA. It supports operations on dense matrices and on many sparse matrix formats (COO, CSR, DIA, ELL and HYB). Among the CUSP's features there are: matrix format conversions, sparse matrix-vector multiplication, matrix transposes, iterative solvers (Conjugate-Gradient, Biconjugate Gradient, Biconjugate Gradient Stabilized, Generalized Minimum Residual, Multi-mass Conjugate-Gradient and Multi-mass Biconjugate Gradient stabilized), pre-conditioners (Algebraic Multigrid based on Smoothed Aggregation, Approximate Inverse and Diagonal) and some utilities.

**ViennaCL** Vienna Computing Language is an open source linear algebra library written in C++, capable of accelerating computations on both many-core and multi-core processors. It is primarily focused on basic linear algebra operations (BLAS level 1, 2 and 3) and the solution for large linear systems by means of iterative methods with optional preconditioners. One of its most prominent features is the adoption of the OpenCL backend, an open-standard programming language for heterogeneous massively parallel computing<sup>6</sup>, by which ViennaCL achieves functional portability across devices and vendors.

With respect to sparse matrix formats, ViennaCL natively supports the following formats: COO, CSR, ELL, and HYB.

## 4 Computational Experiments

### 4.1 Benchmark problems

All measurements were taken using a set of benchmark square matrices as compiled in [5], covering a variety of fields. Table 1 summarizes the characteristics of each sparse matrix considered in this work.

---

<sup>6</sup><http://www.khronos.org>

Problem	Dimension	Non-zeroes ( <i>nnz</i> )	<i>nnz/row</i>	<i>nnz</i> distribution
<i>Protein</i>	36417	4.3mi	119	
<i>FEM/cantilever</i>	62451	4.0mi	65	
<i>FEM/spheres</i>	83334	6.0mi	72	
<i>Economic</i>	206500	1.27mi	6	
<i>Webbase</i>	1000005	3.1mi	3	

Table 1: Summary of the benchmark matrices.

## 4.2 Computational environment

The experiments were carried out on a 64-bit machine featuring two 16-core Intel Xeon E5-2650v2 processor at 2.6GHz (hyperthreading enabled) and an NVIDIA Titan GPU (driver version 319.82) running Debian GNU/Linux (kernel version 3.12). Except for the MKL implementation which was compiled with the Intel `icc` compiler 14.0, the GNU compiler `g++` version 4.8 was used in every other experiment. The optimization flag was always turned on. At least 1000 independent runs were performed for each experiment and the results were averaged. The following versions of the libraries were used: Intel MKL 11.1, CUSP 3.0, NVIDIA cuSPARSE 5.5, and ViennaCL 1.5.1.

The results were obtained using double-precision float-point arithmetic and reported in GFlop/s, measured as

$$GFlop/s = \frac{fp_{operations}}{execution_{time} \times 10^9},$$

where  $fp_{operations}$  is given by  $non_{zeroes} \times 2$ , accounting for products and sums, and  $execution_{time}$  is the wall-clock time in seconds of the actual SpMv operation. The overhead time, i.e. matrix assembly and data transfer times, is not included in the evaluation as the overhead is usually mitigated when running iterated matrix-vector products on the device which do not require data transfer nor reassembling, thus the dominant cost is usually the product itself [1, 3]. Moreover, in this context the transfer time (to and from the device) could be almost completely hidden by carefully overlapping communication and computation.

## 4.3 Results and discussion

Figures 1, 2, 3, 4 and 5 show the performance comparison of the double-precision sparse matrix-vector product for the instances considered in this work (see Table 1). On the left side (a) one can find the results carried out on the CPU and on the right side (b) on the GPU. Since Intel MKL does not support GPU devices, the results are restricted to the CPU.<sup>7</sup> Conversely, as cuSPARSE and CUSP are not compatible with CPUs, only results for GPU were obtained.

One can clearly see that in general the GPU outperforms the CPU by a large margin, which is not surprising. The only tight result took place on the *Economic* matrix (Figure 4), in which the CSR format on the CPU using MKL was able to beat GPU's COO and CSR, and had tied with GPU's HYB using cuSPARSE. This can be explained by the low number of non-zeroes per row (as pointed out in Table 1) which, despite the regular (diagonal) structure of the matrix,

<sup>7</sup>Moreover, Intel MKL also does not provide support for the HYB format.

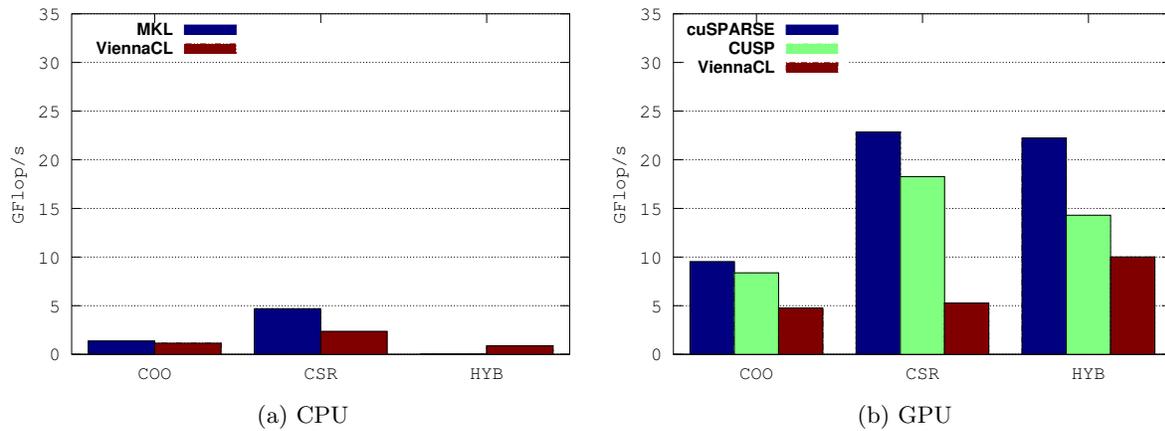


Figure 1: SpMv: *Protein* matrix, double-precision.

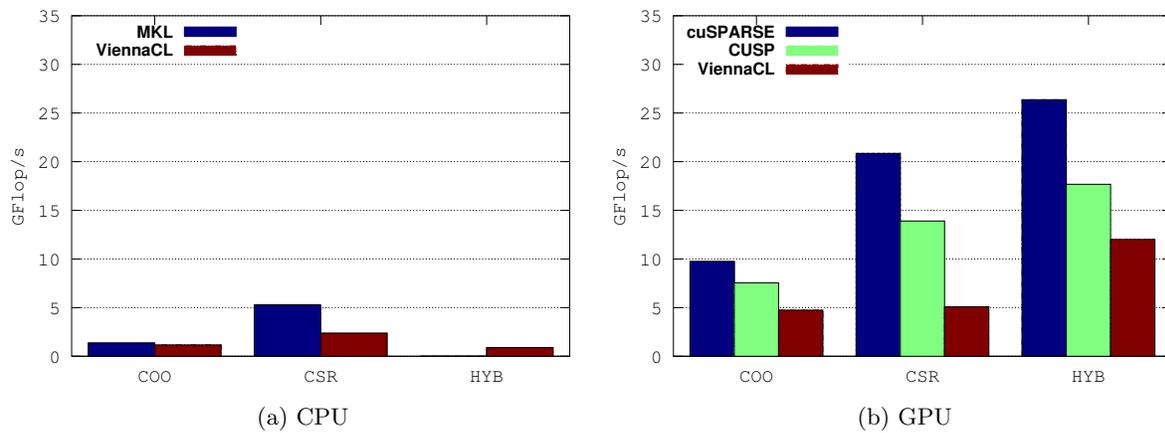


Figure 2: SpMv: *FEM/cantilever* matrix, double-precision.

cannot provide enough work for a GPU thread in order to compensate the scheduling overhead. Another evidence of the relation between low number of non-zeroes and poor GPU performance can be seen in Figure 5. In this case, even achieving a not-so-great performance, the GPU could still outperform the CPU, probably due to the impact of the irregular structure of the matrix on the cache hierarchy.

Concerning the performance variation among the formats on the CPU, one can observe that CSR attained the best results. The main reason of the superiority of CSR is owed to the excess of indexing in COO [2] and to the strided memory access pattern of the ELL storage scheme in HYB, which is suboptimal on architectures such as CPUs, which prefetch in cache consecutive memory regions. Differently, the CSR format results in a cache-friendly linear memory access pattern, because the non-zeroes entries of a row are stored and read consecutively.

It comes as no surprise that the indexing overhead of COO also affects negatively its performance on the GPU, putting the format as the overall worst performing one. Unsurprisingly as well, the most performing format on the GPU is HYB, which relies mostly on ELL. By arranging the non-zeroes elements in a transposed fashion, the ELL format yields a nice coalesced memory access pattern where each adjacent GPU thread will access adjacent memory regions, translating in a very efficient memory access which minimizes memory transactions. Among the instances with high number of non-zeroes per row, that is, *Protein*, *FEM/cantilever* and *FEM/spheres*, it is not coincidence that we have obtained the highest throughputs using HYB exactly on the most regular ones, *FEM/cantilever* and *FEM/spheres*, which better exploit the

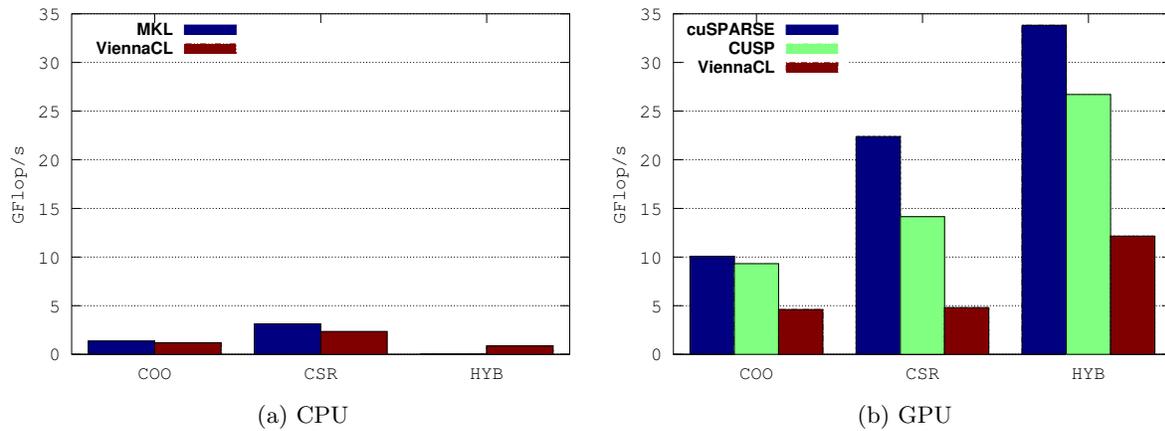


Figure 3: SpMv: *FEM/spheres* matrix, double-precision.

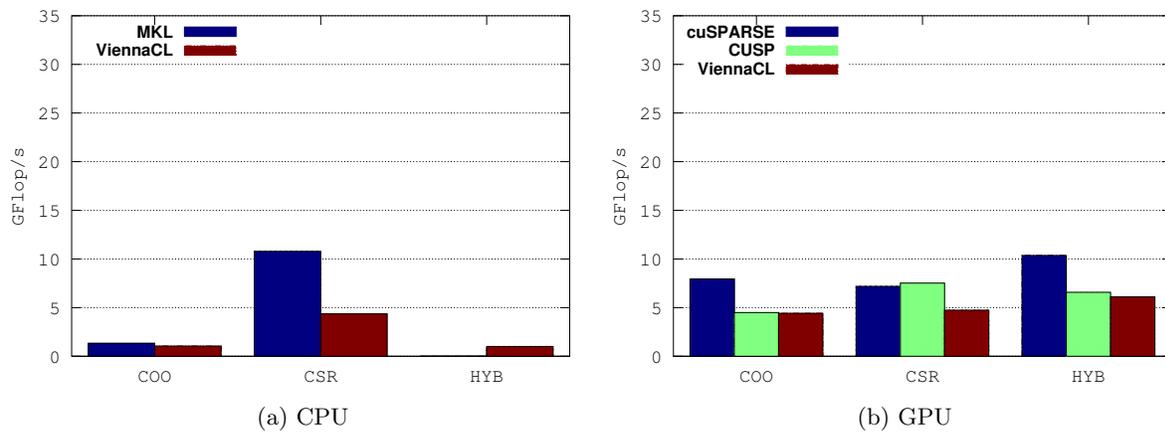


Figure 4: SpMv: *Economic* matrix, double-precision.

ELL format (Figures 2 and 3). As for CSR on the GPU, though ending up ahead of COO in most cases, its sequential memory access pattern optimized for CPU does not, as expected, achieve the best performance on the GPU architecture.

In what regards the libraries' performance, we have two clear winners: the Intel MKL on the CPU side and NVIDIA cuSPARSE on the GPU. In second place comes the CUSP library followed by ViennaCL, both non-proprietary efforts. It is not easy to pinpoint all factors responsible for such differences, but we can speculate some of them. Firstly, both MKL and cuSPARSE target a single architecture and are optimized for devices from the respective vendors, that is, Intel and NVIDIA. This by itself makes the optimization task much easier, but does not tell the whole story, because CUSP supports only a single vendor as well yet wound up behind cuSPARSE. Another factor that may have played a major role is the maturity of the implementation or, in other words, how many effort has been put into each library so far. In this sense, on top of their longevity as projects, Intel MKL and NVIDIA cuSPARSE arguably receive a lot of incentive from their owners, which inevitably results in a faster maturation when compared to their counterparts.

## 5 Conclusion

This paper presented an updated performance evaluation of three popular linear algebra libraries at computing the sparse matrix-vector (SpMv) product using different storage schemes. The

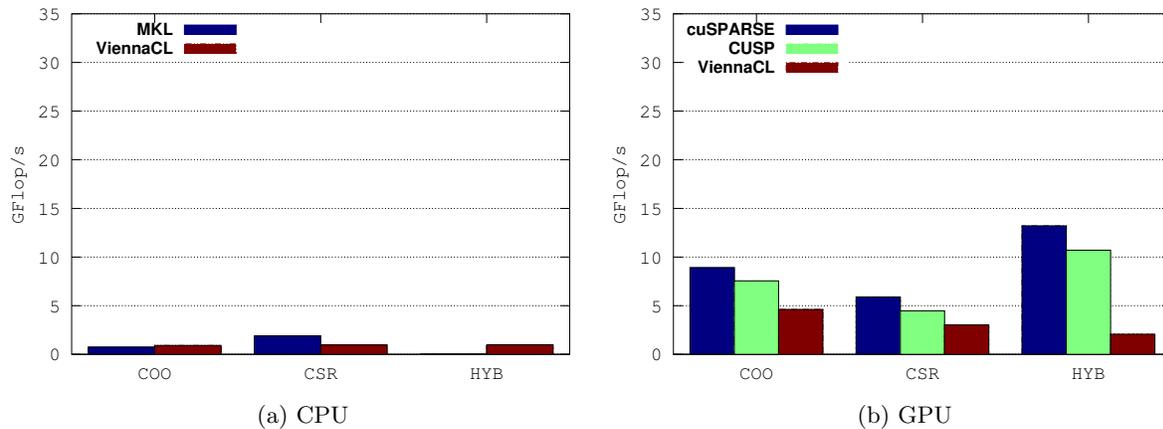


Figure 5: SpMv: *Webbase* matrix, double-precision.

SpMv operation is central in a wide range of applications but at the same time hard to achieve high efficiency on parallel processors due mainly to the difficult in generating regular work. Given our experimental setup and the set of benchmark matrices utilized we can conclude that, firstly, a high-end GPU processor greatly surpasses the performance of a high-end CPU; secondly, out of the considered sparse matrix formats, the CSR is the fittest for the CPU whereas the HYB format is the best suited for the GPU; and finally, Intel MKL and NVIDIA cuSPARSE are, respectively on the CPU and GPU, currently the most performing libraries for SpMv product. On the other hand, due to the fact that the CUSP and ViennaCL libraries are open-source projects—and the latter, on top of that, portable across different architectures and vendors—neither Intel MKL nor NVIDIA cuSPARSE dominate them, despite being better when the objective is the raw throughput.

As future work we are particularly interested in how different library-format combinations scale when tackling the SpMv product on real-world matrix structures and patterns that arise from reservoir simulations.

## References

- [1] Nathan Bell, Michael Garland, Efficient Sparse Matrix-Vector Multiplication on CUDA, *NVIDIA Technical Report NVR-2008-004*, 2008.
- [2] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington, A Sparse Matrix Library in C++ for High Performance Architectures, *Proceedings of the Second Object Oriented Numerics Conference*, pp. 214-218, 1994.
- [3] Ruipeng Li, Yousef Saad, GPU-accelerated preconditioned iterative linear solvers, *The Journal of Supercomputing*, 2013.
- [4] Mhd. Amer Wafai, Sparse Matrix-Vector Multiplications on Graphics Processors, *Master Thesis in Information Technology*, High Performance Computing Center Stuttgart (HLRS), 2009.
- [5] 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. 2007 ACM/IEEE Conference on Supercomputing*, 2007.